# Модели параллельного программирования

И.Е. Федотов

2011

Книга посвящена рассмотрению некоторых высокоуровневых моделей параллельного и распределенного программирования. В порядке усложнения описываются несколько моделей внутренней организации параллельных программ: ярусно-параллельная форма программы, сети конечных автоматов, сети Петри, модель актеров, а также модель квантовых вычислений. Приводятся примеры программной реализации на C++ с использованием различных средств распараллеливания (OpenMP, MPI, POSIX Threads, Windows API). В каждом случае рассматриваются вопросы контекстно-независимой реализации конструкций описываемой модели без привязки к конкретным задачам, а также приведены примеры решения с использованием такой реализации некоторых конкретных задач. Некоторые из описанных моделей (к примеру, модель актеров), в настоящий момент приобретают все большую популярность вследствие распространения основанных на ее использовании языков и библиотек.

Книга ориентирована на подготовленного читателя в области программирования. Будет полезна программистам, желающим освоить высокоуровневые подходы к организации параллельных и распределенных программ, студентам старших курсов, аспирантам и преподавателям технических ВУЗов, преподающим параллельное программирование.

# Оглавление

# Предисловие

Вот уже несколько лет, как современные компьютеры вплотную приблизились к своему пределу тактовых частот. Производительность процессоров (точнее, их отдельных ядер) перестала нарастать сказочными темпами, и теперь остается возможность обеспечить дальнейшее повышение вычислительных мощностей лишь за счет увеличения их количества: производительность компьютеров начала расти «вширь», а не «в высоту». В связи с этим параллельное программирование уже перестало быть узкоспециализированной дисциплиной для высокопроизводительных вычислений и приобретает все большую актуальность для широких масс представителей программистского сообщества. Есть немало публикаций, посвященных рассмотрению архитектур параллельных вычислительных систем и теоретическим вопросам распараллеливания последовательных алгоритмов. Также существует достаточно учебных пособий и статей, посвященных рассмотрению существующих технологий, программных интерфейсов и библиотек. В то же время, довольно давно создано немало моделей, успешно описывающих выполнение параллельных процессов, которые уже являются основой многих программных технологий и некоторых не слишком популярных инструментов построения параллельных программ. Таким моделям посвящено немало учебного материала теоретического характера, однако ощущается некоторый дефицит в области практического их рассмотрения с доступными примерами программной реализации на основе современных средств. В результате освещения таких моделей лишь с теоретической точки зрения они, порой, остаются вне практического арсенала программиста, отчего возможность использования их при решении конкретной задачи зачастую даже не рассматривается. Настоящее издание стремится в какой-то степени восполнить указанную нишу и попытаться исправить такое положение.

## О проблеме параллельного программирования

Главная проблема параллельного программирования основана на относительной сложности построения схемы параллельных вычислений в голове программиста. Человек в принципе мыслит последовательно. Здесь, разумеется, имеется в виду процесс построения умозаключений, а не внутренние процессы мозга на физиологическом уровне. Именно поэтому последовательны большинство существующих сегодня языков программирования. Многие авторы, работающие в области параллельного программирования, сходятся во мнении, что развитой дисциплины параллельного программирования на сегодняшний момент нет, и что индустрия создания параллельных программ движется по тупиковому пути [21, 25, 24, 68].

Большинство сегодняшних параллельных вычислений реализуется в виде параллельно-последовательных программ. При этом в существующем последовательном алгоритме выделяются независимые друг от друга последовательные ветви, которые могут быть выполнены параллельно, и с этим учетом пишется параллельно-последовательная программа. Нередко берется уже существующая последовательная программа, которая пу-

тем добавления неких конструкций распараллеливания преобразуется в параллельно-последовательную. Чем сложнее исходная последовательная программа, чем хуже она структурирована, чем запутаннее зависимости между отдельными ее ветвями, тем сложнее оказывается выполнить ее распараллеливание.

Случаются ситуации, когда существующая последовательная программа реализует некоторый алгоритм, в котором изначально присутствует логический параллелизм, однако реализация этого алгоритма в программе обременена сложным взаимодействием участков кода, которые в идеале могут и должны быть независимыми. В таком случае бывает сложно без существенных изменений программы выделить в ней ветви, выполнение которых может быть физически распараллелено. Использование моделей параллельного программирования, в том числе рассматриваемых в настоящем издании, обеспечивает основу для такого описания задач, при котором присущий им логический параллелизм выявляется изначально и не теряется в дальнейшем.

Существует немало изданий, посвященных интерфейсам и технологиям параллельного программирования, наподобие рассматриваемых ниже MPI и OpenMP. Однако это лишь инструменты, а одного прекрасного знания инструмента, как правило, недостаточно для того, чтобы легко и эффективно реализовать решение конкретной задачи. Помимо знаний о том, какой инструмент нужно использовать и каковы правила его использования (непосредственные требования спецификации), необходимы знания о том, какие цели должны быть при этом достигнуты и, самое главное, каким образом инструмент должен помочь в достижении этих целей (идиоматические конструкции, образцы (patterns) решения типовых задач и т.п.). Именно такую роль в параллельном программировании выполняют соответствующие вычислительные модели.

Многие авторы публикаций по параллельным вычислениям сходятся в том, что для описания параллельных программ необходимо уйти от императивных языков программирования [10, 12]. Императивные языки (от лат. imperativus — повелительный) описывают, какие действия и в каком порядке следует выполнить, чтобы получить результат. Такими является большинство популярных сегодня языков: C++, Pascal, Java, Perl и т.п. В данном же случае требуется использование декларативных языков (от лат. declaratio — заявление, объявление), т.е. таких, которые описывают, чем является результат, который должен быть получен, оставив выполнение действий, как и выбор их последовательности, на долю компилятора или интерпретатора. Примерами декларативных языков являются язык логического программирования Prolog и языки функционального программирования Haskell, Erlang, Lisp и т.п. Различие между императивным и декларативным программированием в чем-то напоминает различие между прямым вычислением функции и решением уравнения, т.е. обращением функции.

С момента становления программирования как дисциплины декларативным языкам пророчили большое будущее, поскольку они зачастую позволяют легко, наглядно и потому надежно описать задачу, что само по себе обычно гораздо проще, чем детально описывать процесс ее решения. При этом возникает другая сложность: создание эффективного компилятора или интерпретатора, который, в общем-то, и принимает в этом случае бремя принятия решений о выполнении действий, снятое с плеч программиста.

Несмотря на потенциальную мощь, декларативные языки не приобрели широкой популярности. Одна из возможных причин заключается в том, что на момент их появления мощности машин не хватало для реализации достаточно эффективных компиляторов и интерпретаторов. Вследствие этого программисты были вынуждены пользоваться языками, которые позволяют «подсказать», а вернее даже «повелеть» машине выполнение конкретных действий, т.е. императивными языками.

Ситуация здесь сродни той, почему в прежние времена было популярно использование

низкоуровневых языков. Оптимизирующие компиляторы языков высокого уровня и вычислительные системы, на которых они работали, не были достаточно мощными, чтобы обеспечить производительность и размер программ, сравнимые с программами на языках низкого уровня. Именно это является одной из причин, по которым Д. Кнут избрал машинно-ориентированный язык для представления алгоритмов в своем известном многотомнике, о чем он и говорит в самом начале первого тома. Когда оптимизаторы кода достигли приемлемого уровня качества, программирование на языках низкого уровня постепенно утратило популярность в угоду повышению переносимости. Однако переход на языки высокого уровня не потребовал смены парадигмы, и языки остались императивными. Смена языка в рамках одной парадигмы гораздо проще смены парадигмы, к тому же для перехода на языки высокого уровня был стимул: повышение переносимости.

Развитие вычислительных мощностей сказалось также и на повышении эффективности использования компиляторов и интерпретаторов декларативных языков. Более того, в отношении смены парадигмы на декларативную теперь также появился стимул: перенос реализации механизмов распараллеливания с программистов на компиляторы и интерпретаторы. Такие механизмы являются частью системы, и забота о них не должна лежать на плечах прикладного программиста. Поэтому можно надеяться, что в скором времени изменится и отношение к декларативному программированию.

Есть мнение, что вместо освоения новых языков программирования необходимо пополнение конструкциями распараллеливания традиционных последовательных языков (императивных), поскольку первое гораздо дороже обходится с точки зрения переобучения персонала. Подобные аргументы допустимы лишь в рамках корпоративной политики отдельно взятых производств, в общем же случае они попросту встают на пути прогресса. Нельзя не признать, однако, что какой-то переходный этап на пути от императивных языков к декларативным неизбежен. Существует и другое мнение, утверждающее, что выбор императивных языков был изначально неверным, поскольку вынуждает задавать в программе больше деталей, чем необходимо для представления алгоритма, тем самым усложняя процесс описания решаемой задачи программистом компьютеру. Задание четкой последовательности действий в ситуации, когда она не важна, — одна из таких «ненужных» подробностей, присущая императивным языкам.

Многие могут возразить: зачем использовать декларативные языки, если машина все равно выполняет императивный поток инструкций? С тем же успехом можно спросить, зачем вообще использовать языки высокого уровня, если можно программировать сразу в машинных инструкциях. Машина «мыслит» совершенно не так, как человек, и здесь не возникает разногласий. Человеку нелегко дается описывать задачу в терминах машины. Высокоуровневое же программирование призвано быть как раз промежуточным слоем, обеспечивающим преобразование описания задачи, представленного в терминах человека (на формальном, но все же приближенном к представлению человека языке), в описание той же задачи, представленное в терминах машины. Чем проще, естественнее и точнее высокоуровневый язык позволяет человеку описать свою задачу, тем лучше он соответствует своему назначению. В этом отношении декларативные языки, определенно, во многих областях опережают императивные.

Многие существующие уже давно языки декларативного программирования, несмотря на свою мощь, не обеспечивают полноценной базовой платформы для удобного описания программы с возможностью неявной передачи присущей ей внутренней параллельной структуры. Причина, вероятно, в том, что спецификации этих языков создавались тогда, когда вопрос параллельного программирования не стоял так остро, как сегодня, и, видимо, поэтому в них зачастую содержатся ограничения, также сводящие вычисления к последовательным. К примеру, наличие в языке Lisp операции присваивания вносит необходимость

задания четкого порядка для вычисления последовательности выражений (к примеру, аргументов функции), что лишает интерпретатор права распараллелить этот процесс, даже если программисту этот порядок не важен [10].

Выходит, для полноценного удобного описания параллельных программ, так или иначе, требуется более современный язык. Возможно, им станет один из уже существующих языков, попыток создания которых, в том числе весьма удачных, на сегодняшний момент известно немало. Среди них, к примеру, мультипарадигменный язык Oz. Как и в случае многих других языков, нельзя сказать, что параллельное программирование является первостепенной его задачей, однако удобство и эффективность, с которыми оно выполняется, является естественным следствием выразительности соответствующих декларативных языковых конструкций. Параллельное выполнение в Oz основано на легковесных потоках управления (green threads) и элементах модели потоков данных (dataflow), близкой к рассматриваемому ниже ярусно-параллельному представлению программы. Функциональный язык Erlang изначально создавался с целью построения распределенных систем, в связи с чем предоставляет не менее удобные конструкции для создания параллельных программ, а также высокую степень эффективности их выполнения. Модель существования и взаимодействия параллельно выполняющихся процессов в Erlang близка к модели актеров, которая также будет рассмотрена ниже. Многие языки, такие как, к примеру, Alice, используют понятие фьючеров (future, будущее значение), которые напоминают переменные и представляют собой результат выполнения каких-либо еще не завершенных операций, выполняемых обычно параллельно. Наконец, существуют языки, которые сами по себе не предоставляют явно конструкций для распараллеливания, однако мощь и выразительность их синтаксиса позволяет создание для этих целей специальных библиотек, удобство и прозрачность использования которых оказывается сравнима с использованием языковых конструкций. Таким является мультипарадигменный язык Scala, компилируемый в байт-код для виртуальной машины Java.

Эти и другие подобные языки, несмотря не свои достоинства, пока не обрели достаточно широкой популярности, и можно лишь надеяться, что они обретут ее в будущем. В противном случае можно надеяться, что появятся и обретут широкую популярность другие высокоуровневые языки для описания параллельных программ. Пока же мы будем отталкиваться от факта их отсутствия, в связи с чем будем рассматривать популярные модели параллельных вычислений с примерами на основе популярного императивного языка.

Следует понимать, что под прозвучавшим утверждением об отсутствии популярных языков параллельного программирования подразумевалось отсутствие таких языков, которые не ориентированы на какую-либо узкоспециализированную область применения и которые используются для реализации параллельных вычислений повсеместно. Стоит отметить, что популярность средства программирования очень важна в сегодняшних условиях, когда большинство программистов вынуждены быть «совместимыми» друг с другом. Временные затраты на освоение многочисленных, хоть и прогрессивных, но зачастую мало распространенных технологий, как правило, не оправдывают себя, вследствие чего большинство программистов вынуждены консервативно пользоваться решениями с использованием, порой, не самых удачных, но устоявшихся средств.

## О целях издания

Представленное в книге описание моделей параллельного программирования, помимо очевидных академических целей, стремится проиллюстрировать следующие два взаимосвязанных момента.

Во-первых, ставится задача показать проблему современного параллельного программирования, заключающуюся, прежде всего, в том, что наиболее популярные сегодня средства программирования приспособлены к описанию методов решения стоящей задачи в соответствии с образом мышления программиста, а именно в последовательной форме. Отсюда вытекает сложность наглядного для человека представления с использованием таких средств параллельных алгоритмов и программ. В связи с этим встает ребром вопрос необходимости использования для параллельного программирования принципиально иных средств, не являющихся модификациями или надстройками существующих популярных языков. Одним из возможных направлений является декларативное программирование, что косвенно и демонстрируется примерами в книге, поскольку приведенные программы, хоть и реализованы на императивном языке, зачастую основаны на декларативном представлении задачи.

Во-вторых, делается попытка проиллюстрировать следующий факт, вытекающий из рассмотрения тех же обстоятельств с обратной стороны. Несмотря на указанные сложности, императивные языки, все же, могут без серьезных препятствий быть использованы для прозрачной (неявной) реализации параллельных алгоритмов. При этом может быть реализована некая абстрактная машина, принимающая на входе набор декларативно описывающих алгоритм данных, которая инкапсулирует внутри себя механизмы параллельного выполнения. Программный интерфейс такой машины принципиально не зависит от средств распараллеливания, с помощью которых она реализована и которые в общем случае могут быть разными. Такой подход позволит не уходить от имеющейся на сегодняшний момент ситуации преобладания императивных языков и, тем самым, в какой-то степени упростит и сгладит освоение параллельного программирования.

Оба этих момента взаимно дополняют друг друга. Учитывая отсутствие на сегодняшний момент развитых и одновременно обладающих широкой популярностью средств декларативного параллельного программирования, в практических целях может быть рассмотрен подход с использованием механизма обработки декларативных данных, реализованного на императивном языке. Однако в целом такие меры являются вынужденными и требуют дополнительных затрат, вследствие чего в будущем требуется появление новых либо популяризация существующих более мощных средств разработки, изначально ориентированных на параллельное программирование и не являющихся видоизменением каких-либо средств последовательного программирования.

Разумеется, ни одна из представленных в настоящем издании моделей не является панацеей и не дает возможности простого описания для любой задачи, вопреки убеждениям некоторых их приверженцев. Одни задачи наиболее удобно описываются одной моделью вычислений, другие — другой, а третьи наиболее просто и наглядно описываются простым циклом на императивном языке. Тем не менее, понимание рассмотренных моделей и парадигм в комплексе может существенно упростить разработку параллельных программ с той точки зрения, что даст возможность программисту смотреть шире и сгладит отсутствие у него «параллельного мышления». Готовые примеры программ «со стороны», наподобие приведенных здесь, практически никогда не оказываются подходящими во всех отношениях, однако это лишь добавляет стимула для более глубокого самостоятельного изучения соответствующей темы.

## О содержании

Факт наличия параллелизма среды выполнения порождает необходимость распараллеливания ресурсоемких вычислений, однако сама по себе эта необходимость в общем случае не привязана к конкретной реализации параллельной среды. В связи с этим в книге не при-

водится классификация параллельных вычислительных систем и не делается акцента на конкретную категорию системы. Для ознакомления с классификацией можно обратиться к другой литературе [1, 4, 9, 15].

Также в настоящем издании не рассматриваются специализированные языки параллельного программирования, такие как, например, occam, или языки с естественной поддержкой параллельного программирования, наподобие Erlang или Oz. Здесь, напротив, делается попытка познакомить читателя с существующими подходами и методами параллельного программирования на основе уже известных ему конструкций. В частности, с использованием известного языка. Для нашего изложения оказался наиболее удобен язык C++, хотя вообще выбор языка непринципиален.

Наконец, поскольку эта книга также не предназначена для первоначального ознакомления с основами программирования, она не содержит обзора используемого в примерах языка. Для понимания изложенного материала необходимо хорошее владение языком, и именно по этим причинам выбран достаточно популярный язык программирования. По тем же причинам нигде в тексте не объясняются правила и принципы работы со стандартными контейнерами библиотеки STL, хотя они широко используются в примерах программ.

Реализация как такового механизма распараллеливания для большинства программ приводится с использованием нескольких различных программных интерфейсов. При этом делается попытка максимально сохранить программный интерфейс, предоставляемый клиентскому коду, чтобы избежать изменений в последнем при переходе от одного способа реализации параллелизма к другому. Такой подход выбран с целью показать, что процесс написания параллельных программ, хоть и привязан во многом к удобству предоставляемых инструментов, все же упирается не в выбор конкретного инструмента и степень владения им, а в правильный выбор представления параллельной программы, наиболее полно соответствующий стоящей задаче.

Поскольку полностью уйти от вопросов реализации нельзя, в главе 1 приводится краткий обзор двух популярных интерфейсов параллельного программирования с примерами программ. Однако за более подробным описанием этих и других программных интерфейсов следует обращаться к спецификациям [71, 74] и специально посвященным им изданиям [2, 3, 4, 46].

В остальных главах рассматриваются несколько моделей параллельного программирования, существующих сегодня и, порой, выдвигаемых их приверженцами в качестве универсальной основы для построения параллельных программ. Рассматриваемые модели являются по существу разными, и в то же время во многом напоминают друг друга, что дает возможность неформально говорить о некоторых обобщениях и усовершенствованиях одной модели для формирования другой. Возможные подходы к построению параллельных программ не ограничиваются рассмотренными вариантами. К примеру, в силу довольно специфичных областей применения здесь не рассмотрены модель PRAM (Parallel Random Access Machine), нейронные сети, клеточные автоматы и т.п.

Глава 2 посвящена рассмотрению ярусно-параллельной формы программы, базовому представлению в виде направленного ациклического графа, в котором выявляются наборы информационно независимых друг от друга операций. Модель предполагает, что все рассматриваемые операции выполняются единожды, поэтому граф не может содержать циклических зависимостей, поскольку ни одна операция не должна зависеть сама от себя. Также вследствие однократного выполнения операций отсутствует необходимость хранения ими промежуточного состояния.

В главе 3 рассматриваются сети конечных автоматов. Все автоматы сети работают синхронно по тактам, при этом предполагается, что тактов в общем случае много. Как следствие, в таких сетях появляется возможность циклических зависимостей автоматов, по-

скольку выходные данные автомата могут прямо или косвенно вернуться на вход ему же. В этих условиях оказывается довольно уместным наличие у автомата внутреннего состояния, т.к. это существенно повышает удобство моделирования ими объектов реального мира. Таким образом, сети конечных автоматов в некотором роде обобщают модель, рассматриваемую в главе 2, являясь, по сути, расширением ярусно-параллельной формы высотой в один ярус наличием обратных связей и хранением промежуточного состояния элементов.

Сеть Петри, описанию которых посвящена глава 4, в свою очередь, можно считать некоторым обобщением конечного автомата, который может находиться одновременно в нескольких состояниях и в общем случае перестает быть конечным.

Модель актеров, рассматриваемая в главе 5, обобщает уже сети конечных автоматов, снимая некоторые их ограничения, такие как необходимость глобальной синхронизации и четкий порядок поступления данных, и тем самым добавляя новые возможности.

Глава 6 имеет принципиально иной характер. Она посвящена квантовым вычислениям — альтернативной модели вычислений, которая, в отличие от классической, изначально базируется на наличии естественного параллелизма. К сожалению, пока эта модель полезна лишь на бумаге. Практическую выгоду от ее использования на текущий момент получить затруднительно вследствие отсутствия соответствующей аппаратуры. Однако сама по себе эта вычислительная модель обладает некоторой завораживающей красотой, и знакомство с ней, безусловно, будет полезно для заинтересованного читателя.

При рассмотрении каждой модели общие механизмы построения ее базовых конструкций и параллельного выполнения независимых ветвей отделяются от специфики решаемых задач. В результате в рамках всех описанных моделей рассматриваемые задачи представляются декларативно: сетевым графиком работ, диаграммой состояний автомата, схемой автоматной сети, схемой сети Петри, схемой взаимодействия актеров и набором описаний их поведения. Параллельное выполнение декларативно описанной программы зависит от внутренней реализации конкретного механизма обработки декларативных данных, а не возлагается на плечи программиста, использующего этот механизм.

Реализация соответствующего механизма может быть как последовательной, так и параллельной с использованием различных подходов к организации физического параллелизма. В нашем случае строится универсальный (контекстно-независимый) набор классов, реализующих конструкции рассматриваемой модели, который может быть использован для решения широкого круга задач, и в том числе используется в приведенных примерах. Решение конкретных задач сводится к использованию этих классов в коде, описывающем задачу в терминах выбранной модели. Осуществление же распараллеливания выполнения отдельных ветвей инкапсулируется внутри этих классов, что позволяет упростить написание клиентского кода и тем самым повысить его надежность. Для каждой рассматриваемой модели такой набор контекстно-независимых классов вынесен в приложение. Примеры их использования, а также варианты изменений с целью смены механизма распараллеливания, обсуждаются в тексте главы, посвященной соответствующей модели.

## Об используемой терминологии

Оговорим некоторые используемые в дальнейшем термины, трактование которых в существующей литературе зачастую различается. К примеру, в одних источниках термин «процессор» используется для указания отдельного последовательного вычислительного устройства (CPU), в других — для указания отдельного узла вычислительной сети. Здесь термин «процессор» будет использоваться в качестве описания отдельного последовательного вычислительного элемента системы с общей памятью, независимо от того, является ли он на самом деле отдельным процессором многопроцессорного сервера, ядром много-

ядерного процессора или элементом вычислительной системы с организацией доступа к общей памяти через коммутатор. Условимся также называть «узлом» отдельный элемент вычислительной системы с распределенной памятью. Это может быть элемент системы с массовым параллелизмом или же просто отдельная машина, работающая в составе вычислительного кластера.

Также следует оговорить некоторые стилистические расхождения в терминах, хотя они не столь принципиальны. Так сложилось, что отдельные параллельные ветви выполняющегося процесса (threads) многие называют потоками (а также тредами, потоками команд, потоками управления, но чаще просто потоками). Есть авторы, которые настойчиво используют термин «нить», поскольку именно «нить» соответствует английскому слову «thread», в то время как «поток» соответствует слову «stream» (а также в некоторых областях слову «flow» — «flow-based programming»), и во избежание путаницы в русскоязычной терминологии следует разделять эти понятия. Безусловно, они правы. Однако здесь приходится учитывать, что на текущий момент для обозначения этого понятия гораздо более широко распространен термин «поток», и с целью общения на едином языке большинству программистов приходится мириться с некорректностью терминологии. В силу того, что этот момент не имеет принципиального значения для дальнейшего изложения, мы будем использовать термин «поток», поскольку по исторически сложившимся причинам этот термин оказывается более привычным для большинства программистов.

Заодно коснемся перевода слова «handle», на предмет которого терминология также не устоялась. Зачастую в литературе хэндлы называют дескрипторами (или еще проще — описателями). Эта терминология некорректна, по той простой причине, что хэндл — не дескриптор. В UNIX-системах, откуда происходит этот термин, понятие файлового дескриптора отождествляется с индексом (номером) соответствующей структуры в таблице файловых дескрипторов процесса, записи в которой, действительно, «описывают» открытые файлы. Записи этой таблицы недоступны программе, однако она может определенным образом ориентироваться на значения индексов при построении своей внутренней логики. Хэндлы же не только ничего не описывают, но и не являются индексом, который может быть как-либо интерпретирован программой. Хэндл является неким «черным ящиком» («темный тип», opaque type), значение которого может быть использовано только в качестве ссылки на объект при обращении к другим функциям системы. Таким образом, хэндл не дает никакой информации и лишь, как и диктует нам дословный перевод, является «ручкой», посредством которой можно так или иначе манипулировать соответствующим объектом. В связи с этим самым удачным, пожалуй, переводом этого слова является «манипулятор», однако этот термин не снискал популярности среди программистов, в связи с чем нами будет использован наиболее простой и популярный вариант — «хэндл».

Разделяют понятия логического и физического параллелизма [15]. В случае наличия логического параллелизма в задаче могут быть выделены независимые подзадачи, которые вследствие своей независимости в принципе могут решаться параллельно. При этом физически их выполнение может производиться как угодно, в том числе последовательно или псевдопараллельно («как будто параллельное» выполнение в последовательной среде). В случае физического параллелизма подразумевают, что задача физически должна выполняться параллельно в некоторой существующей параллельной вычислительной среде. Очевидно, физическое распараллеливание выполнения различных операций возможно лишь для алгоритмов, обладающих логическим параллелизмом. Если решаемая задача не обладает ярко выраженным логическим параллелизмом, внесение физического параллелизма может являться в некотором роде неестественным, сопровождаться сложностью программной реализации и не дать в результате положительного эффекта.

Поскольку изложение зачастую не будет привязано к реализации физического парал-

лелизма, помимо прочего нам потребуется универсальный термин, которым мы будем обозначать некий ресурс, в рамках которого выполняется отдельная последовательная ветвь программы. Будем в дальнейшем называть такую сущность параллельным ресурсом. В зависимости от контекста этот термин может означать поток многопоточного процесса или последовательный процесс, выполняющийся на узле распределенной системы в рамках работы многопроцессной программы.

Наконец, оговорим часто встречающееся в изложении понятие «клиентский код», хотя оно является общепринятым и, скорее всего, упоминания не требует. В широком смысле это понятие обозначает код, использующий некоторый предоставляемый какой-либо системой или библиотекой программный интерфейс. Выше говорилось, что код большинства приводимых программ разделен на общие контекстно-независимые классы и некий код, реализующий конкретную задачу и использующий эти классы. Именно последний, в соответствии с указанным выше значением, и будем в дальнейшем подразумевать под понятием «клиентский код», противопоставляя его коду общих классов, реализующих рассматриваемую модель.

## Некоторые вопросы стиля

Приводимые в книге программы, как правило, не являются полноценными программами для компиляции, а являются лишь фрагментами кода, иллюстрирующими излагаемый материал. Для возможности компиляции эти фрагменты должны быть дополнены стандартными конструкциями, с которыми читатель должен быть знаком (обрамление в функции, объявление констант, включение заголовочных файлов, доступ к пространствам имен и т.п.).

Код умышленно упрощен и не претендует на промышленное качество (к примеру, зачастую чрезмерно упрощен или же вовсе опущен контроль ошибок среды выполнения). С другой стороны, приведенный код, местами, содержит кажущиеся излишними в реальной программе усложнения, внесенные для наиболее полного, по возможности, соответствия программы реализуемой модели. Это сделано в иллюстративных целях, порой, в ущерб эффективности. Для реальных приложений подобная педантичность в большинстве случаев может оказаться излишней, в связи с чем читателю следует расценивать приводимые программы в качестве иллюстрации, из которой каждый вынесет лишь необходимое.

### Оформление кода

Стоит оговорить некоторые вопросы, касающиеся оформления приводимых программ. Многие детали их оформления, как всегда и бывает, несут в себе некоторую долю субъективного подхода. Стиль написания программ — вопрос неоднозначный. Есть моменты, с которыми безусловно соглашаются все, такие как, к примеру, необходимость наличия отступов в каждом вложенном блоке на языках C, C++ и им подобных. Есть и другие моменты, некоторые из которых и по сей день в иных кругах являются предметом споров, такие как, например, размещение открывающей фигурной скобки на отдельной строке или в конце предыдущей. В таких вопросах ключевым становится не конкретный выбор того или иного подхода, а сам факт его выбора и единообразное следование ему. Иначе говоря, есть моменты, отражающие не степень профессионализма, а просто личный выбор программиста.

Автор, разумеется, со временем также выработал для себя свой стиль и придерживается его. Этот стиль содержит немало довольно субъективных правил, для выработки которых были свои не менее субъективные причины. Автор ни в коем случае не пытается навязать

свой стиль или свое мнение на предмет корректности или необходимости использования тех или иных правил оформления, он их просто придерживается. Читателю же следует просто игнорировать раздражающие его детали оформления, поскольку книга, в общем-то, не о них.

Тем не менее, кратко обозначим, все же, причины возникновения некоторых подобных правил, дабы у читателя возникало меньше вопросов на этот счет. Большинство правил возникли как попытка повышения личного удобства автора при чтении своих же программ. Сюда относится довольно распространенное использование `::` перед вызовами системных функций, а также совсем не распространенное использование `private:` в начале описания класса. Последнее — чтобы не смотреть выше и не вникать, класс это или же структура. Также весьма спорной и совершенно не распространенной является манера ставить точку с запятой после составных операторов, ограниченных фигурными скобками. Здесь причиной является то, что в языке C++, в отличие от, к примеру, Java, требуется наличие точки с запятой после некоторых конструкций, после которых ее писать интуитивного стремления нет (в частности, после объявления класса). Этот факт сам по себе порождает манеру писать точку с запятой после каждого логически завершенного оператора или объявления. Помимо этого, таким образом отчасти повышается читабельность в контексте зрительного поиска завершенного оператора. К примеру, после составного оператора в `if` перед `else` точку с запятой поставить не удастся, что логично: это один оператор. То же касается тела цикла `do {...} while`. Поэтому, ставя точку с запятой всегда, когда допускает компилятор, мы гарантированно получаем их наличие только на границах операторов. Исключением из этого правила являются тела функций, после которых точку с запятой можно не ставить никогда, что позволяет быстро зрительно отличать конец тела функции от конца составного оператора.

Какие-то правила возникли по той причине, что некоторые наборы функций, как часто бывает, перемещаются между исходными текстами на разных языках. В частности, это касается явного указания `void` для функций без параметров. В языке C указание пустых скобок при объявлении функции позволяет ее вызов с произвольным списком параметров, тогда как `(void)` означает пустой список. В языке C++ и пустые скобки, и `(void)` означают одно и то же: пустой список. По этой причине в целях единообразия кода в обоих языках автор использует явное указание `void` для пустого списка.

### Обработка ошибок

Ошибки, возникающие во время выполнения программы, можно грубо разделить на две основные категории: ошибки среды выполнения (неверные данные на входе программы, сбои системы или оборудования и т.п.) и ошибки программирования (некорректный алгоритм, опечатки, недопустимые операции). В первом случае при возникновении ошибки программа должна обеспечить ее корректную обработку и, возможно, продолжить работу дальше. Во втором случае, как правило, программа вообще не должна эксплуатироваться. Именно для выявления ошибок такого типа существует конструкция `assert`.

Конструкция `assert` широко используется в примерах, приводимых в книге, для контроля ошибок программирования. Однако, поскольку в задачи настоящего издания не входит демонстрация принципов обработки ошибок, которым уже посвящено немало литературы, практически нигде не выполняется контроль ошибок среды выполнения. В редких случаях такой контроль, все же, осуществляется, для чего снова используется конструкция `assert`. Следует иметь в виду, что это можно считать приемлемым для иллюстративного кода (хотя, тоже вопрос спорный), в реальных же промышленных системах это совершенно непозволительно. Также необходимо помнить, что содержимое конструкций `assert`

отключается с помощью макроопределения NDEBUG и обычно не попадает в отлаженную программу, вследствие чего внутри них нельзя писать код с побочным эффектом (к примеру, вызовы функций, необходимые по логике программы). Именно по этим причинам проверка значений, возвращаемых функциями pthreads, выносится в отдельную функцию chkzero.

### Виртуальные деструкторы

Наконец, в коде часто используются абстрактные классы, однако в них нет виртуальных деструкторов. Многим это покажется странным, неестественным и даже некорректным. Более того, многие яростно выступили бы против такого подхода, поскольку считается, что в таких классах наличие виртуального деструктора необходимо. Однако этот подход автор для себя выбрал умышленно с целью «повышения дисциплины». Следует понимать, для чего именно необходимо наличие виртуального деструктора. А необходимо оно, в самом деле, лишь для одной ситуации: для уничтожения объекта без информации о его реальном типе. Наличие такой возможности в некотором роде расслабляет программиста, позволяя уничтожение объекта на ином уровне, нежели тот, где он был создан, и нарушая тем самым концептуальный подход «кто породил, тот и убил». При этом оказывается, что для других целей, кроме как уничтожение объекта из произвольной точки программы, виртуальный деструктор, как правило, и не нужен вовсе, или же без него легко обойтись.

Разумеется, это является вопросом стиля написания программ и потому вопросом спорным. Не исключено, что многим окажется крайне затруднительным программирование без возможности уничтожения объекта без информации о его реальном типе. Для автора же отсутствие такой возможности не является большой жертвой, поскольку он старается ей вообще не пользоваться.

Автор склонен придерживаться практики, при которой уровень, ответственный за создание объекта (к примеру, фабрика), также ответственен и за его уничтожение. Это, во-первых, дисциплинирует в области структурирования программы, во-вторых, дает возможность принимать решение об уничтожении объекта той стороне, которая его создавала. В частности, это позволяет эффективную реализацию фабрики с инкапсулированным кэшированием объектов, когда схожие объекты создаются и уничтожаются часто, а накладные расходы на эти операции высоки. В этом случае фабрика может вместо уничтожения объектов сохранять их в списке свободных, после чего отдавать клиентскому коду в момент вызова операции создания.

Также распространенным является альтернативный подход, подразумевающий наличие в абстрактном классе метода, выполняющего уничтожение. К примеру, метод Release, уничтожающий объект при обнулении счетчика ссылок. Виртуальный деструктор в таком абстрактном классе также не является необходимым, поскольку в этом случае уничтожение производится на том уровне, где так или иначе известен реальный тип объекта.

### Беззнаковые целочисленные типы

В приводимом коде крайне редко используются беззнаковые типы. Причин несколько. Во-первых, директива for интерфейса OpenMP ранних версий применима только к циклам со знаковым целочисленным параметром. Во-вторых, практически все целочисленные параметры интерфейса MPI, в том числе заведомо неотрицательные, представлены знаковыми типами. В-третьих, необдуманное использование беззнаковых чисел совместно со знаковыми вносит некоторое рассогласование в программу и является причиной, порой, довольно неожиданных ошибок. Разумеется, совместное корректное использование знаковых

и беззнаковых величин возможно, однако требует повышенной внимательности и аккуратности при анализе и контроле возможных ситуаций.

Знаковые и беззнаковые типы имеют в корне разные цели, каждый из них имеет свои достоинства и недостатки в различных ситуациях. Бесспорным преимуществом беззнаковых является то, что по спецификации C++[1] работа с ними всегда ведется в соответствии с правилами модульной арифметики. Т.е. выход за границы соответствующего диапазона в результате, к примеру, сложения двух беззнаковых чисел невозможен принципиально, поскольку сложение модульное. В то же время, выход за границы диапазона знакового числа (переполнение, overflow) по спецификации приводит к неопределенному поведению программы (undefined behaviour). Это связано с тем, что представление знаковых чисел и соответствующая реализация знаковой арифметики зависит от аппаратуры, и спецификация не налагает требований на внутренние правила ее выполнения. Такое положение вносит сложности создания переносимого кода, корректно работающего с большими значениями знаковых типов. С другой стороны, полностью переносимая знаковая арифметика для больших чисел может быть реализована «вручную» на базе модульной арифметики беззнаковых типов (к примеру, путем представления отрицательных чисел беззнаковыми в дополнительном коде).

Таким образом, беззнаковое число представляется неким универсальным низкоуровневым примитивом с четко определенной семантикой и битовым выражением, предоставляющим возможность полного контроля над содержимым соответствующей переменной. Знаковое же число является более высокоуровневым автоматизированным объектом, контроль над которым осуществляет аппаратура.

Неправильное использование беззнаковых чисел обычно возникает вследствие их не вполне корректной интерпретации программистом. Зачастую беззнаковыми объявляются переменные, значения которых просто не могут быть отрицательными. В этой ситуации возникает неоднозначное трактование разности двух беззнаковых чисел. Одни люди уверены, что разность двух беззнаковых чисел должна быть знаковой. Другие уверены, что результат разности двух беззнаковых должен быть также беззнаковым. Правы и те, и другие, но в разных контекстах. Поскольку беззнаковые числа обязаны работать в соответствии с правилами модульной арифметики, результатом разности двух беззнаковых должно быть также беззнаковое из того же диапазона. Как говорилось выше, это базовое положение, утвержденное спецификацией, и на основе него может быть переносимо реализована любая удобная арифметика, в том числе знаковая. С другой стороны, те, кто считает, что разность двух беззнаковых должна быть знаковой, просто отождествляют беззнаковые числа с неотрицательными, что неверно. Отсюда возникают ошибки в программах, если необоснованно использовать для заведомо неотрицательной величины беззнаковый тип. Есть масса простых тому примеров, наподобие обратного прохода по массиву, в которых компилятор может сообщить о возможной ошибке. Мы же приведем классический пример, который абсолютно корректен с точки зрения компилятора, не всегда очевиден и потому потенциально более опасен.

Предположим, Маша и Даша были в летнем лагере, в котором им повезло хорошо питаться. Были произведены замеры веса до и после пребывания, и требуется определить, кто прибавил в весе сильнее:

```
// вес Маши и Даши в начале и конце периода
unsigned m0 = 36, m1 = 39, d0 = 33, d1 = 35;
// Маша поправилась сильнее, чем Даша?
puts((m1 - m0 > d1 - d0) ? "да" : "нет");
```

[1] На момент написания книги актуальным был стандарт 1998 года с изменениями от 2003 года (C++03).

Казалось бы, тип выбран корректно: вес не может быть отрицательным. И поначалу такой код работает. Однако в некоторый момент в лагере произошла реформа, бюджет на питание распределился неравномерно, в результате чего Маша похудела. Тогда по правилам математики в левой части операции сравнения должно быть получено отрицательное число, и результат проверки должен быть отрицательным. Но, поскольку результат арифметической операции между двумя беззнаковыми числами есть также беззнаковое число, мы получим большое положительное число. Таким образом, проверка даст ошибочный положительный результат, непредусмотренный программистом. Этот пример легко исправить, избавив его от вычисления разности:

```
// Маша поправилась сильнее, чем Даша?
puts((m1 + d0 > m0 + d1) ? "да" : "нет");
```

Но тогда сильно страдает читабельность, поскольку мы вынуждены «складывать яблоки с апельсинами». Кроме того, этот пример был довольно простым, а при выполнении более сложных вычислений такой подход требует, опять же, большой внимательности и аккуратности. В то же время, корректность работы этого фрагмента может быть обеспечена путем простой замены типа `unsigned` на `int`. Учитывая это, а также человеческий фактор (человек неизбежно допускает ошибки), оказывается гораздо более надежным использовать для целочисленной переменной знаковый тип, если она интерпретируется как число, а не как набор битов или аргумент модульных операций. Именно по этим причинам от беззнаковых чисел отказались разработчики языка Java (хотя при этом они четко специфицировали поведение программы при переполнении числа). Беззнаковые же типы удобно использовать там, где биты числа используются как данные, т.е. при битовых манипуляциях, хешировании, хранении флагов и прочих подобных операциях.

Разумеется, в случае использования знаковых чисел вдвое сужается диапазон допустимых неотрицательных значений. Это неудобно, однако расчет на преимущество беззнаковых в этом отношении ненадежен, поскольку, если не хватает половины диапазона, довольно скоро не хватит и полного, и тогда в любом случае потребуется число более высокой разрядности. Другое неудобство: при использовании знаковых во избежание неопределенного поведения программы мы должны быть уверены, что значения аргументов операций достаточно малы, чтобы не вызвать переполнения. Распространено следующее правило: переполнения не будет, если оба аргумента сложения или вычитания по абсолютной величине не превышают `INT_MAX/2`.

Следует оговориться, что речь идет лишь о нецелесообразности необдуманного применения беззнаковых чисел совместно со знаковыми или в знаковом контексте. Вопрос о том, использовать ли в программе в основном лишь знаковые числа или, наоборот, беззнаковые, является очередным предметом «священных войн». Многие убеждены, что следует, наоборот, использовать лишь беззнаковые числа. Автор не склонен пытаться с этим спорить. Спор о том, что лучше, аналогичен спору о том, правильнее ли пользоваться ручным инструментом или автоматизированным: все упирается в задачи и личные предпочтения. Если стараться придерживаться одного подхода, со временем программист вырабатывает некоторые шаблоны и идиомы, позволяющие ему не задумываться в рамках этого подхода и быть аккуратным при написании любого кода.

Автор придерживается использования в коде знаковых чисел. Однако полностью избежать использования беззнаковых типов не удается по той причине, что они используются стандартной библиотекой. В частности, количество элементов в контейнерах STL, возвращаемое функцией `size`, имеет беззнаковый тип, обычно совпадающий с `size_t`. Это может вызвать проблемы, аналогичные упомянутым выше. В то же время, трудно утверждать однозначно, что беззнаковый тип для представления размеров контейнеров и позиций в

них утвержден спецификацией не напрасно, поскольку, как будет оговорено ниже, полноценного расширения диапазона это не дает.

Естественным образом возникает вопрос, насколько корректно использование знаковых типов даже там, где, казалось бы, правильнее использовать беззнаковые. К примеру, интересует вопрос корректности использования знакового типа (к примеру, совпадающего по разрядности с `ptrdiff_t`) в качестве индекса массива. На большинстве платформ разрядности типов `ptrdiff_t` и `size_t` совпадают. Считается, что в таких условиях тип `ptrdiff_t` неспособен работать с большими массивами, поскольку потенциально возможные размеры массивов в C++ определяются диапазоном беззнакового типа `size_t`. В самом деле, тип `size_t` лишь определяет значения, возвращаемые оператором `sizeof`, т.е. он обязан позволять представить размер любой структуры или массива, которые программа может выделить, однако же он не налагает требований к максимальному размеру выделяемых блоков памяти. В то же время, при создании любого массива байтов, мы можем получить указатели на его первый элемент и на элемент за последним. В свою очередь, мы можем вычислить разность этих указателей, которая одновременно должна быть равна размеру массива и корректно представляться знаковым типом `ptrdiff_t`. Строго говоря, спецификация допускает возможность существования указателей, разность между которыми не представляется типом `ptrdiff_t` (в этом случае происходит переполнение, если типы `size_t` и `ptrdiff_t` имеют одинаковую разрядность), однако выполнение такой операции снова ведет к неопределенному поведению программы. Во избежание таких ситуаций многие системы и компиляторы позволяют выделять лишь массивы размером, не большим максимального значения `ptrdiff_t` (обычно равного `SIZE_MAX/2`). Как следствие, использование возможности выделения массива, размер которого не укладывается в диапазон `ptrdiff_t`, даже если некоторые компиляторы ее и предоставляют, чревато проблемами при смене платформы. Если же обойтись в программе без выделения массивов таких размеров (что, в принципе, несложно: в крайнем случае достаточно выделить два массива вместо одного), нет никаких препятствий к использованию в качестве индексов знаковых чисел, совпадающих по разрядности с `ptrdiff_t`.

Аналогичную ситуацию имеем и с контейнерами STL. Тип `size_type`, которым определяется размер контейнера, хоть и беззнаковый, обязан представлять лишь неотрицательные значения знакового типа `difference_type`, характеризующего разность итераторов. Тому простая причина заключается в том, что, по спецификации, функция `size` возвращает число, равное разности между итераторами конца и начала контейнера, т.е. как раз преобразованное к `size_type` неотрицательное значение типа `difference_type`. Значит, максимальный размер контейнера не может быть больше максимального значения типа `difference_type`. (Здесь возникает вопрос, а почему бы тогда не возвращать сразу значение знакового типа `difference_type`, но мы его опустим. Видимо, это следствие стремления STL к обеспечению интерфейса итераторов и контейнеров, единообразного с указателями и массивами, а для последних операция получения размера дает беззнаковую величину.) Таким образом, даже если некоторые реализации позволяют поместить в контейнер большее количество элементов, чем может быть представлено знаковым числом, эту возможность не следует использовать, поскольку это плохо согласуется со спецификацией и чревато проблемами при переносе между платформами и реализациями библиотеки.

Разумеется, возможна такая реализация библиотеки, которая позволит создание контейнера большего размера, чем допускает максимальное число типа `difference_type`. При этом не будет учтено требуемое спецификацией равенство размера контейнера разности итераторов, и функция `size` вернет корректную величину. Однако в реальности разработчики реализаций стандартной библиотеки этим вопросом, видимо, не занимаются. Доказательством тому является тот факт, что программа, содержащая следующий код, при

компиляции и выполнении на многих платформах завершается некорректно, несмотря на полное ее соответствие спецификации:

```
std::vector<bool> v;
v.assign(v.max_size() / 2 + 2, false);
(*v.rbegin()).flip();
```

Это происходит в случаях, когда функция `max_size` возвращает число, равное `size_type(-1)`, которое интерпретируется как максимальное значение типа `size_type`, т.е. число, не представимое положительным значением типа `difference_type` (обычно имеющего ту же разрядность). Побитовое хранение значений в контейнере `std::vector<bool>` позволяет создать контейнер заявленного объема при сравнительно невысоком потреблении памяти. Однако индекс последнего элемента в знаковой интерпретации оказывается отрицательным числом, что и приводит к некорректной работе библиотеки. С некоторыми реализациями библиотеки такой фрагмент отрабатывает успешно, но это происходит тогда, когда функция `max_size` возвращает значение, «урезанное» до представимого знаковым типом. И, в целом, это логично: если с большими беззнаковыми числами возникают проблемы у тех, кто использует библиотеку, вполне вероятно их возникновение и у тех, кто ее реализует.

Более того, полностью корректное использование контейнеров с размерами за границами знакового диапазона вследствие несогласованности типов невозможно принципиально. К примеру, функция `std::distance` для итераторов начала и конца контейнера возвращает знаковое значение, которое в общем случае может быть не равно тому, что возвращает функция `size`. И, в отличие от разности указателей, в данном случае спецификация не оговаривает явно возможность неопределенного поведения функции (которое неизбежно возникнет при переполнении) при передаче ей итераторов начала и конца большого контейнера. А значит, корректная реализация библиотеки, использующая типы `size_type` и `difference_type` одинаковой разрядности, должна не позволять создание таких больших контейнеров (и именно поэтому некорректны реализации, в которых `max_size` возвращает `size_type(-1)`).

Выходит, что расширение диапазона, которое появляется вследствие использования беззнаковых чисел, оказывается столь же бесполезным, каким было бы и расширение в область отрицательных чисел, если бы для тех же целей был использован знаковый тип. И при таком отсутствии преимуществ мы имеем наличие потенциальных проблем при совместном использовании типов `size_type` и `difference_type` в рамках выполнения арифметических операций и сравнений.

Здесь мы неявно подразумевали 32-битные системы, не затрагивая вопросы переносимости на платформы с более высокой разрядностью. Считается, что в целях сокращения трудозатрат при переводе программ с 32-битных систем на 64-битные для всех операций, схожих с индексированием массивов и прочей адресной арифметикой, следует использовать вместо `int` и `unsigned` соответственно типы `ptrdiff_t` и `size_t`. Действительно, это поможет в ситуациях, когда на новой платформе размер типа `ptrdiff_t` окажется больше размера `int` (как обычно и бывает). В этих случаях стандартные контейнеры оказываются более емкими, но программы, использующие для индексации тип `int`, не могут задействовать этот потенциал для работы с большими объемами данных. В связи с этим в программах, которые подлежат переносу на последующие платформы с большей разрядностью, следует во многих случаях использовать вместо `int` тип `ptrdiff_t`. В настоящем издании эта тенденция, однако, не отражена, поскольку в целом автором преследуется иная цель, и в коде примеров традиционно используются встроенные типы.

# Глава 1.

# Программные интерфейсы

В этой главе будут рассмотрены некоторые технологии, предоставляющие возможность распараллеливания последовательных программ. Сами по себе эти технологии не дают в явном виде широких возможностей для распараллеливания, поскольку для этого необходим, прежде всего, алгоритм с наличием логического параллелизма. Предоставляемые средства скорее позволяют расширить возможности выполнения существующих последовательных программ с наличием логического параллелизма для выполнения в параллельной среде, т.е. внести в программу физический параллелизм.

Общий подход при использовании всех подобных технологий заключается в разработке в четыре этапа:

1) написание последовательной программы;

2) полноценная отладка последовательной программы;

3) распараллеливание программы с использованием выбранного средства;

4) отладка параллельной программы.

Казалось бы, выполнение отладки подразумевается на этапах реализации последовательной и параллельной версий программы, и потому следовало обозначить всего два этапа. Однако же отладка вынесена нами в отдельные этапы, дабы подчеркнуть ее важность. Нередко программисты, не завершив этап отладки последовательной программы, начинают ее распараллеливать, после чего сталкиваются с наличием ошибок, источник которых найти быстро не удается. При этом позже оказывается, что ошибка закралась еще на этапе реализации последовательной программы, искать же ее в распараллеленной версии оказалось существенно сложнее. Поиск ошибки, выявленной на последнем этапе, наиболее сложен, поскольку ее источник может крыться как в ошибочном алгоритме последовательной программы, так и в некорректном распараллеливании. Поэтому следует максимально избавиться от ошибок последовательного алгоритма, прежде чем переходить к распараллеливанию.

Мы рассмотрим две популярные технологии параллельного программирования. Одна из них рассчитана на использование в системах с общей памятью, другая — в системах с распределенной памятью. Поскольку в жизни часто встречаются гибридные системы, так же часто встречается и гибридное программирование — с использованием обеих описанных технологий.

Детали использования обеих технологий будут рассмотрены нами достаточно поверхностно. Многие функции и директивы будут лишь упомянуты, а не описаны подробно. Это

сделано намеренно, поскольку в задачи текущей главы входит создание общего представления о предлагаемых возможностях, за деталями же в любом случае необходимо обращаться к соответствующим спецификациям.

## 1.1. Интерфейс OpenMP

Интерфейс OpenMP (Open Multi-Processing) предназначен для распараллеливания программ в системах с общей памятью. Он предоставляет программисту удобный и переносимый способ многопоточного распараллеливания последовательной программы, оставляя за кадром тонкости создания потоков и управления ими.

Мы приведем поверхностное описание OpenMP версии 2.0 с целью создания общего представления о принципах работы с предоставляемым интерфейсом. Последующие версии стандарта принципиальных отличий не имеют. Для более подробного ознакомления с форматом описания директив и вызова функций следует обращаться к спецификациям [73, 74, 75]. Помимо этого, описанию OpenMP специально посвящено несколько литературных изданий [3, 22].

Весь программный интерфейс OpenMP можно разделить на директивы препроцессора и runtime-функции. Использование runtime-функций допустимо лишь в случае наличия поддержки OpenMP при включении в программу соответствующего заголовочного файла. Использование же директив OpenMP во время написания программы возможно всегда, при этом реально задействованы они будут лишь при использовании соответствующего флага во время компиляции в системе с поддержкой OpenMP.

Одним из несомненных достоинств OpenMP является возможность компиляции исходного текста распараллеленной программы в системе без поддержки OpenMP. При этом программа компилируется так, как будто в ней отсутствуют директивы OpenMP. Разумеется, в этом случае программа будет последовательной, однако это избавляет от необходимости поддержки нескольких версий программы. Фактически, вследствие такого положения становится возможным использовать директивы OpenMP в любой программе с тем, чтобы рано или поздно путем добавления соответствующего флага компиляции программа могла быть распараллелена.

Это, однако, касается лишь использования директив OpenMP. Использование же runtime-функций OpenMP не столь удобно, о чем подробнее будет сказано ниже при описании этих функций.

### 1.1.1. Беглый взгляд «под капот» OpenMP

Одним из довольно популярных простейших примеров вычислительных задач, используемых для демонстрации распараллеливания программ, является программа вычисления числа $\pi$. Обычно при этом приближенно вычисляют определенный интеграл от производной арктангенса. Мы же используем ряд Лейбница, поскольку получаемая при этом программа менее обременена как таковыми вычислениями и потому в нашем случае может быть более наглядна.

Ряд Лейбница для вычисления числа $\pi$ выглядит следующим образом:

$$\sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}. \tag{1.1}$$

Одним из свойств ряда (1.1) является очень медленная сходимость. Обычно при приближенном вычислении знакочередующихся рядов для сокращения накапливаемой погрешно-

сти выполняют суммирование членов парами, при этом полное количество вычисляемых членов регулируется заданной требуемой точностью исходя из абсолютной величины каждого очередного члена. Мы же позволим себе вычислять фиксированное количество членов ряда по одному за итерацию, поскольку вопросы точности лежат вне обсуждаемой темы.

Простая последовательная программа вычисления суммы первых $n$ членов ряда оказывается довольно короткой:

```
enum { NUM_ITER = 1000000 };

int main(int argc, char *argv[])
{
 double sum = 0.0;
 for (int i = 0; i < NUM_ITER; ++i)
  sum += ((i & 1) ? -1.0 : 1.0) / ((i << 1) | 1);
 fprintf(stdout, "%.16f\n", sum * 4.0);
 return 0;
}
```

Она состоит из одного цикла, в котором текущее значение частичной суммы ряда последовательно пополняется значениями очередных его членов. В конце полученная сумма увеличивается в четыре раза, чтобы получить аппроксимацию числа $\pi$. При вычислении каждого члена ряда не используются рекуррентные соотношения, все итерации являются независимыми друг от друга, что позволяет нам свободно выполнять распараллеливание итераций.

Для иллюстрации возможностей OpenMP попробуем сначала выполнить распараллеливание с помощью низкоуровневых средств, которыми пользуются сами реализации OpenMP. Распределим приведенные вычисления равномерно между $N$ потоками. Будем считать для простоты, что $n$ кратно $N$. Поскольку все итерации являются независимыми друг от друга и приблизительно одинаковыми по длительности, наиболее естественным способом распараллеливания работы будет распределение ее между потоками равными интервалами по $n/N$ итераций (рис. 1.1). Каждый поток должен вычислить локальную сумму своей части ряда, после чего полученные во всех потоках результаты должны быть просуммированы.

Для низкоуровневой организации многопоточного распараллеливания существуют различные программные интерфейсы. К примеру, можно воспользоваться интерфейсом Windows API, предоставляемым операционными системами семейства Microsoft Windows. В табл. 1.1 перечислены некоторые из функций, предоставляемых Windows API для создания потоков и управления ими. Приведенный набор, однако, не является достаточным для написания полноценных многопоточных приложений. В частности, здесь не приведены основные примитивы синхронизации и функции работы с ними, хотя некоторые из приведенных функций, такие как функции ожидания, являются универсальными и могут работать и с потоками, и с объектами синхронизации. Для знакомства с доступными объектами синхронизации и функциями работы с ними, а также за более подробным описанием перечисленных функций, следует обращаться к документации интерфейса Windows API.

С использованием некоторых из перечисленных функций приведенный выше код может быть распараллелен, к примеру, следующим образом:

```
enum { NUM_ITER = 1000000, NUM_THREADS = 4 };

// структура параметров потока
struct thr_param
{
```

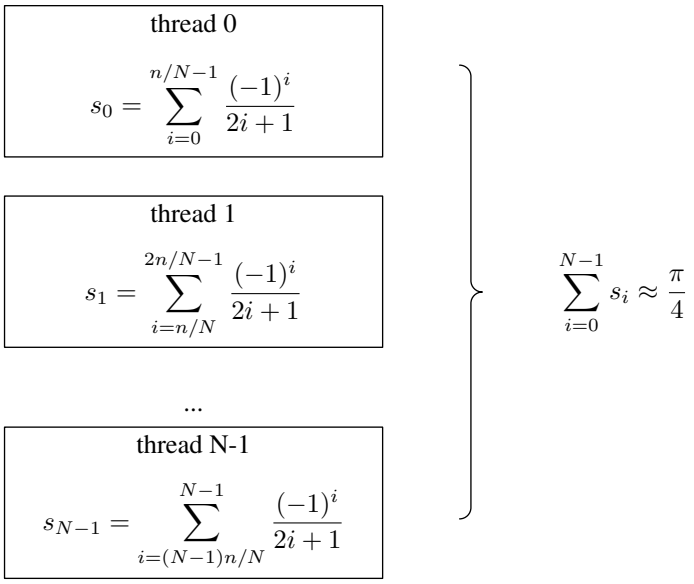



Рис. 1.1. Разбиение конечного ряда для вычисления в нескольких потоках

```
 int begin;
 int end;
 double result;
};

// функция потока
DWORD WINAPI thr_proc(LPVOID param)
{
 thr_param &p = *static_cast<thr_param *>(param);
 for (int i = p.begin; i < p.end; ++i)
  p.result += ((i & 1) ? -1.0 : 1.0) / ((i << 1) | 1);
 return 0;
}

int main(int argc, char *argv[])
{
 // заполнение параметров для каждого потока
 thr_param param[NUM_THREADS];
 for (int i = 0; i < NUM_THREADS; ++i)
 {
  param[i].begin = i * (NUM_ITER / NUM_THREADS);
  param[i].end = (i + 1) * (NUM_ITER / NUM_THREADS);
  param[i].result = 0.0;
 };
 // создание потоков
 HANDLE hdl[NUM_THREADS];
 DWORD dwId[NUM_THREADS];
 for (int i = 0; i < NUM_THREADS; ++i)
  hdl[i] = ::CreateThread(
   NULL, 0,
   thr_proc, &param[i],
   0, &dwId[i]);
 // ожидание завершения работы потоков
```

Таблица 1.1. Некоторые функции интерфейса Windows API для управления потоками

| Функция | Действие |
|---|---|
| `CreateThread` | создание потока, функция возвращает хэндл созданного потока (handle — ручка, рукоять), т.е. некую сущность, позволяющую получать информацию об объекте (созданном потоке) и управлять им; |
| `CloseHandle` | закрытие хэндла (объект, возможно, продолжает существовать, однако обращение к нему через закрытый хэндл невозможно); |
| `GetCurrentThread` | получение хэндла текущего потока; |
| `SuspendThread` | приостановление выполнения потока; |
| `ResumeThread` | возобновление выполнения потока; |
| `ExitThread` | завершение текущего потока; |
| `TerminateThread` | аварийное завершение другого потока; |
| `WaitForSingleObject` | ожидание сигнализации объекта (в данном случае — завершения потока) в течение заданного интервала времени; |
| `WaitForMultipleObjects` | ожидание сигнализации одного из нескольких объектов (завершения одного из нескольких потоков) или же сразу всех. |

```
  ::WaitForMultipleObjects(NUM_THREADS, hdl, TRUE, INFINITE);
  // освобождение ресурсов
  for (int i = 0; i < NUM_THREADS; ++i)
    ::CloseHandle(hdl[i]);
  // суммирование результатов воедино
  double sum = 0.0;
  for (int i = 0; i < NUM_THREADS; ++i)
    sum += param[i].result;
  fprintf(stdout, "%.16f\n", sum * 4.0);
  return 0;
}
```

Программа осуществляет создание `NUM_THREADS` потоков, после чего ожидает их завершения, освобождает ресурсы, суммирует промежуточные результаты и выводит полученное значение. Каждый из созданных потоков выполняет свою часть вычислений. В начале программы осуществляется заполнение массива структур `thr_param`, которые служат для передачи каждому потоку входных параметров и получения результата. В этих структурах содержатся данные о границах интервала суммирования, а также переменная, в которой будет сохранено значение промежуточной суммы, вычисленной в соответствующем потоке.

Как таковое выполнение итераций полностью возлагается на функцию потока `thr_proc`. В этой функции осуществляется вычисление суммы членов ряда из заданного интервала. Полученный результат сохраняется в структуре, адрес которой был передан функции потока.

При наличии в системе достаточного количества свободных процессоров все потоки могут выполнять свою работу параллельно, вследствие чего результат вычислений может быть получен гораздо быстрее, нежели в последовательном варианте. Не следует, однако, ожидать, что он будет получен в $N$ раз быстрее (где $N$ — количество потоков). Это

Таблица 1.2. Некоторые функции интерфейса pthreads

| Функция | Действие |
|---|---|
| `pthread_create` | создание потока, идентификатор созданного потока возвращается в качестве выходного параметра; |
| `pthread_self` | получение идентификатора текущего потока; |
| `pthread_equal` | сравнение двух идентификаторов потоков (может использоваться, к примеру, для сравнения некоторого идентификатора потока со значением, возвращаемым функцией `pthread_self`); |
| `pthread_join` | ожидание завершения некоторого потока и освобождение ресурсов, выделенных при его создании; |
| `pthread_detach` | пометка на освобождение ресурсов, выделенных при создании потока, без ожидания его завершения; |
| `pthread_exit` | завершение текущего потока; |
| `pthread_cancel` | запрос на принудительное завершение другого потока. |

идеальный, но недостижимый выигрыш в производительности, невозможность которого обусловлена законом Амдала [9, 46, 1].

В качестве альтернативного варианта рассмотрим программный интерфейс POSIX Threads (pthreads), предоставляемый приложениям для организации многопоточной работы многими UNIX-системами. Основные функции управления потоками перечислены в табл. 1.2.

Помимо функций управления потоками интерфейс pthreads предоставляет также функции для работы с некоторыми примитивами синхронизации — объектом взаимного исключения (mutual exclusion, `mutex`), условной переменной (`cond`, аналог объекта `Event` в Windows API) и блокировкой чтения/записи. Подробно о функциях, предоставляемых интерфейсом pthreads, можно прочитать в [36, 16, 37, 56, 81].

Распараллеленный с использованием интерфейса pthreads код во многом похож на вариант распараллеливания с помощью функций Windows API. Изменению подлежит лишь фрагмент от создания потоков до освобождения ресурсов:

```
// ...
// создание потоков
pthread_t pth[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; ++i)
 ::pthread_create(&pth[i], NULL, thr_proc, &param[i]);
// ожидание завершения работы потоков и освобождение ресурсов
for (int i = 0; i < NUM_THREADS; ++i)
 ::pthread_join(pth[i], NULL);
// суммирование результатов воедино
// ...
```

Вследствие различий программных интерфейсов, также требует изменения сигнатура функции потока, при этом содержимое ее не меняется:

```
void *thr_proc(void *param)
{
 // ...
 return NULL;
```

```
}
```

В обоих приведенных случаях параллельные вычисления выполняются в NUM_THREADS потоках, главный же поток вычислений не выполняет, а лишь ожидает завершения остальных. Таким образом, в процессе присутствует на один поток больше, чем необходимо. Чтобы этого не происходило, осуществим создание меньшего на единицу количества дополнительных потоков, при этом главный поток перед выполнением ожидания должен выполнить свою часть работы. Модифицированная в итоге программа с использованием интерфейса pthreads имеет следующий вид:

```
enum { NUM_ITER = 1000000, NUM_THREADS = 4 };

// структура параметров потока
struct thr_param
{
 int begin;
 int end;
 double result;
};

// функция потока
void *thr_proc(void *param)
{
 thr_param &p = *static_cast<thr_param *>(param);
 for (int i = p.begin; i < p.end; ++i)
  p.result += ((i & 1) ? -1.0 : 1.0) / ((i << 1) | 1);
 return NULL;
}

int main(int argc, char *argv[])
{
 // заполнение параметров для каждого потока
 thr_param param[NUM_THREADS];
 for (int i = 0; i < NUM_THREADS; ++i)
 {
  param[i].begin = i * (NUM_ITER / NUM_THREADS);
  param[i].end = (i + 1) * (NUM_ITER / NUM_THREADS);
  param[i].result = 0.0;
 };
 // создание потоков
 pthread_t pth[NUM_THREADS - 1];
 for (int i = 0; i < NUM_THREADS - 1; ++i)
  ::pthread_create(&pth[i], NULL, thr_proc, &param[i + 1]);
 // выполнение в главном потоке
 thr_proc(&param[0]);
 // ожидание завершения работы потоков и освобождение ресурсов
 for (int i = 0; i < NUM_THREADS - 1; ++i)
  ::pthread_join(pth[i], NULL);
 // суммирование результатов воедино
 double sum = 0.0;
 for (int i = 0; i < NUM_THREADS; ++i)
  sum += param[i].result;
 fprintf(stdout, "%.16f\n", sum * 4.0);
 return 0;
}
```

В приведенной программе первый элемент массива структур параметров передается функции, выполняемой в главном потоке, остальные передаются создаваемым потокам.

Теперь самое время обозначить главный момент текущего раздела. Последней приведенной программе по функциональности полностью эквивалентен следующий код:

```
enum { NUM_ITER = 1000000, NUM_THREADS = 4 };

int main(int argc, char *argv[])
{
 double sum = 0.0;
 #pragma omp parallel for num_threads(NUM_THREADS) reduction(+: sum)
 for (int i = 0; i < NUM_ITER; ++i)
  sum += ((i & 1) ? -1.0 : 1.0) / ((i << 1) | 1);
 fprintf(stdout, "%.16f\n", sum * 4.0);
 return 0;
}
```

Видно, что этот код отличается от последовательной версии лишь наличием директивы препроцессора, являющейся одной из директив высокоуровневого интерфейса многопоточного программирования OpenMP. И именно в этой директиве скрыта организация всего описанного функционала по созданию и управлению потоками.

Наличие директивы `parallel for` указывает компилятору на необходимость выполнения многопоточного распараллеливания следующего непосредственно за ней оператора цикла. Параметр `num_threads` предписывает выполнить при этом создание указанного в скобках количества потоков. Точнее, выполняется распараллеливание между указанным количеством потоков, создается же на один поток меньше, поскольку главный поток программы также участвует в параллельном выполнении цикла. Параметр `reduction` в нашем случае предписывает осуществить выделение соответствующего количества переменных для хранения промежуточных результатов суммирования, а также осуществить последующее выполнение их суммирования с помещением результата в исходную переменную.

Таким образом, видно, что весь функционал, который мы только что выполняли вручную с помощью низкоуровневых интерфейсов управления потоками, может быть выполнен автоматически и практически без нашего участия. Более того, этот эффект будет достигнут лишь при использовании специального флага компилятора. В противном случае программа скомпилируется так, как будто в ней не содержится директив OpenMP, т.е. будет последовательной. Таким образом, в одном исходном коде мы получаем последовательную и параллельную версии программы, при этом параллельная версия не привязана к какому-либо конкретному низкоуровневому интерфейсу.

Теперь, проиллюстрировав мощь интерфейса OpenMP на примере, рассмотрим его более детально.

### 1.1.2. Основные конструкции параллельного выполнения

Использование директив OpenMP заключается во вставке строк с директивами препроцессора перед подлежащими распараллеливанию участками кода. Каждая такая строка содержит имя директивы и, возможно, список параметров и имеет следующий вид:

```
#pragma omp <name> [<param1> [<param2>] ...]
```

Перечислим основные необходимые для распараллеливания программы с помощью OpenMP директивы. Прежде всего, это директива `parallel`, объявляющая параллельный регион:

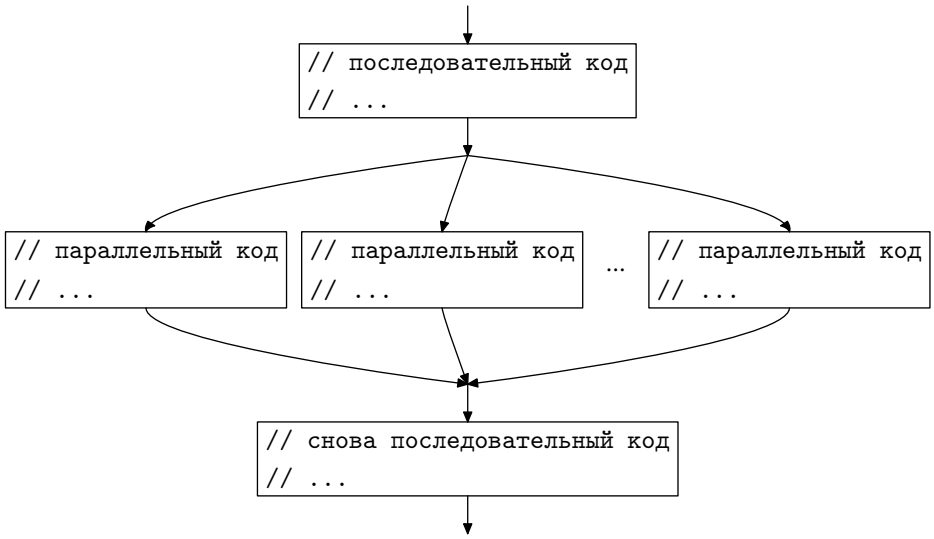



Рис. 1.2. Схема параллельного выполнения кода

```
// последовательный код
// ...
#pragma omp parallel
{
 // параллельный код
 // ...
}
// снова последовательный код
// ...
```

Объявленный такой директивой параллельный регион ограничивается оператором, следующим сразу за ним. Это может быть и составной оператор, ограниченный фигурными скобками, как в приведенном примере.

В начале параллельного региона создается некоторое количество дополнительных потоков, после чего все они, включая главный поток, выполняют участок кода, заключенный в параллельный регион (рис. 1.2). Под главным потоком подразумевается тот, который выполнял текущую ветвь кода в момент достижения параллельного региона. В конце региона выполняется барьерная синхронизация всех выполняющих регион потоков, т.е. каждый поток по достижении конца региона останавливается и ждет, пока все остальные потоки также не достигнут конца региона. После завершения выполнения региона всеми потоками выход из него осуществляется лишь главным, остальные же формально завершаются за ненадобностью (вопрос возможности кэширования потоков здесь не рассматривается, поскольку является вопросом внутренней реализации OpenMP).

С помощью директивы `parallel` объявляется участок кода, который выполняется не просто параллельно, а многократно (рис. 1.2). К примеру, следующий фрагмент кода на многопроцессорной машине выведет в общем случае несколько строк приветствия:

```
#pragma omp parallel
printf("Hello, World!\n");
```

Строго говоря, подобный код может вывести несколько строк отнюдь не последовательно, а вперемешку, поскольку здесь все потоки выполнения осуществляют одновременный

доступ к общему ресурсу — стандартному потоку вывода. По этой причине здесь и далее в подобных случаях вместо функции `printf` мы будем использовать специально реализованную функцию `synprintf`, которая имеет формат вызова, аналогичный `fprintf`, но отличается по функциональности тем, что операция вывода в файловый поток выполняется ей атомарно. В различных средах такая функция может быть реализована по-разному в зависимости от доступных средств и программных интерфейсов. К примеру, для использования в среде с поддержкой OpenMP простая реализация такой функции может быть, к примеру, следующей:

```
int synprintf(FILE *stream, const char *fmt, ...)
{
 using namespace std;
 va_list argptr;
 va_start(argptr, fmt);
 int rc = vsnprintf(NULL, 0, fmt, argptr);
 if (rc >= 0)
 {
  vector<char> buf(rc + 1);
  vsnprintf(&buf.front(), buf.size(), fmt, argptr);
  #pragma omp critical (synprintf)
  {
   fputs(&buf.front(), stream);
   fflush(stream);
  };
 };
 va_end(argptr);
 return rc;
}
```

Здесь выполнено явное разграничение доступа к общему ресурсу с помощью директивы `critical`, которая будет описана ниже. Формирование строки является сравнительно тяжеловесной операцией, поэтому она вынесена из критической секции, атомарной же является лишь операция помещения полученной строки в файловый поток. Следует иметь в виду, что это не самая удачная реализация, поскольку атомарность соблюдается не только между операциями над одним файловым потоком, но и над разными. Таким образом, эта реализация обеспечивает лишь корректность работы, но не эффективность. Для внесения атомарности в рамках работы с каждым файловым потоком отдельно удобно использовать описанные ниже блокировки OpenMP.

Если запрашиваемое количество создаваемых в параллельном регионе потоков в явном виде не указано, оно определяется реализацией OpenMP. К примеру, общее количество потоков может быть равно количеству установленных в системе процессоров. Управлять количеством создаваемых потоков можно несколькими способами, один из них — с помощью дополнительного параметра `num_threads`:

```
#pragma omp parallel num_threads(5)
synprintf(stdout, "Hello, World!\n");
```

В результате выполнения такой программы будет выведено пять строк приветствия (при выключенном режиме динамического управления количеством создаваемых потоков). Возможно также управление не только количеством потоков, но и вообще фактом осуществления распараллеливания текущего региона. К примеру, если нам требуется выполнять регион параллельно лишь в случае удовлетворения каких-либо условий, мы можем воспользоваться параметром `if`:

```
int n = (argc > 1) ? strtol(argv[1], NULL, 0) : 1;
#pragma omp parallel if(n > 1 && n <= 16) num_threads(n)
synprintf(stdout, "Hello, World!\n");
```

Такой код выведет заданное извне количество строк лишь в случае, если это количество попадает в некий указанный диапазон.

Директива `parallel` может также иметь и другие параметры, касающиеся разделения переменных между потоками. Эту тему мы затронем позже.

Очевидно, многократное параллельное выполнение идентичных действий в большинстве случаев лишено смысла (за исключением, пожалуй, создания каких-либо тестовых нагрузок). Обычно распараллеливаемый код содержит какие-либо участки, которые могут быть выполнены параллельно, но каждый из них при одних условиях должен быть выполнен лишь однократно. В таких случаях требуется распределить выполнение параллельного региона частями между параллельными потоками, для чего внутри параллельного региона может быть выполнено разделение работы. Для такого разделения предусмотрены директивы `for`, `sections` и `single`.

Директива `sections` позволяет распределить между потоками разные информационно независимые участки кода, т.е. такие, из которых выполнение одних не зависит от результата выполнения других. К примеру, если некоторый регион содержит последовательно несколько независимых подзадач, все они могут быть выделены в отдельные секции участка `sections`:

```
#pragma omp parallel num_threads(3)
#pragma omp sections
{
 #pragma omp section
 synprintf(stdout, "Hello, World 1!\n");
 #pragma omp section
 synprintf(stdout, "Hello, World 2!\n");
 #pragma omp section
 synprintf(stdout, "Hello, World 3!\n");
}
```

Каждый независимый фрагмент участка `sections` обрамляется в отдельный участок `section`. В приведенном примере в начале параллельного региона принудительно создается количество потоков, равное количеству секций. Однако такой подход не всегда оптимален и удобен. Обычно правильнее не задавать количество потоков, а оставить его выбор на совесть системы OpenMP, чтобы обеспечить большую гибкость при смене платформы.

Наконец, одна из наиболее важных директив — директива `for`. Как известно, наибольший потенциал к распараллеливанию содержат циклы. Директива `for` позволяет распараллелить выполнение отдельных итераций некоторого цикла. Следует отметить, что для возможности распараллеливания цикла необходимо, чтобы итерации были независимыми между собой по входным и выходным данным. Иначе говоря, результат выполнения цикла не должен зависеть от порядка выполнения отдельных итераций. К примеру, итерации на основе рекуррентных формул являются зависимыми (входными данными для каждой итерации являются выходные данные предыдущих), поэтому их распараллеливать обычно гораздо сложнее, если вообще возможно.

Директива `for` распределяет итерации одноименного цикла между существующими потоками параллельного региона:

```
#pragma omp parallel
#pragma omp for
```

```
for (int i = 0; i < 3; ++i)
 synprintf(stdout, "Hello, World %d!\n", i);
```

При этом аргументы оператора цикла должны быть в так называемой канонической форме. Говоря коротко, аргументы должны содержать инициализацию некой целочисленной знаковой переменной, сравнение ее значения с заданным значением границы диапазона и ее приращение, которое может содержать увеличение или уменьшение на фиксированную величину. Подробнее об определении понятия «каноническая форма цикла» можно прочитать в спецификации [73, 74].

Итерации цикла, объявленного в контексте директивы `for`, распределяются между потоками объемлющего региона `parallel`. Для явного указания способа распределения итераций между потоками используется параметр `schedule`. К примеру, если указан параметр `schedule(static)`, будет выполнено статическое блочно-циклическое распределение итераций между потоками. Размер блоков может быть указан при этом в скобках через запятую после ключевого слова `static`. Если же будет указан параметр `schedule(dynamic)`, будет выполнено динамическое распределение, т.е. итерации будут выдаваться потокам порциями по мере выполнения ими предыдущей работы. Такой режим обычно менее производителен, чем в случае статического распределения, однако оказывается гораздо оптимальнее при выполнении итераций с разной длительностью. Режим, включаемый параметром `schedule(guided)`, также обеспечивает динамическое распределение, однако при этом распределяемые между потоками порции итераций постепенно уменьшаются. Наконец, параметр `schedule(runtime)` обеспечивает возможность выбора способа распределения итераций во время выполнения программы на базе значения переменной окружения `OMP_SCHEDULE`.

Во многих случаях бывает необходимо выделить в параллельном регионе участок кода, который будет выполняться лишь одним потоком. Для этого предназначена директива `single`. К примеру, в следующем фрагменте кода вывод сообщения каждый раз будет производиться из всей группы потоков лишь одним:

```
#pragma omp parallel
{
 #pragma omp single
 synprintf(stdout, "Stage 1\n");

 #pragma omp for
 for (int i = 0; i < n; ++i)
 {
  // ...
 }

 #pragma omp single
 synprintf(stdout, "Stage 2\n");

 #pragma omp for
 for (int i = 0; i < n; ++i)
 {
  // ...
 }
}
```

В конце участков `sections`, `for` и `single` неявно выполняется барьерная синхронизация всех потоков объемлющего параллельного региона, кроме случаев, когда в соответствующей директиве указан параметр `nowait`. Указание такого параметра может в некоторых

ситуациях обеспечить повышение быстродействия за счет исключения задержек на ожидание. В следующем фрагменте предполагается, что первые два цикла информационно независимы друг от друга, а третий зависит от них обоих:

```
#pragma omp parallel
{
 #pragma omp for nowait
 for (int i = 0; i < n; ++i)
 {
  // ...
 }
 #pragma omp for
 for (int j = 0; j < n; ++j)
 {
  // ...
 }

 #pragma omp for
 for (int k = 0; k < n; ++k)
 {
  // ...
 }
}
```

Вследствие указания параметра `nowait` перед первым циклом, после него не выполняется синхронизация потоков, в результате чего освободившиеся потоки могут сразу приступать к выполнению второго цикла. Поскольку третий цикл зависит от первых двух, необходимо, чтобы перед его выполнением вся предыдущая работа была завершена. Так и происходит по причине того, что во второй директиве `for` параметр `nowait` не указан, а значит, после второго цикла выполняется барьерная синхронизация.

В более сложных случаях может потребоваться использование явной барьерной синхронизации потоков, для чего служит директива `barrier`:

```
#pragma omp barrier
```

Директива `parallel` зачастую используется совместно с директивами `for` и `sections`, поэтому для удобства программирования предусмотрены формы более короткой записи:

```
#pragma omp parallel for
// ...
#pragma omp parallel sections
// ...
```

Каждая такая запись эквивалентна указанию двух последовательных строк — одной директиве `parallel` и одной директиве `for` или `sections` соответственно.

### 1.1.3. Некоторые вспомогательные директивы

Помимо описанных директив, которых в большинстве простых случаев бывает достаточно, интерфейс OpenMP предлагает также несколько других вспомогательных конструкций.

Директива `master` объявляет участок внутри параллельного региона, который должен быть выполнен только главным потоком. Как уже говорилось выше, главным потоком является тот, который породил текущую группу потоков параллельного региона. По смыслу

эта директива близка к `single`, но, помимо жесткой привязки к главному потоку, отличается еще отсутствием барьерной синхронизации на границе соответствующего участка.

Другой директивой, также в некотором смысле близкой к `single`, является директива `critical`. Она объявляет участок внутри параллельного региона, который может выполняться лишь одним потоком в каждый момент времени. С использованием такой конструкции можно внутри параллельного региона осуществить поочередный доступ к каким-либо общим данным:

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; ++i)
{
 int item = calc(i);
 #pragma omp critical
 sum += item;
}
```

В приведенном фрагменте осуществляется изменение общей переменной `sum`. Поскольку при этом каждый раз производится последовательное чтение ее значения из памяти, изменение значения и сохранение результата обратно в память, при параллельном выполнении возможны коллизии, вследствие которых будет получен неверный результат. Использование директивы `critical` в данном случае гарантирует, что пока один поток не сохранит измененное значение, другой не начнет его чтение.

Следует отметить, что приведенный пример грамотнее реализовать с использованием параметра `reduction` директивы `parallel for`. Директива же `critical` предназначена для более сложных ситуаций, нежели приведенная, к примеру, для атомарного доступа одновременно к нескольким ресурсам, или же когда возникает необходимость объединить несколько операций с одним ресурсом. Такая ситуация иллюстрируется следующим кодом:

```
list<int> stack;
// ...

n = stack.size();
#pragma omp parallel for
for (int i = 0; i < n; ++i)
{
 int arg;
 #pragma omp critical
 {
  arg = stack.back();
  stack.pop_back();
 }
 process(arg);
}
```

Здесь получение очередного аргумента из вершины стека выполняется с помощью отдельных операций чтения вершины и ее изъятия. При этом критическая секция обеспечивает атомарное выполнение совокупности обеих операций.

В параллельном регионе может быть объявлено много участков `critical`, и по умолчанию выполнение каждого из них исключает одновременное выполнение всех остальных участков. Однако зачастую такие участки требуют взаимоисключающего выполнения не со всеми, а лишь с некоторыми объявленными в том же регионе участками `critical`. Как правило, в таких участках происходит обращение к одному ресурсу. В этих случаях следу-

ет задавать в качестве параметра директивы `critical` имя соответствующей критической секции в круглых скобках:

```
set<int> a, b;
#pragma omp parallel for
for (int i = 0; i < n; ++i)
{
 int item1 = calc1(i);
 #pragma omp critical (seta)
 a.insert(item1);

 int item2 = calc2(i);
 #pragma omp critical (setb)
 b.insert(item2);

 int item3 = calc3(i);
 #pragma omp critical (setb)
 b.insert(item3);
}
```

В этом случае поток по достижении начала критической секции будет ждать до тех пор, пока она не станет свободна, т.е. пока не будет достигнута ситуация, когда нет ни одного потока, выполняющегося в рамках участка `critical` с тем же именем. Критические секции, имя для которых не указано, считаются одноименными.

Директива `atomic` является более упрощенным вариантом `critical`. В то время как область действия конструкции `critical` может быть какой угодно, область действия конструкции `atomic` ограничивается одним оператором модификации переменной, таким как в первом приведенном примере с директивой `critical`. При этом критическим участком является левая часть оператора, а именно изменение какой-либо переменной. Правая часть не попадает в критический участок, поэтому, если в правой части стоит, к примеру, вызов длительной вычислительной функции, это не вызовет задержки всех остальных потоков:

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; ++i)
{
 #pragma omp atomic
 sum += calc(i);
}
```

В некоторых случаях бывает необходимо выполнять некоторые участки тела распараллеленного цикла в том порядке, в котором они выполнялись бы в последовательном цикле. Для обозначения таких участков в теле распараллеленного цикла служит директива `ordered`. Директива `for` соответствующего цикла должна при этом также содержать параметр `ordered`:

```
#pragma omp parallel for ordered
for (int i = 0; i < n; ++i)
{
 int item = calc(i);
 #pragma omp ordered
 synprintf(stdout, "Result of iteration %d: %d\n", i, item);
}
```

В результате выполнения такого кода будет выведен упорядоченный по номерам итераций список строк. Вычисления на каждой итерации будут завершаться в общем случае в

неупорядоченной последовательности, однако последний этап каждой итерации, т.е. вывод строки, будет выполнен в том же порядке, как и в случае последовательного цикла. При отсутствии директивы `ordered` был бы неупорядоченным и вывод программы.

Во время работы потоков параллельного региона с какими-либо общими переменными неизбежно возникают ситуации, когда измененное каким-либо потоком состояние переменной не отражено в памяти (к примеру, хранится в регистрах процессора и пока не помещено в память), и потому не доступно другим потокам. На входе и выходе конструкций `parallel`, `critical`, `ordered`, а также на выходе из конструкций `for`, `sections`, `single` и `barrier`, неявно выполняется синхронизация переменных. В некоторых случаях может потребоваться выполнить такую синхронизацию явно, для этого служит директива `flush`:

```
#pragma omp flush [(<var1> [, <var2>] ...)]
```

Директива `flush` может быть указана без параметров, в этом случае будет выполнена синхронизация всех доступных общих переменных. Однако такие затраты могут оказаться избыточными и сказаться на производительности, поэтому в скобках может быть явно указан список переменных, которые необходимо синхронизировать.

### 1.1.4. Разделение данных

Очевидно, что при многопоточной работе потребуется определить, какие данные будут общими для всех потоков, а какие будут частными для каждого из них. С этой целью интерфейс OpenMP предусматривает директиву `threadprivate`, а также несколько параметров для директив `parallel`, `sections`, `for` и `single`.

По умолчанию переменные, объявленные вне параллельного региона, считаются общими; объявленные внутри, кроме статических, — частными. Такой режим может быть изменен с помощью параметра `default(none)` директивы `parallel`. Если параметр `default` не указан, система ведет себя так, как если бы был указан параметр `default(shared)`. В этом случае предполагается, что все переменные, видимые в контексте появления директивы `parallel`, являются общими, если не оговорено иное с помощью одного из параметров, описанных ниже. Если же указан параметр `default(none)`, для всех переменных, используемых в параллельном регионе и видимых в момент его объявления, должно присутствовать явное указание относительно того, являются ли они общими или частными (за исключением некоторых случаев, оговоренных в спецификации [73, 74]).

Директива `threadprivate` объявляет разделенный запятыми список глобальных либо статических переменных, которые следует рассматривать частными для потоков, создаваемых в параллельных регионах. В момент создания потоков в начале параллельного региона для каждого из них создается своя копия такой переменной. До момента первого обращения каждая копия инициализируется в соответствии со строкой инициализации исходной переменной. Если в строке инициализации исходной переменной фигурируют какие-либо объекты или другие переменные, их значения не должны меняться до момента инициализации копии переменной, т.е. до первого обращения к ней. Поскольку с момента инициализации переменной в главном потоке до момента создания потоков в параллельном регионе ее значение может быть изменено, значения частных копий могут отличаться от значения в главном потоке. Однако при входе в параллельный регион созданным частным переменным может быть присвоено значение переменной из главного потока. Для этого в директиве `parallel` следует использовать параметр `copyin`:

```
static int a = 10;
#pragma omp threadprivate(a)
```

```
a = 5;

#pragma omp parallel for num_threads(5) copyin(a)
for (int i = 0; i < 10; ++i)
 synprintf(stdout, "Local value of 'a' is %d\n", a);
```

В приведенном фрагменте при отсутствии параметра `copyin` значение копии статической переменной внутри цикла будет отличаться от значения в главном потоке, в связи с чем выведенные на экран значения будут различными. Наличие `copyin` заставляет компилятор в начале параллельного региона присвоить копиям значение из главного потока.

Для объявления частными либо общими переменных, не являющихся глобальными либо статическими, предусмотрено несколько параметров для директив `parallel`, `sections`, `for` и `single`. С помощью параметра `private` достигается схожий с предыдущим результат — для всех перечисленных в скобках через запятую переменных в каждом потоке создается своя копия. При этом она остается неинициализированной, для объектов используется конструктор по умолчанию.

Параметр `firstprivate` действует аналогично `private`, за исключением того, что созданная копия инициализируется значением оригинальной переменной, которое она имела непосредственно во время достижения текущей директивы. Для объектов используется конструктор копирования.

Наконец, параметр `lastprivate` также действует аналогично `private`, за исключением того, что значения указанных переменных, полученные на последней по номеру итерации цикла конструкции `for` либо последней секции конструкции `sections`, на выходе из соответствующего участка копируются в исходную переменную с помощью оператора присваивания.

Есть еще один похожий параметр, использование которого допустимо лишь в конструкции `single`. При указании частной переменной в параметре `copyprivate`, ее значение после выполнения текущего участка `single` копируется в соответствующие частные переменные остальных потоков параллельного региона с помощью оператора присваивания.

Параметр `shared` позволяет в явном виде объявить перечисленные в скобках переменные общими. Такая необходимость может возникнуть в случае использования параметра `default(none)` директивы `parallel`.

Наконец, параметр `reduction` предназначен для обозначения общих переменных, в которые попадает результат некой ассоциативной операции над множеством аргументов, выполняемой внутри параллельного региона. К примеру, это может быть сумма, произведение или битовое сложение по модулю 2 для большого количества аргументов. Аргумент параметра `reduction` состоит из двух частей, разделенных двоеточием: обозначение операции и список соответствующих переменных через запятую.

При входе в параллельный регион для каждой такой общей переменной в каждом новом потоке создается своя копия и инициализируется «пустым» начальным значением. Для суммы это ноль, для произведения — единица, для логического «ИЛИ» — «ЛОЖЬ» и т.д. После выполнения параллельного региона значения всех созданных копий с помощью той же операции «сливаются» с переменной главного потока.

Следующий фрагмент кода показывает, как приведенный выше пример использования директивы `critical` может быть реализован с использованием параметра `reduction`:

```
int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < n; ++i)
{
 int item = calc(i);
```

```
 sum += item;
}
```

В отличие от реализации с помощью `critical`, здесь не будет выполняться никаких ожиданий между потоками, за исключением барьера в конце региона, вследствие чего работать такой код может гораздо эффективнее.

### 1.1.5. Runtime-функции

Помимо директив компиляции, интерфейс OpenMP предлагает также некоторые функции для вызова непосредственно из кода программы. Такими функциями во многих случаях удобно пользоваться при отладке или проверке быстродействия распараллеленной программы.

Однако следует иметь в виду, что использование runtime-функций ставит под удар одно из главных достоинств OpenMP, а именно возможность компиляции распараллеленной программы в системе без поддержки OpenMP. Поэтому, если есть стремление иметь такую возможность, удобнее бывает не злоупотреблять использованием runtime-функций, а поискать такие пути построения алгоритма, при которых все распараллеливание вместе с блокировками будет реализовано с использованием только директив.

При наличии поддержки OpenMP и включении соответствующего флага компилятора объявляется макрос `_OPENMP`, что дает возможность использовать runtime-функции внутри конструкций условной компиляции наподобие следующей:

```
 int proc;
#ifdef _OPENMP
 proc = omp_get_num_procs();
#else
 proc = 1;
#endif //_OPENMP
```

Разумеется, это позволяет содержать в одном исходном тексте и параллельную, и последовательную версии программы даже с использованием runtime-функций. Однако по мере усложнения программы такой подход приводит к трудности восприятия и поддержания соответствия фрагментов обеих версий и, в конечном итоге, становится шагом в сторону поддержки двух версий программы.

В качестве альтернативы использованию условной компиляции спецификация OpenMP предлагает применение функций-заглушек, эмулирующих работу runtime-функций в последовательной среде выполнения. Такие функции могут быть реализованы и приложены к программе, использующей runtime-функции OpenMP, если она должна быть скомпилирована в среде без поддержки OpenMP.

Мы не будем углубляться в описание runtime-функций OpenMP, а ограничимся лишь их перечислением, за подробным же описанием следует обратиться к спецификации [73, 74]. Прежде всего, это функции среды выполнения:

- `omp_set_num_threads` — установка количества потоков, создаваемых по умолчанию в последующих параллельных регионах без явного указания параметра `num_threads`;
- `omp_get_num_threads` — получение количества потоков, выполняющихся в текущем параллельном регионе, из которого вызвана функция;
- `omp_get_max_threads` — получение максимального количества потоков, которое может быть создано в параллельных регионах;

- `omp_get_thread_num` — получение номера текущего потока в текущем параллельном регионе в диапазоне $[0; N-1]$, где $N$ — количество потоков, возвращаемое функцией `omp_get_num_threads`;
- `omp_get_num_procs` — получение количества доступных процессоров;
- `omp_in_parallel` — получение информации о том, вызвана ли функция изнутри параллельного региона;
- `omp_set_dynamic` — разрешение/запрещение режима, при котором среда во время выполнения имеет право регулировать количество создаваемых потоков в последующих параллельных регионах для наиболее оптимального потребления доступных ресурсов;
- `omp_get_dynamic` — получение информации о том, включен ли режим динамического управления количеством создаваемых потоков;
- `omp_set_nested` — разрешение/запрещение вложенного параллелизма, т.е. выполнения распараллеливания вложенных параллельных регионов;
- `omp_get_nested` — получение информации о том, разрешен ли вложенный параллелизм.

Некоторые из перечисленных функций предназначены для управления параметрами среды выполнения, что зачастую бывает довольно удобным. Поскольку в некоторых ситуациях, упомянутых выше, бывает предпочтительнее избегать использования функций OpenMP, проблема управления параметрами среды может быть решена путем использования переменных окружения `OMP_SCHEDULE`, `OMP_NUM_THREADS`, `OMP_DYNAMIC` и `OMP_NESTED`, что подробнее описано в спецификации [73, 74].

Помимо обеспечения доступа к параметрам среды, перечисленные runtime-функции позволяют выполнять более тонкое управление распределением задач по сравнению с возможностями, предоставляемыми директивами. По сути, runtime-функции предоставляют несколько более низкоуровневый интерфейс. К примеру, вот так может быть выполнено распределение итераций цикла, которое могло бы быть реализовано одной директивой `parallel for`:

```
// если уже в параллельном регионе
// и вложенный параллелизм запрещен
if (omp_in_parallel() && !omp_get_nested())
{
 // выполним цикл последовательно
 for (int i = 0; i < n; ++i)
  process(i);
}
else
{
 // иначе получим количество процессоров
 int num = omp_get_num_procs();
 // зададим соответствующее количество потоков
 omp_set_num_threads(num);
 // выполним цикл параллельно
 #pragma omp parallel
 {
  // получим реальное количество потоков
```

```
  #pragma omp single
  num = omp_get_num_threads();
  // и номер нашего потока среди них
  int id = omp_get_thread_num();
  // выполним нашу часть итераций
  // в соответствии с циклическим распределением
  for (int i = id; i < n; i += num)
   process(i);
 }
}
```

Кроме функций среды выполнения, интерфейс OpenMP предлагает также функции работы с блокировками. Эти функции обеспечивают возможность более гибкого построения конструкций, по смыслу близких к `critical`. Для использования функций блокировки программа должна объявить переменную, которая будет ассоциирована с блокировкой. Каждая такая переменная в один момент времени может быть захвачена лишь одним потоком. При попытке прочих потоков захватить ее они переводятся в состояние ожидания до тех пор, пока захвативший поток ее не освободит.

Блокировки поддерживаются двух типов — простые и вкладываемые. Простая блокировка может быть захвачена потоком, только если она не захвачена ни одним из потоков, включая текущий. Вкладываемая блокировка может быть захвачена в случае, когда она свободна, либо когда она уже захвачена текущим потоком. В этом случае происходит увеличение счетчика вложенности блокировки. При вызове функции освобождения вложенной блокировки производится уменьшение счетчика вложенности, фактическое же освобождение происходит при его обнулении.

Для работы с простыми и вкладываемыми блокировками предусмотрены типы данных `omp_lock_t` и `omp_nest_lock_t` соответственно, а также следующие функции:

- `omp_init_lock` и `omp_init_nest_lock` — создание блокировки;
- `omp_destroy_lock` и `omp_destroy_nest_lock` — уничтожение блокировки;
- `omp_set_lock` и `omp_set_nest_lock` — захват блокировки;
- `omp_unset_lock` и `omp_unset_nest_lock` — освобождение блокировки;
- `omp_test_lock` и `omp_test_nest_lock` — проверка блокировки, попытка захвата без выполнения ожидания.

Явные функции управления блокировками позволяют производить более тонкое разграничение доступа к критическим участкам, нежели предлагаемое конструкцией `critical`. К примеру, они могут быть использованы, когда критические участки частично перекрываются.

Допустим, внутри параллельного региона осуществляется извлечение головных элементов очереди и помещение их в хранилище — контейнер `std::map`. Новый ключ формируется как текущий размер хранилища, т.е. каждый новый ключ не совпадает с предыдущими, поскольку удаления из хранилища не происходит. По новому ключу головной элемент помещается в хранилище, после чего изымается из очереди:

```
// формирование уникального в хранилище идентификатора
int id = store.size();
// помещение в хранилище головного элемента очереди
store[id] = queue.front();
```

```
// извлечение головного элемента очереди
queue.pop_front();
```

В этом примере используется два ресурса — очередь и хранилище. Соответственно, с ними связано два критических участка, которые частично перекрываются и потому не могут быть реализованы двумя конструкциями `critical`. Разумеется, можно использовать одну такую конструкцию на оба участка, а можно использовать runtime-функции блокировки:

```
list<some_type> queue;
map<int, some_type> store;

omp_lock_t lockstore, lockqueue;
omp_init_lock(&lockstore);
omp_init_lock(&lockqueue);

// ...

#pragma omp parallel
{
 // ...

 omp_set_lock(&lockstore);

 // формирование уникального в хранилище идентификатора
 int id = store.size();

 omp_set_lock(&lockqueue);

 // помещение в хранилище головного элемента очереди
 store[id] = queue.front();

 omp_unset_lock(&lockstore);

 // извлечение головного элемента очереди
 queue.pop_front();

 omp_unset_lock(&lockqueue);

 // ...
}

// ...

omp_destroy_lock(&lockstore);
omp_destroy_lock(&lockqueue);
```

Другое удобство, которое появляется при использовании runtime-функций для работы с блокировками, заключается в том, что исключается опасность совпадения блокировок по именам, которая возникает при использовании директив `critical`. К примеру, некоторый параллельный регион может содержать именованный участок `critical` и вызывать какую-либо функцию, которая внутри себя также использует участок `critical` с непреднамеренно совпадающим именем. Тогда эти участки будут взаимоисключать выполнение друг друга, в связи с чем могут возникнуть ненужные задержки или даже взаимоблокировка (deadlock). Если же вызываемая функция будет использовать свою переменную блокировки, это гарантированно исключит возможность случайного совпадения по имени

с какой-либо критической секцией или другой переменной блокировки.

Есть и другой пример, чаще возникающий в практике, и усложненный вариант которого встретится нам позже в главе, посвященной модели актеров. Допустим, есть некая рекурсивная функция, содержащая параллельный регион с критическими секциями, которые защищают доступ к ее локальным переменным между потоками. При этом рекурсивный вызов функции выполняется изнутри параллельного региона. При включенном режиме вложенного параллелизма во время выполнения такой функции распараллеливание будет осуществляться на каждом рекурсивном вызове, вследствие чего может работать одновременно много групп потоков различных параллельных регионов разных уровней. В такой ситуации все секции с одним именем взаимоисключают одновременное выполнение друг друга не в потоках группы текущего параллельного региона, а во всех параллельно выполняющихся потоках всех групп. В большинстве случаев такая ситуация бывает непреднамеренной и может по недосмотру быть не учтенной, вследствие чего, как и в предыдущем случае, могут возникать излишние задержки выполнения (в лучшем случае) или (в худшем) взаимоблокировка. Использование же в функции локальной переменной блокировки вместо критической секции ограничивает блокируемые потоки лишь группой текущего параллельного региона.

Может показаться, что вследствие простоты OpenMP трудно создать ситуацию, в которой может возникнуть взаимоблокировка. Однако не следует обольщаться, это не так. Возможность возникновения взаимоблокировки иллюстрирует следующий фрагмент кода:

```
// возможна взаимоблокировка!!!
#pragma omp parallel sections num_threads(2)
{
 #pragma omp section
 {
  #pragma omp critical (a)
  {
   // удерживание a и ожидание b
   #pragma omp critical (b)
   synprintf(stdout, "Hello, World 1!\n");
  }
 }
 #pragma omp section
 {
  #pragma omp critical (b)
  {
   // удерживание b и ожидание a
   #pragma omp critical (a)
   synprintf(stdout, "Hello, World 2!\n");
  }
 }
}
```

Наконец, OpenMP предлагает две runtime-функции для оценки времени выполнения:

- `omp_get_wtime` — получение количества секунд в виде числа с двойной точностью, прошедшего с некоторого фиксированного момента времени;
- `omp_get_wtick` — получение величины разрешения таймера, т.е. величины интервала времени между двумя последовательными изменениями таймера (тиками).

Первая из этих функций позволяет приблизительно оценить интервал времени между двумя точками выполнения программы, вторая выдает погрешность этой оценки:

```
double t, r;
// начало замера
t = omp_get_wtime();
// длительная операция
process();
// конец замера
t = omp_get_wtime() - t;
// относительная погрешность замера
r = omp_get_wtick() / t;

// время выполнения и погрешность в процентах
synprintf(stdout, "Time elapsed: %e sec +- %f%%", t, 100 * r);
```

За более подробным описанием runtime-функций OpenMP и форматом их вызова следует обратиться к спецификации [73, 74].

### 1.1.6. Вычисление определенного интеграла

Для завершения знакомства с возможностями интерфейса OpenMP рассмотрим небольшой пример распараллеливания некой вычислительной процедуры. Покажем некоторые моменты, которые удобно учитывать при написании программы для дальнейшего распараллеливания.

Приведенный ниже код иллюстрирует вычисление определенного интеграла на интервале $[a; b]$ от некоторой функции одного аргумента методом средних прямоугольников:

```
// количество интервалов - начальное разбиение
int n = 10;
// количество выполненных итераций
int iter = 0;
// текущий и предыдущий результаты
double sum = 0.0, sumpre = 0.0;

double h, x, f;
int i;

do
{
 // шаг разбиения
 h = (b - a) / n;
 // вычисление интеграла для заданного разбиения
 sumpre = sum;
 sum = 0.0;
 for (i = 0; i < n; ++i)
 {
  // середина интервала
  x = a + h * (i + 0.5);
  // значение интегрируемой функции
  f = func(x);
  // площадь прямоугольника
  sum += f * h;
 };
 // удвоение количества интервалов
 n <<= 1;
 // если итерация была первая, продолжаем
```

```
} while (iter++ == 0 || fabs(sum - sumpre) > eps);

synprintf(stdout, "Result: %.16f, iterations: %d\n", sum, iter);
```

Программа задает некоторое начальное количество интервалов разбиения области интегрирования и итеративно вычисляет приближенные значения интеграла, удваивая количество интервалов разбиения после каждой итерации. Выполнение завершается, когда разница между вычисленными приближенными значениями интеграла на двух последних итерациях становится меньше заданной допустимой погрешности. Для осуществления такой проверки должно быть выполнено минимум две итерации.

На каждой итерации выполняется вычисление шага текущего разбиения и цикл суммирования площадей прямоугольников. В теле цикла осуществляется вычисление середины интервала и значение интегрируемой функции в ней, после чего вычисленное значение площади прямоугольника добавляется к текущей сумме.

В приведенном примере верхняя граница внешнего цикла не известна заранее, поэтому, несмотря на отсутствие зависимостей между итерациями, его не удается распараллелить автоматически. Распараллеливанию подлежит внутренний цикл, в котором осуществляется суммирование. При объявлении этого цикла параллельным нам потребуется атомарный доступ к внешней по отношению к циклу переменной `sum`, в которой хранится текущее значение суммы, либо использование параметра `reduction`. Помимо этого, в теле цикла используются еще две переменные. Поскольку они являются внешними по отношению к циклу и видимыми в момент объявления цикла параллельным, по умолчанию они являются общими. Однако для корректной работы параллельного цикла они должны быть сделаны частными. В результате получаем параллельный цикл следующего вида:

```
#pragma omp parallel for private(x, f) reduction(+: sum)
for (i = 0; i < n; ++i)
{
 // середина интервала
 x = a + h * (i + 0.5);
 // значение интегрируемой функции
 f = func(x);
 // площадь прямоугольника
 sum += f * h;
};
```

Мы видим, что даже в такой достаточно простой вычислительной процедуре потребовалось явное объявление переменных частными. Если бы по каким-либо причинам эти переменные были объявлены статическими или глобальными, ситуация потребовала бы использования дополнительных директив. При реализации более сложных вычислительных алгоритмов задача явного указания частных и общих переменных может оказаться гораздо сложнее, в результате чего станет легко ошибиться в деталях. По этой причине общей рекомендацией здесь является следование такому стилю написания программ, когда объявление переменных происходит лишь внутри блока, в котором они непосредственно используются. Использования глобальных или статических переменных при параллельном программировании по возможности стоит вообще избегать. При таком подходе действия, выполняемые по умолчанию конструкциями OpenMP, в большинстве случаев соответствуют требованиям реализуемого алгоритма.

К примеру, приведенный код может быть переписан следующим образом (комментарии опущены):

```
int n = 10;
```

```
int iter = 0;
double sum = 0.0, sumpre = 0.0;

do
{
 double h = (b - a) / n;
 sumpre = sum;
 sum = 0.0;
 #pragma omp parallel for reduction(+: sum)
 for (int i = 0; i < n; ++i)
 {
  double x = a + h * (i + 0.5);
  double f = func(x);
  sum += f * h;
 };
 n <<= 1;
} while (iter++ == 0 || fabs(sum - sumpre) > eps);

synprintf(stdout, "Result: %.16f, iterations: %d\n", sum, iter);
```

Объявление локальных по отношению к циклам переменных перенесено из начала фрагмента в соответствующие блоки. В результате этого изменилась их область видимости, вследствие чего в момент объявления директивы распараллеливания они не становятся общими, и потому не требуется их явное объявление частными.

В приведенном коде распараллеливание происходит на каждой итерации приближенного вычисления интеграла. Можно пойти дальше и внести изменения в критерий остановки внешнего цикла, что позволит внести его внутрь параллельного региона:

```
// количество интервалов - начальное разбиение
int n = 10;
// количество выполненных итераций
int iter = 0;
// текущий и предыдущий результаты
double sum = 0.0, sumpre = 0.0;
// критерий выполнения итераций
bool perform = true;

#pragma omp parallel
while (perform)
{
 // шаг разбиения
 double h = (b - a) / n;
 // вычисление интеграла для заданного разбиения
 #pragma omp single
 {
  sumpre = sum;
  sum = 0.0;
 };
 #pragma omp for reduction(+: sum)
 for (int i = 0; i < n; ++i)
 {
  // середина интервала
  double x = a + h * (i + 0.5);
  // значение интегрируемой функции
  double f = func(x);
```

```
  // площадь прямоугольника
  sum += f * h;
 };
 #pragma omp single
 {
  // удвоение количества интервалов
  n <<= 1;
  // если итерация была первая, продолжаем
  perform = iter++ == 0 || fabs(sum - sumpre) > eps;
 };
};

synprintf(stdout, "Result: %.16f, iterations: %d\n", sum, iter);
```

В этом варианте создание параллельного региона осуществляется один раз, после чего внутри него многократно распараллеливается внутренний цикл. Такой подход не лишен смысла, поскольку создание потоков в начале параллельного региона — довольно тяжеловесная операция. Однако многие реализации OpenMP выполняют кэширование потоков, т.е. созданные однажды потоки по завершении параллельного региона не уничтожаются, а ожидают входа в следующий параллельный регион. В этих случаях подобные усовершенствования теряют пользу для производительности, однако могут существенно усложнить код.

## 1.2. Интерфейс передачи сообщений MPI

Описанию программного интерфейса MPI (Message-Passing Interface), помимо спецификаций [71, 72], посвящено немало изданий, в том числе русскоязычных [1, 2, 4, 28, 46], поэтому мы не будем описывать его подробно. Вместо этого мы рассмотрим варианты решения с использованием MPI нескольких наиболее простых для распараллеливания задач, многие из которых, тем не менее, возникают довольно часто.

Мы не будем стремиться рассмотреть все тонкости и моменты, существующие в различных реализациях MPI. На момент написания книги далеко не все реализации MPI охватывают спецификацию версии 2.0, поэтому в дальнейшем для определенности будем рассматривать интерфейс MPI версии 1.1. Это необходимо помнить, поскольку, порой, будут звучать утверждения касательно именно этой версии интерфейса, к примеру, касательно отсутствия некоторых возможностей, появившихся в версии 2.0.

### 1.2.1. Снова ряд Лейбница

Прежде всего, приведем простой пример использования библиотеки MPI на примере уже рассмотренного нами ранее частичного вычисления ряда Лейбница (1.1). Последовательное приближенное вычисление числа $\pi$ на основе суммы первых $n$ членов ряда выглядит следующим образом:

```
// вычисление числа пи
double sum = 0.0;
for (int i = 0; i < n; ++i)
 sum += ((i & 1) ? -1.0 : 1.0) / ((i << 1) | 1);
sum *= 4.0;
```

Для простоты будем предполагать, что у нас в наличии однородная вычислительная система, а размер задачи $n$ (в данном случае — количество вычисляемых членов ряда) де-

Таблица 1.3. Некоторые функции интерфейса sockets

| Функция | Действие |
|---|---|
| `socket` | создание сокета для работы по заданному протоколу; |
| `bind` | привязка сокета к конкретному адресу; |
| `listen` | переход в режим ожидания входящих соединений и задание размера очереди соединений; |
| `accept` | принятие входящего соединения, открытие канала связи и создание нового сокета, ассоциированного с каналом; |
| `connect` | соединение с удаленным сервером (создание канала с клиентской стороны); |
| `send` | отправка данных по каналу; |
| `recv` | прием данных, пришедших по каналу; |
| `select` | выбор сокетов, готовых к выполнению операции отправки или приема без блокирования; |
| `shutdown` | запрет приема и/или отправки данных по каналу (закрытие канала связи); |
| `close` | уничтожение сокета. |

лится нацело на количество вычисляющих процессов $N$. Тогда можем распределить нагрузку между процессами равномерно. Некоторые вопросы неравномерного распределения нагрузки будут затронуты позже.

Будем осуществлять распараллеливание по той же схеме, что и ранее — одинаковыми интервалами по $n/N$ итераций (рис. 1.1). Отличие заключается лишь в том, что сейчас итерации распределяются не по потокам, а по процессам.

Как и при рассмотрении интерфейса OpenMP, прежде, чем переходить к примеру использования интерфейса MPI, приведем сначала пример «ручного» распараллеливания вычислений между некоторым количеством процессов, работающих, возможно, на разных машинах, с использованием более низкоуровневых средств. Для выполнения этой задачи необходимо использование какого-либо механизма межпроцессного взаимодействия (IPC, Interprocess Communication). Мы используем связь процессов по протоколу TCP через интерфейс так называемых сокетов (Socket Interface, sockets).

Протокол TCP предполагает создание канала связи типа «один к одному» между двумя процессами. Для организации каналов и обмена данными по ним интерфейс sockets предлагает функции, перечисленные в табл. 1.3.

Использование этого интерфейса для организации канального взаимодействия программ предполагает следующую последовательность (рис. 1.3). Программа-сервер создает сокет с помощью функции `socket`, осуществляет привязку к адресу, на котором будет ожидать соединения (`bind`), и задает размер очереди входящих соединений (`listen`). После этого путем вызова функции `accept` возможно принятие входящего запроса на соединение, если таковой есть. Если запроса на соединение нет, функция `accept` не возвращает управления до его прихода (в случае работы с блокирующими сокетами). Программа-клиент создает сокет также с помощью функции `socket` и осуществляет соединение с программой-сервером с помощью функции `connect`. Когда запрос на соединение от клиента приходит на сервер, функция `accept` возвращает управление, создав новый сокет для общения с клиентом. После этого первоначально созданный сервером сокет может снова принимать

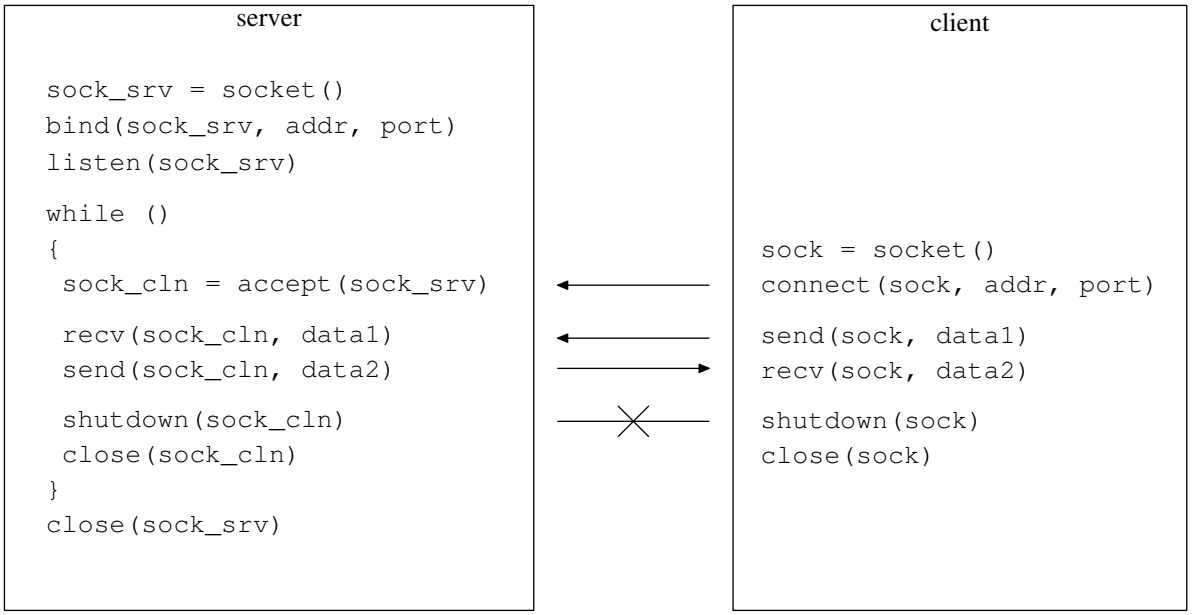



Рис. 1.3. Схема клиент-серверного взаимодействия на основе сокетов

соединения функцией `accept`, а только что созданный сокет, привязанный к каналу связи с клиентом, может быть использован для отправки и получения данных с помощью функций `send` и `recv` соответственно. После завершения сеанса связи обе программы закрывают канал функцией `shutdown`, после чего может быть уничтожен соответствующий сокет с помощью функции `close`. Функция `select` позволяет работать с несколькими блокирующими сокетами в одном потоке.

Следует иметь в виду, что вызовы функций на рис. 1.3 показаны схематично и отражают лишь последовательность, однако не соответствуют реальному формату вызова функций интерфейса sockets. Подробно об использовании этого интерфейса и формате вызова его функций можно прочитать в [31, 37].

Воспользуемся описанным интерфейсом для организации взаимодействия между несколькими процессами с целью распараллеливания между ними частичного вычисления ряда (1.1). Зададим следующую структуру параллельной программы. Процесс-сервер осуществляет прием соединений от процессов-клиентов и раздает им задания на выполнение. Процессы-клиенты в количестве $N$ экземпляров осуществляют соединение с процессом-сервером, получают от него задание, выполняют его и отсылают результат обратно процессу-серверу. Получив все результаты, процесс-сервер суммирует их и выводит полученное значение. Код клиентской и серверной сторон содержится в одной программе:

```
// структуры входных и выходных параметров процессов
struct proc_in_param
{
 // начало интервала итераций
 int begin;
 // конец интервала итераций
 int end;
};
struct proc_out_param
{
 // результат вычислений
```

```
 double result;
};


int main(int argc, char *argv[])
{
 int rc = 2;

 // признак, является ли программа сервером или клиентом
 char mode = (argc > 1) ? argv[1][0] : '\0';
 // адрес и порт сервера (в обоих режимах)
 unsigned long srvaddr;
 int srvport;
 // количество вычисляющих процессов (только для сервера)
 int size;
 // полное количество итераций (только для сервера)
 int n;

 // ------ чтение и проверка входных параметров ------
 if (!((mode == 's' && argc == 6) || (mode == 'c' && argc == 4)) ||
  (srvaddr = inet_addr(argv[2])) == INADDR_NONE ||
  (srvport = strtol(argv[3], NULL, 0)) == 0 ||
  (mode == 's' && (size = strtol(argv[4], NULL, 0)) == 0) ||
  (mode == 's' && (n = strtol(argv[5], NULL, 0)) == 0))
 {
  fprintf(stderr,
   "usage:\n"
   "%s s <addr> <port> <procnum> <iternum>\n"
   "or\n"
   "%s c <addr> <port>\n",
   argv[0], argv[0]);
 }
 else
 {
  int ret;
  // проверка кратности итераций процессам (только в сервере)
  if (mode != 's' || n % size == 0)
  {
   // формирование структуры адреса сервера
   struct sockaddr_in san = {0};
   san.sin_family = AF_INET;
   san.sin_port = htons(srvport);
   san.sin_addr.s_addr = srvaddr;

   if (mode == 's')
   {
    // ------ код сервера ------
    int srvsock;
    // создание серверного сокета
    if ((srvsock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) != -1)
    {
     // привязка к адресу и задание размера очереди соединений
     if (bind(srvsock, (struct sockaddr *) &san, sizeof(san)) == 0 &&
      listen(srvsock, SOMAXCONN) == 0)
     {
```

```
    // раздача заданий
    vector<int> clnsock(size);
    for (int rank = 0; rank < size; ++rank)
    {
     // принятие очередного соединения
     clnsock[rank] = accept(srvsock, NULL, 0);
     assert(clnsock[rank] != -1);

     // формирование задания - интервал итераций
     proc_in_param param = {0};
     param.begin = rank * (n / size);
     param.end = (rank + 1) * (n / size);
     // отправка назначенного задания
     ret = send(clnsock[rank], &param, sizeof(param), 0);
     assert(ret == sizeof(param));
    };

    // сбор результатов
    double sum = 0.0;
    for (int rank = 0; rank < size; ++rank)
    {
     // получение очередного промежуточного результата
     proc_out_param param = {0};
     int rsize = 0;
     do
      ret = recv(
       clnsock[rank], &((char *) &param)[rsize],
       sizeof(param) - rsize, 0);
     while (ret > 0 && size_t(rsize += ret) < sizeof(param));
     assert(rsize == sizeof(param));

     // учет полученного результата
     sum += param.result;

     // уничтожение клиентского сокета
     shutdown(clnsock[rank], SHUT_WR);
     close(clnsock[rank]);
    };

    sum *= 4.0;
    fprintf(stdout, "%.16f\n", sum);

    // успешное завершение
    rc = 0;
   }
   else
    fprintf(stderr, "error: can't bind server socket or listen\n");
   // уничтожение серверного сокета
   close(srvsock);
  }
  else
   fprintf(stderr, "error: can't create server socket\n");
}
else
{
```

```
    // ------ код клиента ------
    int sock;
    // создание сокета
    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) != -1)
    {
     // соединение с сервером
     if (connect(sock, (struct sockaddr *) &san, sizeof(san)) == 0)
     {
      // получение назначенного задания
      proc_in_param inparam = {0};
      int rsize = 0;
      do
       ret = recv(
        sock, &((char *) &inparam)[rsize],
        sizeof(inparam) - rsize, 0);
      while (ret > 0 && size_t(rsize += ret) < sizeof(inparam));
      assert(rsize == sizeof(inparam));

      // выполнение задания - вычисление своей части ряда
      double locsum = 0.0;
      for (int i = inparam.begin; i < inparam.end; ++i)
       locsum += ((i & 1) ? -1.0 : 1.0) / ((i << 1) | 1);

      // отправка результата
      proc_out_param outparam = {0};
      outparam.result = locsum;
      ret = send(sock, &outparam, sizeof(outparam), 0);
      assert(ret == sizeof(outparam));
      // закрытие канала
      shutdown(sock, SHUT_WR);

      // успешное завершение
      rc = 0;
     }
     else
      fprintf(stderr, "error: can't connect to server\n");
     // уничтожение сокета
     close(sock);
    }
    else
     fprintf(stderr, "error: can't create client socket\n");
   };
  }
  else
   fprintf(stderr, "error: iternum isn't divisible by procnum\n");
 };
 return rc;
}
```

Приведенная программа для своей задачи выглядит весьма громоздко, и это учитывая, что контроль ошибок был сильно упрощен. В самом начале запущенный процесс осуществляет чтение и анализ параметров командной строки. Этот фрагмент в целях наглядности также максимально упрощен, поскольку в нашем случае не является ключевым. Через параметры командной строки процессу передаются:

1) признак, должен ли процесс работать в качестве сервера или клиента;

2) адрес узла, на котором работает процесс-сервер;

3) порт, на котором процесс-сервер ожидает соединений от процессов-клиентов;

4) общее количество процессов-клиентов;

5) общее количество вычисляемых членов ряда.

Последние два параметра передаются лишь процессу-серверу. Первым делом формируется структура адреса сервера, которая используется в процессах обоих типов. Процесс-сервер осуществляет создание сокета и привязку его к адресу, после чего переходит в режим ожидания соединений. Как только принимается запрос на соединение и осуществляется соединение с очередным клиентом, сервер вычисляет границы интервала итераций, которые надлежит выполнить клиенту, и передает их ему. После раздачи заданий всем клиентам, сервер осуществляет прием и суммирование промежуточных результатов от них.

Клиентская часть программы заметно проще серверной. Процесс-клиент осуществляет создание сокета и соединение по заданному адресу. Как только соединение выполнено, клиент получает от сервера информацию о границах интервала итераций, которые ему надлежит выполнить, и приступает к их выполнению. По завершении вычислений клиент отправляет результат серверу и закрывает соединение. Данные между процессами передаются в двоичном виде, что, с одной стороны, позволяет избежать потерь точности, с другой, заставляет ограничиться выполнением программы на машинах с одинаковой архитектурой (одинаковым представлением данных).

Полный запуск такой параллельной программы на выполнение предполагает запуск одного экземпляра процесса-сервера и $N$ экземпляров процессов-клиентов. Эту задачу можно выполнять, к примеру, с помощью следующего shell-скрипта:

```
#!/bin/sh

PROGNAME=./pi-sock
SRVADDR=127.0.0.1
SRVPORT=3456

if test $# -ne 2
then
  echo "usage: $0 <procnum> <iternum>" 1>&2
else
  $PROGNAME s $SRVADDR $SRVPORT $1 $2 &
  for ((i = 0; i < $1; ++i))
  do
    $PROGNAME c $SRVADDR $SRVPORT &
  done
  wait
fi
```

Соответственно, запуск такого скрипта для выполнения 1000 итераций четырьмя процессами может быть осуществлен следующей командой:

```
./pi-sock.sh 4 1000
```

Приведенная программа получилась гораздо более громоздкой, нежели в случае «ручного» многопоточного распараллеливания тех же вычислений в рамках одного процесса.

Это естественно, если учесть, что задача распараллеливания между процессами гораздо более трудоемка, поскольку требует использования механизмов межпроцессного взаимодействия с целью обмена данными.

Теперь рассмотрим, как похожая схема распределенного вычисления того же ряда может быть реализована с использованием функций MPI:

```
enum { MASTER_PROC_ID = 0 };

int main(int argc, char *argv[])
{
 int rc = 2;
 // количество итераций
 int n;

 // инициализация MPI
 MPI_Init(&argc, &argv);

 // получение количества итераций из командной строки
 if (argc == 2 && (n = strtol(argv[1], NULL, 0)) > 0)
 {
  int rank, size;
  // получение количества процессов и номера текущего процесса
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // проверка кратности итераций процессам
  if (n % size == 0)
  {
   // вычисление числа пи
   double locsum = 0.0;
   double sum = 0.0;

   int nloc = n / size;
   for (int i = nloc * rank; i < nloc * (rank + 1); ++i)
    locsum += ((i & 1) ? -1.0 : 1.0) / ((i << 1) | 1);

   MPI_Allreduce(
    &locsum, &sum, 1, MPI_DOUBLE,
    MPI_SUM, MPI_COMM_WORLD);
   sum *= 4.0;

   // вывод результата в одном процессе
   if (rank == MASTER_PROC_ID)
    fprintf(stdout, "%.16f\n", sum);

   // успешное завершение
   rc = 0;
  }
  else
   if (rank == MASTER_PROC_ID)
    fprintf(stderr, "error: iternum isn't divisible by procnum\n");
 }
 else
  fprintf(stderr, "error: iternum is not specified\n");
```

```
 MPI_Finalize();
 return rc;
}
```

В представленной программе первым делом осуществляется инициализация интерфейса MPI с передачей аргументов функции `main`. Далее задается количество вычисляемых членов ряда, которое в нашем случае читается из параметра командной строки. В общем случае оно может быть прописано в программе константой, задано вводом пользователя, прочитано из файла конфигурации или как-либо еще. После этого с помощью функции `MPI_Comm_size` выполняется получение количества процессов в группе коммуникатора `MPI_COMM_WORLD`, т.е. общего количества процессов, выполняющихся в рамках работы текущей параллельной программы. Также производится получение номера текущего процесса в этой группе с помощью функции `MPI_Comm_rank`. Поскольку мы условились, что количество вычисляемых членов ряда должно быть кратно количеству процессов, мы убеждаемся, что остаток от соответствующего деления равен нулю.

Наконец, мы вычисляем размер `nloc` интервала итераций для текущего процесса. В нашем случае он во всех процессах одинаков, поэтому левая и правая его граница для каждого процесса легко вычисляются на основе произведения размера интервалов и номера соответствующего процесса.

После вычисления каждым процессом локальной части ряда все полученные результаты должны быть просуммированы. Для этого используется редукционная операция с параметром `MPI_SUM`. Во время вызова функции `MPI_Allreduce` значения переменных `locsum` из всех процессов группы коммуникатора `MPI_COMM_WORLD` суммируются, после чего полученное значение рассылается снова во все процессы и помещается в переменную `sum`. После увеличения значения этой переменной в четыре раза все процессы хранят в ней значение, близкое к числу $\pi$.

В конце программы выполняется освобождение ресурсов, занятых под свои нужды средствами MPI.

Запуск такой программы также должен осуществляться средствами MPI (с помощью команды `mpirun` или `mpiexec`). К примеру, для запуска из текущей директории программы `pi-mpi`, выполняющей 1000 итераций с распараллеливанием между четырьмя процессами, можно использовать команду следующего вида:

```
mpirun -np 4 ./pi-mpi 1000
```

Нельзя не признать, что распараллеленная с использованием функций MPI программа получилась гораздо длиннее, чем в случае использования OpenMP. В то же время, несмотря на использование механизмов межпроцессного взаимодействия, она оказалась гораздо менее громоздкой, чем в случае «ручного» распараллеливания на основе интерфейса sockets, имея при этом практически ту же функциональность. Главное существенное отличие в функциональности заключается в том, что здесь создается $N$ процессов и все они выполняют вычисления, тогда как в программе на сокетах один процесс был нами выделен для работы в качестве диспетчера, т.е. всего количество выполняющихся процессов составляло $N+1$. Другое отличие в том, что в данной программе интервалы итераций каждый процесс вычисляет самостоятельно, тогда как в предыдущем случае эта работа была возложена на диспетчер, поскольку именно он назначал порядковые номера вычисляющим процессам по мере их подключения.

Здесь текст MPI-программы был приведен практически полностью, включая все необходимые вызовы функций MPI общего назначения. В дальнейшем они во многих случаях будут опускаться, вплоть до вызовов `MPI_Comm_size` и `MPI_Comm_rank`. Приводиться будут

лишь фрагменты кода, иллюстрирующие непосредственно описываемый механизм взаимодействия с интерфейсом MPI с целью осуществления обмена данными.

Приведенная программа выполняет, как уже было сказано, равномерное распределение нагрузки между процессами для случая, когда количество итераций кратно количеству процессов. Разумеется, такое стечение обстоятельств возникает далеко не всегда, поэтому позже мы опишем возможные варианты неравномерной нагрузки.

### 1.2.2. Краткое описание предоставляемых функций

Спецификация MPI определяет интерфейс, предоставляющий удобный обмен данными между процессами в распределенной среде. При этом, помимо непосредственно функций обмена данными, предоставляются также вспомогательные функции, так или иначе повышающие удобство такого обмена, такие как формирование произвольных типов данных или произвольных групп процессов.

Мы перечислим кратко основные предоставляемые программным интерфейсом MPI функции. Эти функции не будут рассмотрены нами детально, также не будет охвачено все множество предоставляемых функций. Для этого уже существует немало литературы, в частности [1, 2, 4, 28, 46, 78]. Мы лишь упомянем поверхностно задачи, выполняемые функциями предоставляемого интерфейса, с тем, чтобы была понятна его структура и принципы функционирования, поскольку это необходимо для понимания приводимых далее примеров программ. Некоторые особенности и детали выполняемых функциями действий, а также их формат вызова, станут ясны при разборе этих программ. Если же этого окажется недостаточно, следует обращаться к упомянутой литературе или спецификациям [71, 72].

#### Функции работы с группами и коммуникаторами

Прежде чем описывать обменные операции, необходимо кратко осветить коммуникационную модель MPI. MPI-программа выполняется в виде множества взаимодействующих процессов. Каждая коммуникационная операция осуществляется в контексте некоторого объекта-коммуникатора, который указывается при вызове этой операции. С каждым коммуникатором связана группа процессов, которые могут обмениваться данными через этот коммуникатор. Во время выполнении каждого процесса MPI-программы (вернее, после вызова `MPI_Init` и до вызова `MPI_Finalize`) ему доступно два коммуникатора: коммуникатор `MPI_COMM_WORLD`, включающий все процессы программы, и `MPI_COMM_SELF`, включающий лишь один текущий процесс. На основе них могут формироваться новые коммуникаторы для обмена между произвольными подгруппами процессов.

Группа процессов представляется отдельным объектом, ассоциированным с объектом коммуникатора. Получение группы, связанной с некоторым коммуникатором, возможно с помощью функции `MPI_Comm_group`. Между группами процессов возможно выполнение операций объединения (`MPI_Group_union`), пересечения (`MPI_Group_intersection`), разности (`MPI_Group_difference`). Новую группу можно сформировать, перечислив включаемые в нее процессы некой существующей группы (функция `MPI_Group_incl`), или же наоборот, перечислив процессы, которые не должны быть включены в новую (`MPI_Group_excl`). Все перечисленные функции создают на выходе новый объект-группу, который после использования должен быть уничтожен функцией `MPI_Group_free`.

Все процессы, входящие в группу, пронумерованы по порядку с нуля. Получить значение своего номера (ранга) в некоторой группе процесс может с помощью функции `MPI_Group_rank`. Общее количество процессов, входящих в группу, возвращается функцией `MPI_Group_size`. Процесс, входящий в разные группы, имеет в них в общем случае разные

номера. Для получения номера некоторого процесса в одной группе по его же номеру в другой используется функция MPI_Group_translate_ranks. Для сравнения содержимого двух групп может быть использована функция MPI_Group_compare.

Для получения информации о процессах, входящих в группу некоторого коммуникатора, доступны функции MPI_Comm_size, MPI_Comm_rank, MPI_Comm_compare. Эти функции обеспечивают быстрый доступ к группе процессов, ассоциированной с коммуникатором, минуя явное получение объекта группы и его последующее уничтожение. При выполнении коммуникационных операций целевой процесс характеризуется совокупностью двух значений — коммуникатором и рангом процесса в рамках этого коммуникатора.

Создание коммуникатора на основе некоторой уже сформированной группы осуществляется функцией MPI_Comm_create. Функция MPI_Comm_dup обеспечивает создание нового коммуникатора, полностью совпадающего с переданным ей в качестве аргумента. На основе некоторого коммуникатора может быть создано сразу несколько разных коммуникаторов, ассоциированных с набором не перекрывающихся подгрупп процессов исходного коммуникатора. Для этого используется функция MPI_Comm_split. В аргументах ей передается признак, по которому процессы объединяются в соответствующие подгруппы. Каждый созданный процессом коммуникатор должен быть уничтожен с помощью функции MPI_Comm_free.

В дополнение к описанным функциям работы с коммуникаторами MPI также содержит функции для работы с двумя типами виртуальных топологий. Виртуальная топология описывает заданную программистом схему связей в некоторой группе процессов. Главное предназначение виртуальных топологий заключается в повышении удобства адресации процессов между собой. Работа с виртуальными топологиями осуществляется посредством использования коммуникаторов с установленным соответствующим атрибутом. Для определения, представляет ли некий коммуникатор какую-либо виртуальную топологию, может быть использована функция MPI_Topo_test.

Декартова топология предоставляет возможность создания многомерной прямоугольной решетки процессов. Для заданных размерности решетки и полного количества процессов в ней с помощью функции MPI_Dims_create может быть вычислено сбалансированное число процессов в каждом измерении. На основе полученных таким образом (или каким-либо другим) размеров с помощью функции MPI_Cart_create может быть создан коммуникатор с атрибутом декартовой топологии. При этом в каждом измерении может быть задана периодичность (т.е. соответствующее измерение может быть закольцовано). В любой момент от сформированного таким образом коммуникатора может быть получена переданная ему при создании информация с помощью функций MPI_Cartdim_get и MPI_Cart_get. Каждый процесс, входящий в решетку, помимо ранга имеет и логические декартовы координаты. С помощью функции MPI_Cart_coords могут быть получены координаты произвольного процесса по его рангу, с помощью же функции MPI_Cart_rank наоборот, на основе координат может быть получен ранг процесса. Каждый процесс, используя функцию MPI_Cart_shift, может получить ранги двух других процессов, отстоящих от него в заданном измерении на заданном расстоянии. Это бывает удобно для кольцевого обмена данными, когда каждый процесс отправляет данные в одну сторону и получает с другой. Наконец, вся многомерная решетка может быть разбита на подрешетки меньшей размерности с помощью функции MPI_Cart_sub. В результате образуется множество новых коммуникаторов, что сродни результату выполнения MPI_Comm_split.

Топология графа обеспечивает возможность задания произвольных схем связей в соответствии с потребностями программы. Данные о вершинах графа и связывающих их ребрах задаются при создании коммуникатора с установленным атрибутом графа с помощью функции MPI_Graph_create. Переданная при создании информация о графе может быть

получена позже с помощью функций MPI_Graphdims_get и MPI_Graph_get. Для каждого процесса по его рангу с помощью функции MPI_Graph_neighbors в любой момент может быть получен список рангов его непосредственных соседей (вершин графа, соединенных ребрами с заданной). Количество элементов в этом списке может быть получено путем вызова функции MPI_Graph_neighbors_count.

Вообще говоря, MPI предусматривает два типа коммуникаторов: интра-коммуникаторы и интер-коммуникаторы. К первым относятся коммуникаторы, предназначенные для обмена данными между процессами, принадлежащими одной группе. Вторые позволяют обмен данными между процессами из разных групп. До сих пор, говоря о коммуникаторах, мы подразумевали интра-коммуникаторы (и также будем делать в дальнейшем, если не оговорено иное). Интер-коммуникаторы существуют для повышения модульности при построении многопроцессных программ и могут участвовать лишь в парных коммуникациях. С каждым интер-коммуникатором ассоциировано две не перекрывающихся группы процессов. Соответственно, для каждого такого процесса одна группа является локальной, другая — удаленной. При вызове какой-либо функции парного обмена данными с использованием интер-коммуникатора целевой процесс характеризуется его номером в удаленной группе.

Любой коммуникатор является либо интер-коммуникатором, либо интра-коммуникатором. Чтобы определить тип заданного коммуникатора существует функция MPI_Comm_test_inter. Информации о локальной группе интер-коммуникатора может быть получена с помощью уже упомянутых функций MPI_Comm_size, MPI_Comm_rank, MPI_Comm_group. Для получения информации об удаленной группе предусмотрены функции MPI_Comm_remote_size и MPI_Comm_remote_group.

Новый интер-коммуникатор создается на основе двух интра-коммуникаторов с помощью функции MPI_Intercomm_create. Группы процессов, ассоциированные с исходными интра-коммуникаторами, не должны пересекаться. В каждой группе выделяется так называемый «лидер» — процесс, выступающий от имени соответствующей группы в инициализационных коммуникациях. Оба лидера должны входить в группу некоторого третьего интра-коммуникатора, который также используется при инициализации и, вместе с рангами лидеров, указывается при вызове MPI_Intercomm_create (зачастую используется MPI_COMM_WORLD или его копия). Функция MPI_Intercomm_merge производит в некоторой степени обратную операцию, объединяя локальную и удаленную группы интер-коммуникатора и формируя на выходе интра-коммуникатор. Как и обычные коммуникаторы, интер-коммуникаторы могут дублироваться функцией MPI_Comm_dup, а после использования должны уничтожаться функцией MPI_Comm_free.

### Парный обмен данными

Для осуществления межпроцессного взаимодействия MPI предоставляет, прежде всего, функции парного обмена данными, т.е. функции передачи данных типа «один к одному». Во всех функциях парного обмена указывается коммуникатор, в контексте которого производится соответствующая операция, и ранг целевого процесса (которому отправляются или от которого принимаются данные) в рамках этого коммуникатора.

Среди функций парного обмена доступны блокирующие и неблокирующие операции. При блокирующем вызове операций отправки или приема буфер данных, адрес которого был передан соответствующей функции в параметрах, может быть использован сразу после возврата из нее. В случае отправки это означает, что данные из буфера уже считаны, и он может быть использован повторно, в случае же успешного приема в нем уже содержатся принятые данные. Чтобы отправить данные другому процессу, некоторый процесс исполь-

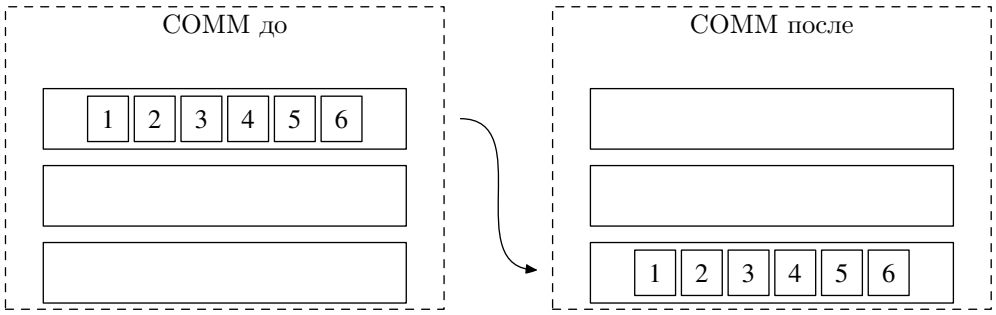



Рис. 1.4. Простая передача данных от одного процесса к другому в рамках группы процессов некоторого коммуникатора `COMM`

зует вызов функции `MPI_Send` (или один из трех ее вариантов, описание которых мы здесь опускаем). Другой процесс, чтобы получить эти данные, вызывает функцию `MPI_Recv`) (рис. 1.4). Во многих случаях процессу требуется получить информацию о принимаемом сообщении, но пока не осуществлять его прием. К примеру, это может быть удобно для того, чтобы выделить область памяти необходимого размера. Для получения информации о принимаемом сообщении используется функция `MPI_Probe`, после чего размер сообщения может быть вычислен с помощью функции `MPI_Get_count`.

Для осуществления неблокирующего обмена интерфейс предоставляет аналогичный набор функций: `MPI_Isend` с ее вариантами, `MPI_Irecv`, `MPI_Iprobe`. После вызова неблокирующей функции возврат управления клиентскому коду производится без ожидания завершения соответствующей операции. В связи с этим буфер данных, переданный функции в качестве аргумента, нельзя использовать до тех пор, пока соответствующая операция не завершится. В момент вызова неблокирующей операции создается объект-запрос, характеризующий операцию, который в дальнейшем используется для определения ее статуса. Дождаться завершения какой-либо неблокирующей операции можно путем вызова функции `MPI_Wait`, с помощью же функции `MPI_Test` можно определить факт завершения операции, не выполняя ожидания. Аналогичным путем можно дождаться завершения одной, нескольких или же сразу всех неблокирующих операций отправки/приема из некоторого заданного множества (функции `MPI_Waitany`, `MPI_Waitsome`, `MPI_Waitall`) или проверить факт их завершения (функции `MPI_Testany`, `MPI_Testsome`, `MPI_Testall`). При завершении некоторой неблокирующей операции функция ожидания или проверки завершения освобождает соответствующий объект-запрос. Если же дожидаться завершения не требуется, этот объект может быть явно помечен на освобождение с помощью функции `MPI_Request_free`.

Помимо вопросов производительности, использование неблокирующих функций позволяет избежать возникновения взаимоблокировок в ситуациях, когда каждый процесс в рамках некоторой комплексной операции обмена должен совершить более одной операции отправки или приема. К примеру, когда несколько процессов одновременно обмениваются данными по кольцевой топологии (каждый получает данные от соседа с одной стороны и отправляет соседу с другой стороны), с этой целью может быть использована ставшая идиоматической последовательность вызовов функций `MPI_Irecv`, `MPI_Send` и `MPI_Wait`. Но, поскольку подобная ситуация возникает довольно часто, для сокращения записи интерфейс предлагает комбинированную функцию отправки/приема `MPI_Sendrecv`. Часто бывает так, что при этом отправленные данные уже не нужны, а принимаемые данные совпадают с отправленными по типу и размеру, поэтому интерфейс также предусматривает функцию `MPI_Sendrecv_replace`, которая помещает принятые данные в тот же буфер, из которого ушли отправленные.

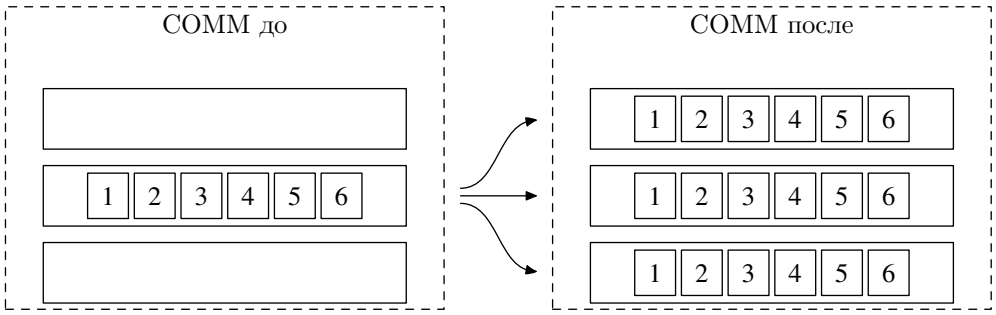



Рис. 1.5. Широковещательная рассылка данных (операция Bcast)

Наконец, в рамках парного обмена интерфейс предусматривает функции работы с так называемыми стойкими запросами (часто их называют также отложенными). Стойкий (persistent) запрос сродни объекту-запросу, создаваемому неблокирующими функциями. Отличие его в том, что, во-первых, при его создании выполнение соответствующей операции не начинается, во-вторых, при завершении операции он не освобождается соответствующей функцией ожидания или проверки завершения, в связи с чем может быть использован многократно. Цель использования таких запросов заключается в сокращении накладных расходов при выполнении однотипных операций обмена. Создание стойких запросов на отправку и прием данных выполняется соответственно функциями `MPI_Send_init` (и тремя ее вариантами) и `MPI_Recv_init`. Запуск такой операции осуществляется путем вызова функции `MPI_Start` или, если запросов много, функцией `MPI_Startall`. После завершения соответствующей операции обмена (информация о чем может быть получена с помощью упомянутых ранее функций ожидания или проверки завершения) соответствующий буфер данных может быть считан или модифицирован, и операция может быть запущена повторно. Освобождение стойких запросов всегда выполняется явно путем вызова `MPI_Request_free`.

### Коллективные коммуникации

Помимо парного обмена данными, MPI предоставляет функции коллективных коммуникаций. Под коллективными коммуникациями подразумевается выполнение некоторых операций сразу всеми процессами из группы некоторого коммуникатора. Функции коллективных коммуникаций можно условно разбить на три группы.

К первой группе относится только одна функция — `MPI_Barrier`. Она осуществляет барьерную синхронизацию всех процессов коммуникатора, т.е. не возвращает вызвавшему ее процессу управление до тех пор, пока все оставшиеся процессы коммуникатора также ее не вызовут. Функция `MPI_Barrier` вынесена в отдельную группу по той причине, что, хоть и относится к коллективным коммуникациям, операцией передачи данных в явном виде не является.

Ко второй группе относятся операции коллективной передачи данных. В рамках выполнения каждой такой операции какие-либо данные некоторым образом перераспределяются между процессами коммуникатора.

Самая простая операция коллективного обмена данными — `MPI_Bcast`. Она выполняет рассылку типа «один ко многим» некоторой области данных из какого-либо процесса всем остальным процессам коммуникатора. На рис. 1.5 изображен пример размещения некоторых данных в группе процессов до вызова `MPI_Bcast` и после него.

Следующей операцией, также выполняющей рассылку типа «один ко многим», явля-

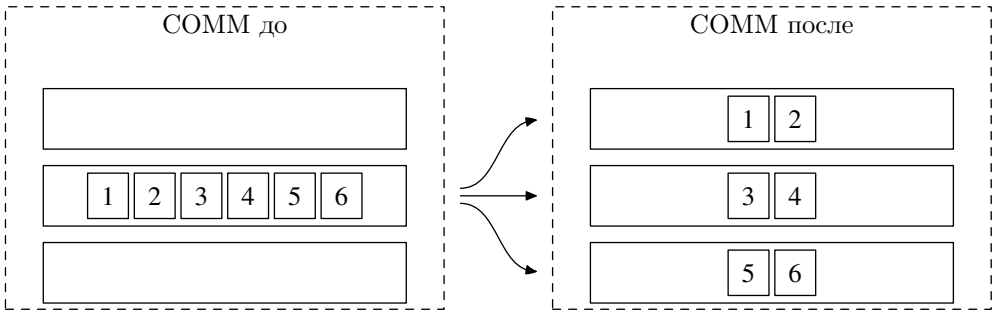



Рис. 1.6. «Рассеяние» данных (операция Scatter)

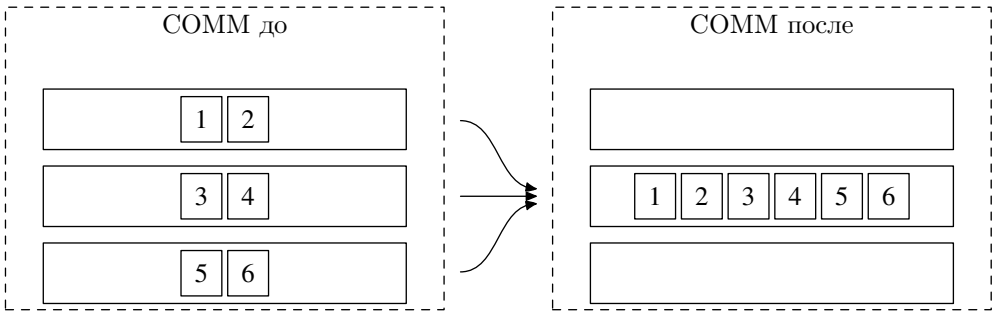



Рис. 1.7. «Сбор» данных (операция Gather)

ется операция `MPI_Scatter`. В отличие от `MPI_Bcast`, она выполняет пересылку в каждый процесс не всей области данных, а лишь ее части (рис. 1.6). При этом область данных разбивается на соответствующие части равномерно. Для случая, когда необходимо «рассеивать» данные между процессами частями разных размеров, предусмотрен другой вариант той же функции — `MPI_Scatterv`.

В некотором роде обратной к `MPI_Scatter` является функция `MPI_Gather`. Она выполняет передачу типа «много к одному» и осуществляет пересылку некоторых областей данных одинаковых размеров из всех процессов коммуникатора в один из них, формируя в нем непрерывный массив (рис. 1.7). Для случая, когда требуется «собрать» в процессе области данных разных размеров, предусмотрена функция `MPI_Gatherv`.

Если требуется, чтобы результат операции `MPI_Gather` попал не в один, а во все процессы коммуникатора, в принципе, можно после ее вызова использовать функцию `MPI_Bcast`. Но такой вариант увеличивает время на коммуникации между процессами и лишает систему возможности оптимизировать их выполнение. Поэтому для выполнения такой операции предусмотрена единая функция `MPI_Allgather`, осуществляющая обмен данными типа «много ко многим» (рис. 1.8). Также для случая исходных областей разных размеров, как и ранее, предусмотрен вариант функции `MPI_Allgatherv`.

Самая, пожалуй, «тяжеловесная» коллективная операция пересылки осуществляется функцией `MPI_Alltoall`. Она выполняет перераспределение данных между процессами по типу «много ко многим». В спецификации присутствует некоторый намек на то, что эта функция выполняет, по сути, комбинацию операций `MPI_Scatter` и `MPI_Gather`. Однако это, скорее, интуитивное описание, поскольку, как мы уже ранее упоминали, `MPI_Gather` — операция, обратная к `MPI_Scatter`, и требуемого результата мы таким образом не получим. При выполнении операции `MPI_Alltoall` данные из каждого процесса разбиваются на равные части по количеству процессов (напоминает `MPI_Scatter`), после чего каждый процесс

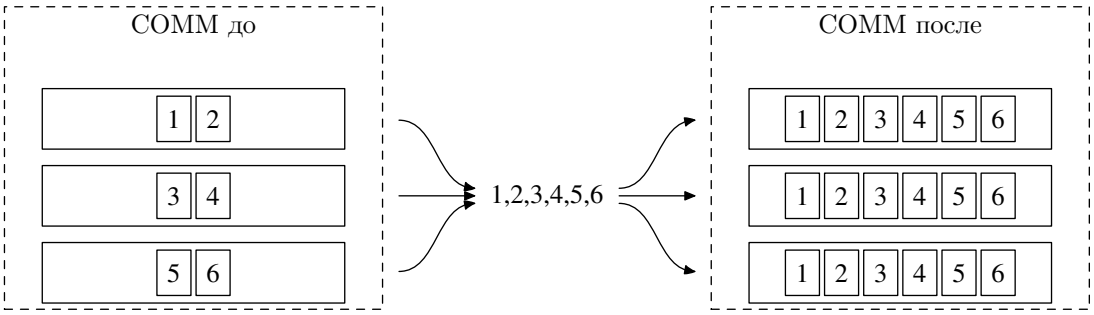



Рис. 1.8. «Сбор» данных всеми процессами (операция Allgather)

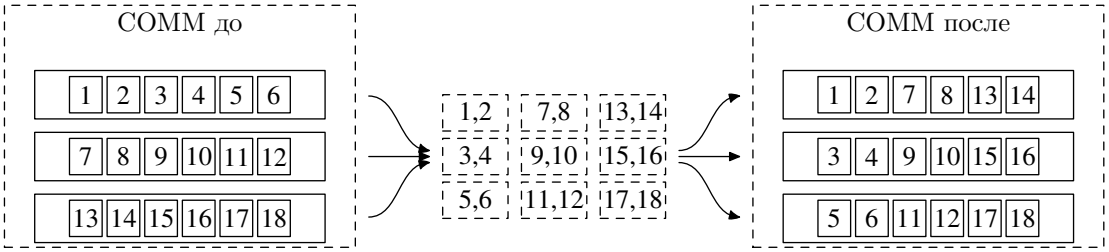



Рис. 1.9. Рассылка данных от каждого к каждому (операция Alltoall)

собирает все куски, ему предназначенные (близко к `MPI_Gather`) (рис. 1.9). Альтернативно выполнить действие, осуществляемое функцией `MPI_Alltoall`, можно было бы циклом по количеству процессов из вызовов `MPI_Scatter` со сдвигом в буфере назначения. Другой вариант — аналогичный цикл из вызовов `MPI_Gather` со сдвигом в буфере-источнике. Как и ранее, спецификация предусматривает расширенный вариант функции `MPI_Alltoallv`.

К третьей группе коллективных коммуникаций относятся операции редуцирования. Редукционные функции осуществляют выполнение некоторой ассоциативной и, возможно, коммутативной операции над распределенными между процессами данными. В качестве таких операций могут выступать сложение, умножение, поиск максимума, логические или битовые операции «И»/«ИЛИ» и т.п. Спецификацией предопределено более десятка подобных операций. Плюс к этому программа может определить свою операцию с помощью функции `MPI_Op_create`, которая после использования должна быть уничтожена функцией `MPI_Op_free`. Каждая операция имеет свой идентификатор, указываемый в качестве параметра вызываемой редукционной функции.

Простейшая редукционной функция — `MPI_Reduce`. При ее вызове указанная операция редуцирования выполняется над соответствующими элементами массивов данных из различных процессов. Результатом является массив того же размера, помещаемый в указанный при вызове процесс. На рис. 1.10 изображен пример результата выполнения `MPI_Reduce` с указанием операции `MPI_SUM`.

В случае, если необходимо поместить результат выполнения `MPI_Reduce` не в один процесс, а во все, можно использовать последующий вызов `MPI_Bcast`. Однако этот подход, как уже было указано выше, был бы не слишком эффективен, в связи с чем интерфейсом предусмотрена функция `MPI_Allreduce`, выполняющая такое действие (рис. 1.11).

В некоторых случаях бывает нужно рассредоточить результат выполнения `MPI_Reduce` частями по всем процессам коммуникатора. Здесь, как и ранее, можно осуществить последующий вызов другой коллективной операции (`MPI_Scatter`), но правильнее и

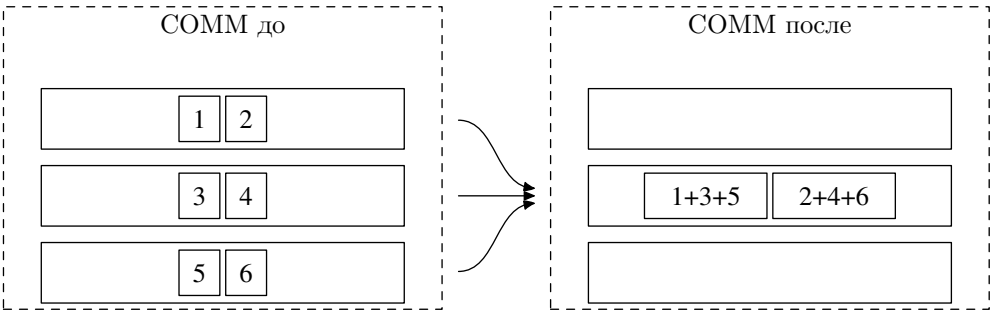



Рис. 1.10. Редуцирование данных (операция Reduce, случай суммирования)

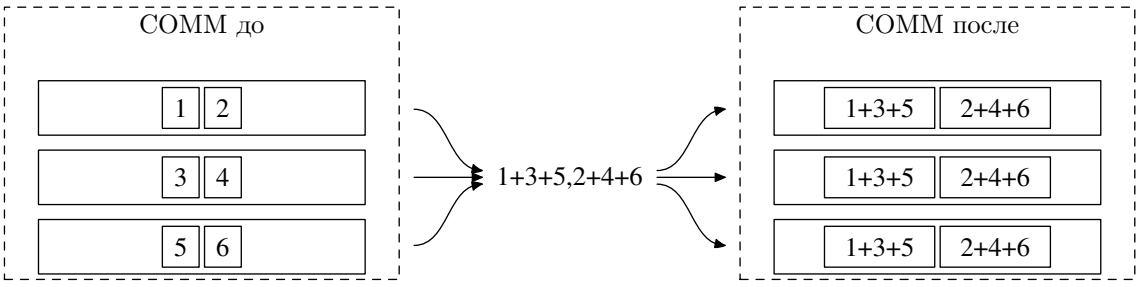



Рис. 1.11. Редуцирование данных во все процессы (операция Allreduce, случай суммирования)

эффективнее использовать специально предусмотренную для этого единую операцию `MPI_Reduce_scatter` (рис. 1.12).

Наконец, иногда бывает необходимо осуществлять в программе частичное выполнение некоторой редукционной операции (частичная сумма, частичное произведение и т.п.). Для этого интерфейсом предусмотрена функция `MPI_Scan`. Она получает на входе массивы данных одинаковой длины и выполняет соответствующую операцию над их элементами с одинаковыми индексами. На выходе `MPI_Scan` формируются массивы, элементы которых являются частичным результатом выполнения указанной операции над исходными последовательностями. К примеру, на рис. 1.13 изображен пример результата выполнения частичного суммирования.

### Работа со сложными типами данных

При выполнении коммуникационных операций процессы могут обмениваться данными как предопределенных базовых типов, так и сложных типов, определенных пользователем.

Для создания составных типов данных предусмотрено несколько последовательно усложняющихся функций (рис. 1.14). Самая простая из них — `MPI_Type_contiguous`. Она предназначена для создания типа, описывающего непрерывный массив из фиксированного количества элементов какого-либо другого типа (который также может быть составным). Более общую схему предлагает функция `MPI_Type_vector`. Создаваемый ею тип описывает набор массивов одинакового размера, отстоящих друг от друга на одинаковом расстоянии. Следующей ступенью является тип, создаваемый функцией `MPI_Type_indexed`. Он описывает набор массивов элементов некоторого типа, которые могут иметь разные размеры и отстоять друг от друга на разном расстоянии. Наконец, функция `MPI_Type_struct` предла-

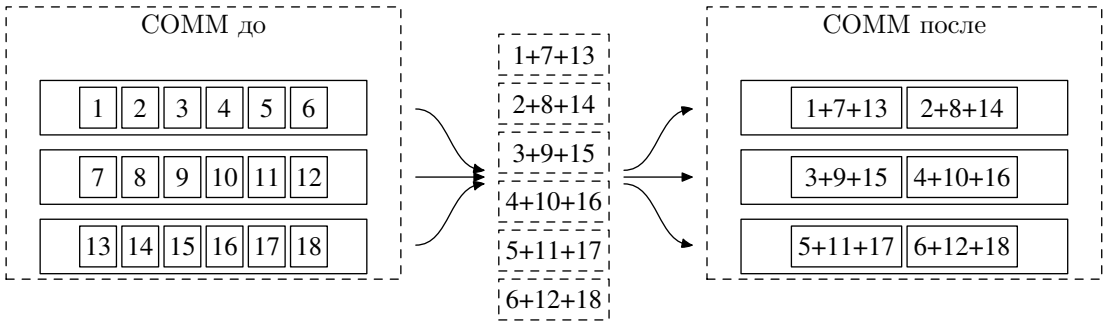



Рис. 1.12. Редуцирование данных с рассеянием (операция Reduce-scatter, случай суммирования)

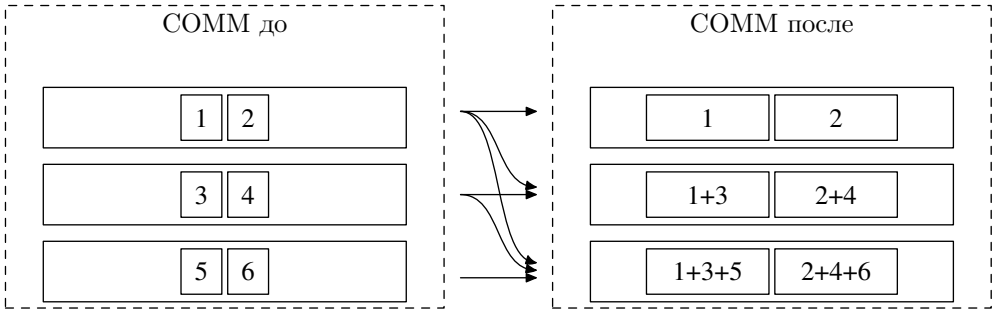



Рис. 1.13. Частичное редуцирование данных (операция Scan, случай суммирования)

гает самый общий подход к созданию произвольного типа. Полученный с ее помощью тип описывает набор массивов элементов различного размера на различном расстоянии друг от друга, при этом тип элементов, содержащихся в массивах, также может различаться.

Объекты типов, создаваемые с помощью перечисленных функций, зачастую используются программой для последовательного построения неких сложных типов данных с многоуровневой вложенностью. В таком случае коммуникационной системе не требуется знать о промежуточных типах, в связи с чем они ей этими функциями не передаются. Для учета в коммуникационной системе именно тех типов, которые будут использоваться в операциях обмена, должна быть явно вызвана функция `MPI_Type_commit`. После использования созданный программой объект типа должен быть уничтожен функцией `MPI_Type_free`.

В перечисленных схемах создания составных типов данных (рис. 1.14) расстояния, на которых отстоят друг от друга непрерывные массивы элементов, могут быть в общем случае произвольными (в т.ч. повторяющимися и даже отрицательными). Соответственно, области данных могут перекрываться и размещаться раньше базового адреса элемента соответствующего составного типа. В связи с этим возникают понятия нижней и верхней границ данных элемента соответствующего типа, представляющие смещения в байтах относительно базового адреса элемента. Получить эти значения для заданного типа можно с помощью функций `MPI_Type_lb` и `MPI_Type_ub`. Разница между этими величинами характеризует ширину в байтах минимального диапазона (с учетом выравнивания), охватывающего все включаемые в этот тип массивы данных, и возвращается функцией `MPI_Type_extent`. Функция `MPI_Type_size` возвращает суммарное количество байтов, занимаемое всеми описанными в типе данными, при этом не учитываются выров-

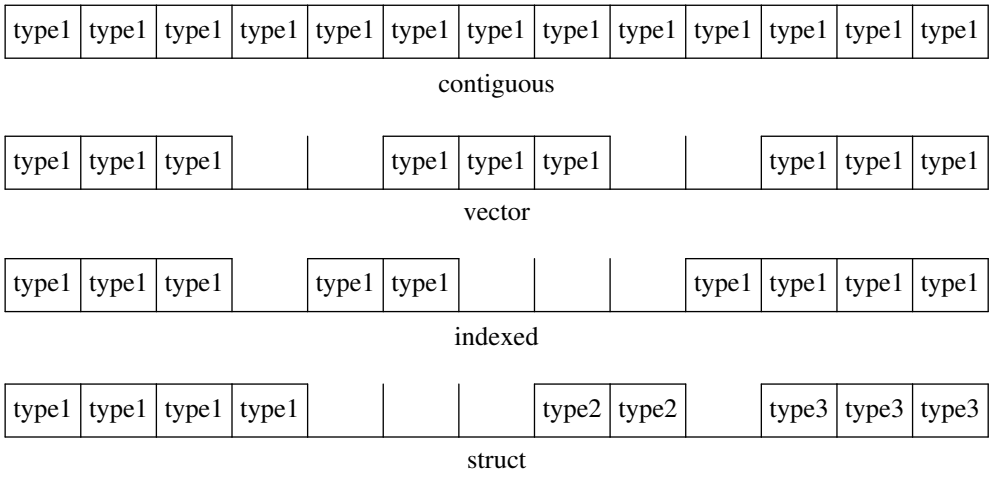



Рис. 1.14. Схемы формирования составных типов данных

нивание и возможные перекрытия областей данных. Значения, возвращаемые функциями `MPI_Type_size` и `MPI_Type_extent` могут совпадать или различаться в любую сторону. К примеру, если массивы в сложном типе не перекрываются и отстоят между собой на некотором ненулевом расстоянии, функция `MPI_Type_extent` вернет большее значение. Если же массивы перекрываются, и свободных областей между ними нет, большее значение вернет `MPI_Type_size`.

Помимо перечисленных функций формирования произвольных типов, интерфейс поддерживает функции упаковки и распаковки данных. Они позволяют формировать из произвольного набора данных с произвольным размещением непрерывную область в памяти для передачи в виде единого сообщения. По отношению к функциям формирования сложных типов это несколько более низкоуровневый интерфейс, позволяющий, порой, больше возможностей. К примеру, таким образом может производиться обмен сообщениями сложного типа с нефиксированными размерами массивов данных. Упаковка некоторого набора данных производится путем многократного вызова функции `MPI_Pack`. Результат упаковки помещается в область памяти, которая предварительно должна быть выделена программой. Для определения необходимого размера этой области программа может использовать многократный вызов функции `MPI_Pack_size`. Извлечение данных из упакованного сообщения на приемной стороне производится функцией `MPI_Unpack`.

### Прочие функции

Перед использованием любых функций системы MPI программа должна вызвать функцию `MPI_Init` для ее инициализации. Исключением является функция `MPI_Initialized`, возвращающая признак, была ли эта инициализация уже выполнена. В конце программы должна быть вызвана функция `MPI_Finalize`. Между вызовами `MPI_Init` и `MPI_Finalize` размещается параллельный участок программы, выполняемый параллельно заданным при запуске программы количеством процессов. Участки кода до вызова `MPI_Init` и после вызова `MPI_Finalize` не входят в параллельный участок и могут выполняться как всеми процессами, так и лишь одним, в зависимости от реализации MPI.

Интерфейс предусматривает функции установки обработчиков и анализа ошибок. Также спецификацией предусмотрено наличие механизма профилирования программ, т.е. оценки производительности различных частей программы с точки зрения времени работы вызываемых функций MPI. Для оценки производительности с точки зрения клиентского

кода доступны функции получения информации от таймера. Функция `MPI_Wtime` возвращает дробное число секунд, прошедшее с некоторого фиксированного момента в прошлом, что может быть использовано для оценки скорости работы различных фрагментов программы. Погрешность этой величины определяется платформой и возвращается функцией `MPI_Wtick`.

Здесь были перечислены далеко не все функции, предоставляемые интерфейсом MPI. Для ознакомления с остальными функциями, также как и для получения более детальной информации о перечисленных, рекомендуется обращаться к спецификации [71] или специально посвященной этой теме литературе [1, 2, 4, 46].

### 1.2.3. Распределение вычислений в однородной среде

Поскольку интерфейс MPI разработан специально для работы программы в распределенной системе, т.е. в рамках нескольких узлов, при работе с ним становится актуальной проблема равномерности загрузки процессов (load balancing). Далеко не все задачи могут быть легко распределены по подзадачам на узлы произвольной существующей системы. Более того, даже простые для распараллеливания задачи зачастую не могут быть распределены равномерно. К примеру, такая ситуация возникает при количестве подзадач, не кратном числу процессов, а также в случае, если имеющаяся вычислительная система неоднородна.

#### Блочное и циклическое распределение

Рассмотрим, к примеру, возможные варианты параллельного выполнения приблизительно одинаковых по длительности независимых итераций цикла. Существует два основных подхода к распределению нагрузки в подобных ситуациях — блочное и циклическое распределение [1]. При блочном распределении (рис. 1.15) итерации раздаются процессам последовательными интервалами значений их номеров. При этом длина интервала обычно является фиксированной величиной и равна ближайшему сверху целому от деления количества итераций на количество процессов, т.е. $n_k = \left\lceil \frac{n}{N} \right\rceil, k = 0, \ldots, N-2$. Последний процесс по понятным причинам получает в общем случае меньшее количество итераций, а именно $n_{N-1} = n - (N-1)\left\lceil \frac{n}{N} \right\rceil$. При распараллеливании таким способом некоторого цикла программный код, выполняемый каждым процессом с номером `rank` из группы размером в `size` процессов, реализуется следующим образом:

```
// блочное распределение итераций
int nloc = (n + (size - 1)) / size;
for (int i = nloc * rank; i < nloc * (rank + 1) && i < n; ++i)
  // ... выполнение итерации с номером i
```

При циклическом распределении (рис. 1.16) все итерации рассредоточиваются по процессам последовательным чередованием, или прореживанием по номеру итерации с шагом, равным количеству процессов:

```
// циклическое распределение итераций
for (int i = rank; i < n; i += size)
  // ... выполнение итерации с номером i
```

Также существует блочно-циклическое распределение (рис. 1.17), которое является комбинацией обоих подходов. В этом случае итерации раздаются процессам циклически, но не по одной, а некоторыми интервалами (блоками).

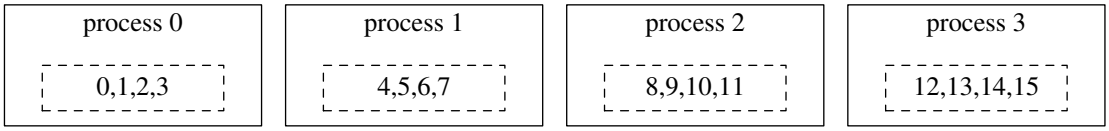



Рис. 1.15. Блочное распределение подзадач

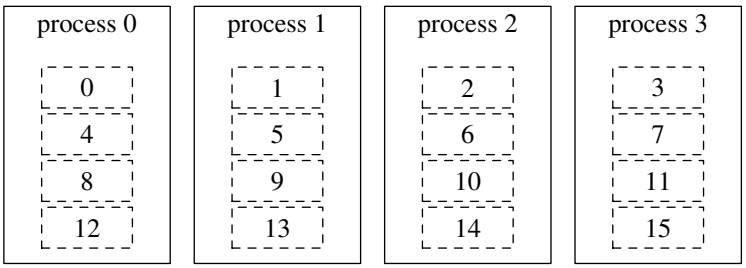



Рис. 1.16. Циклическое распределение подзадач

### Сбалансированное блочное распределение

Рассмотрим теперь, чем блочное распределение качественно отличается от циклического. Допустим, у нас есть задача, состоящая из десяти независимых подзадач примерно одинаковой вычислительной сложности, и четыре процесса на кластере из четырех узлов. Если бы мы воспользовались блочным распределением подзадач, мы бы получили распределение $\{\{0,1,2\},\{3,4,5\},\{6,7,8\},\{9\}\}$. При циклическом распределении десяти подзадач на четыре процесса мы получили бы $\{\{0,4,8\},\{1,5,9\},\{2,6\},\{3,7\}\}$.

На этом примере видно основное преимущество циклического распределения перед блочным — первое обеспечивает более равномерную загрузку в однородных системах в тех случаях, когда количество подзадач не кратно количеству процессов. Иначе говоря, достигается наименьшая разница между занятостью отдельных процессов — максимальное различие состоит лишь из одной подзадачи. Именно поэтому во избежание существенной неравномерности загрузки процессов зачастую прибегают к циклическому распределению подзадач.

Однако по различным причинам блочное распределение, порой, бывает предпочтительнее циклического, как с точки зрения программирования, так и с точки зрения производительности. Такая ситуация может возникнуть в силу специфики конкретной решаемой задачи. К примеру, если исходные данные для всех подзадач лежат в последовательном массиве, и нам необходимо распределить эти данные между узлами, удобнее и быстрее будет посылать данные нескольких последовательных подзадач одним непрерывным блоком, нежели несколькими посылками по одной подзадаче или же путем формирования нового непрерывного блока. Также при решении некоторых конкретных задач скорость выполнения итераций может быть повышена путем использования каких-либо рекуррентных соотношений, связывающих соседние подзадачи, и в этом случае нам также гораздо удобнее схема блочного распределения.

Чтобы в этом случае нам не пришлось мириться с более высокой неравномерностью распределения нагрузки по процессам, мы можем воспользоваться следующей простой схемой вычисления количества подзадач для блочного распределения. Допустим, имеется цикл в $n$ приблизительно одинаковых по длительности итераций, и требуется распределить его между $N$ процессами, причем $n$ не кратно $N$. Тогда на основе деления с остатком количество итераций может быть представлено через целые числа $a, b$ в следующей форме:

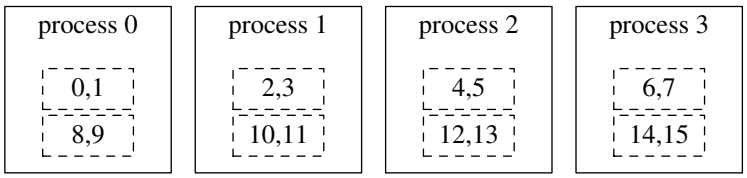



Рис. 1.17. Блочно-циклическое распределение подзадач

$$n = aN + b, \quad 0 \leqslant b < N.$$

В этом случае ширина интервала итераций для каждого процесса $n_k, k = 0, \ldots, N-1$ может быть вычислена следующим образом:

$$n_k = \begin{cases} a+1, & k < b; \\ a, & k \geqslant b. \end{cases}$$

Первые $b$ интервалов оказываются при этом шире остальных на единицу. Левая граница каждого интервала $n_k^l, k = 0, \ldots, N-1$ вычисляется на основе тех же соотношений:

$$n_k^l = \sum_{m=0}^{k} n_m - n_k = ak + \begin{cases} k, & k < b; \\ b, & k \geqslant b. \end{cases}$$

Таким образом, на каждый процесс распределяется интервал итераций $[n_k^l; n_k^l + n_k)$. Описанную схему иллюстрирует следующий код:

```
// вычисление размера и левой границы интервала индексов
int a = n / size, b = n % size;
int nloc = a + ((rank < b) ? 1 : 0);
int nleft = a * rank + ((rank < b) ? rank : b);
for (int i = nleft; i < nleft + nloc; ++i)
 // ... выполнение итерации с номером i
```

В случае применения такого подхода при решении десяти подзадач в четырех процессах будет получено распределение $\{\{0,1,2\},\{3,4,5\},\{6,7\},\{8,9\}\}$. Таким образом, описанный подход позволяет совместить удобства блочного распределения и приемлемую сбалансированность нагрузки.

### Сбалансированное распределение неравномерных итераций

При описании блочного и циклического распределения мы отталкивались от предположения, что итерации имеют приблизительно одинаковую длительность выполнения. Однако зачастую возникают задачи, в которых длительность итераций неравномерна. В общем случае балансировка нагрузки при этом может быть обеспечена, пожалуй, только динамическим распределением, однако во многих частных случаях, когда нам известна какая-то информация о соотношении длительностей итераций между собой, возможно построение и статических схем.

К примеру, рассмотрим следующую задачу. Требуется вычислить объем прямой призмы, усеченной некоторой кривой поверхностью $z = f(x, y)$ (рис. 1.18). Основанием призмы является равнобедренный прямоугольный треугольник с катетом $a$.

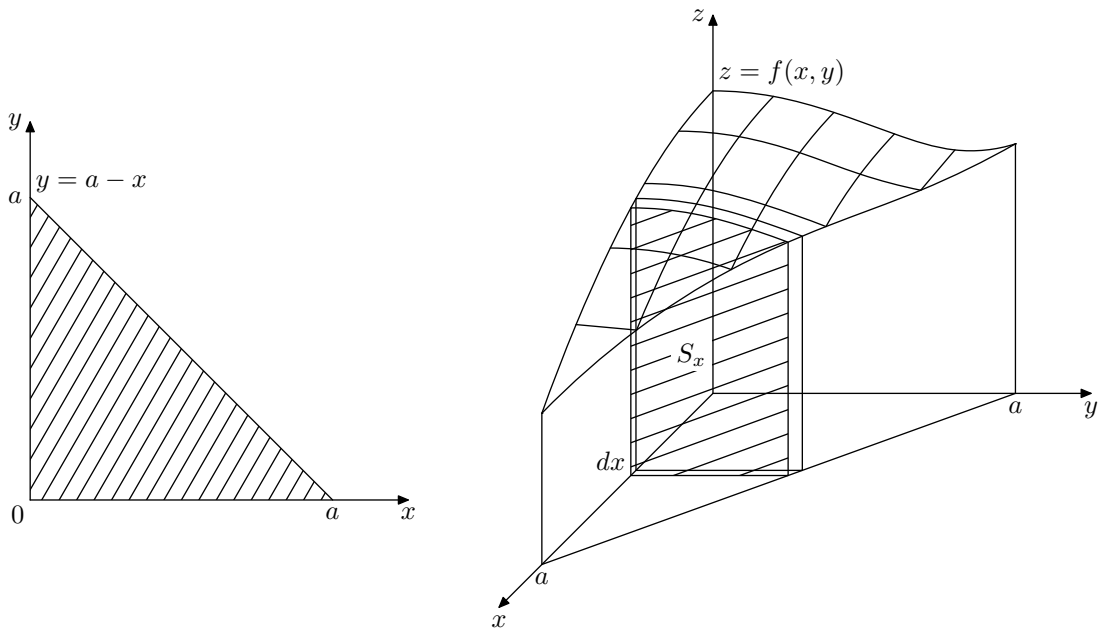



Рис. 1.18. Прямая призма, усеченная поверхностью $z = f(x, y)$

Объем $V$ такой фигуры определяется следующим интегралом:

$$V = \int_0^a S_x dx = \int_0^a \int_0^{a-x} f(x, y) dy dx. \tag{1.2}$$

Реализуем численное интегрирование, для чего введем равномерную сетку дискретизации на основании призмы и выполним конечномерную аппроксимацию интеграла (1.2) (рис. 1.19).

Будем разбивать каждый катет основания призмы на $n$ частей, деля таким образом основание на квадраты со стороной, равной шагу дискретизации $h = a/n$. Интеграл функции $f(x, y)$ на каждом квадрате аппроксимируем объемом прямоугольного параллелепипеда высотой, равной значению $f(x, y)$ в центре этого квадрата, т.е. значению $f(x_i, y_j)$:

$$x_i = (i + \frac{1}{2})h, \quad y_j = (j + \frac{1}{2})h, \quad i, j = 0, \ldots, n - 1.$$

В узлах дискретизации, размещенных вдоль гипотенузы основания призмы (рис. 1.19), интеграл $f(x, y)$ аппроксимируем половиной объема соответствующего параллелепипеда. В результате конечномерной аппроксимации получаем, что искомый объем всей призмы приблизительно равен:

$$V \approx \sum_{i=0}^{n-1} \sum_{j=0}^{n-1-i} f\left((i + \frac{1}{2})h, (j + \frac{1}{2})h\right) \cdot \begin{cases} h^2, & j < n - 1 - i; \\ \frac{h^2}{2}, & j = n - 1 - i. \end{cases}$$

Производим в полученном выражении замену $i$ на $n - 1 - i$, чтобы расположить внутренние частичные суммы в порядке возрастания количества слагаемых, и получаем более привычное представление вложенной суммы:

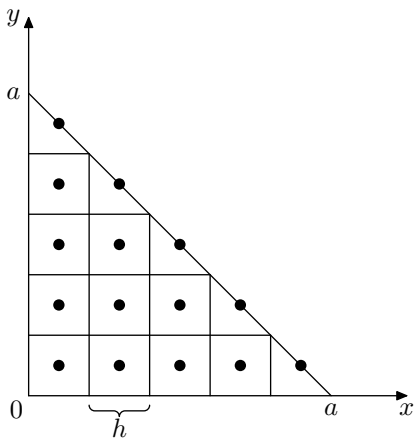



Рис. 1.19. Узлы конечномерной аппроксимации на основании призмы

$$V \approx \sum_{i=0}^{n-1} \sum_{j=0}^{i} f\left((n-i-\frac{1}{2})h, (j+\frac{1}{2})h\right) \cdot \begin{cases} h^2, & j < i; \\ \frac{h^2}{2}, & j = i. \end{cases} \tag{1.3}$$

Последовательное приближенное вычисление объема заданной фигуры на основе выражения (1.3) реализуется следующим фрагментом кода:

```
// количество разбиений
int n = /*...*/;
// сторона треугольного основания призмы
double a = /*...*/;

double h = a / n, v = 0.0;

for (int i = 0; i < n; ++i)
{
 for (int j = 0; j <= i; ++j)
 {
  double x = (n - i - 0.5) * h;
  double y = (j + 0.5) * h;
  double s = h * h * ((j < i) ? 1.0 : 0.5);
  v += f(x, y) * s;
 };
};

fprintf(stdout, "%.16f\n", v);
```

Программа содержит цикл двойной вложенности, причем границей внутреннего цикла является параметр внешнего (иногда такой двойной цикл называют «треугольным»). Вследствие полной независимости итераций внешнего цикла (в рамках каждой из которых выполняется внутренний) их выполнение может быть распараллелено. Однако мы сталкиваемся с тем обстоятельством, что длительность этих итераций неравномерна, поскольку количество итераций внутреннего цикла в каждом случае свое. В связи с этим нам необходимо построить такое распределение итераций внешнего цикла на $N$ процессов, чтобы каждый из них выполнил приблизительно одинаковое суммарное количество итераций внутреннего цикла.

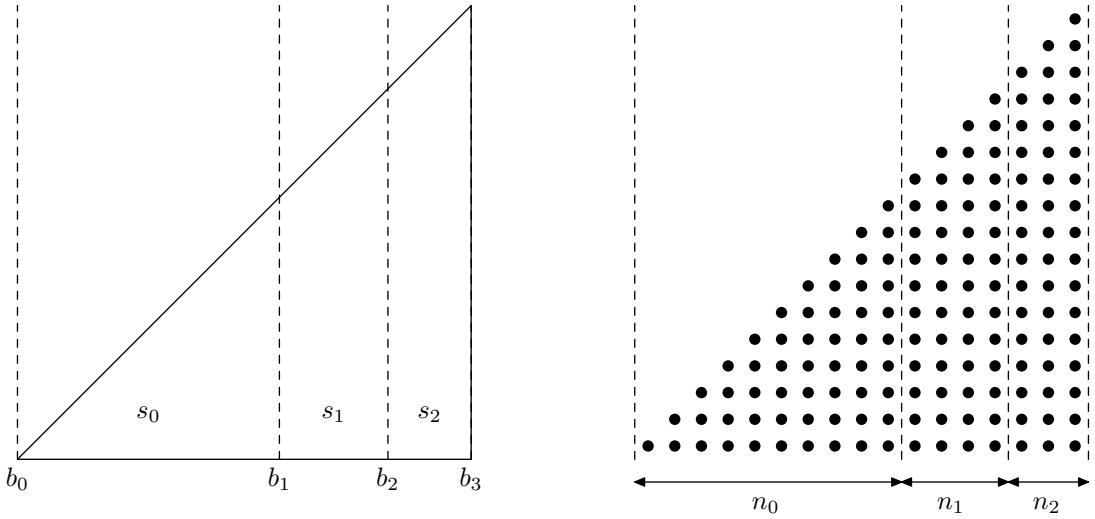



Рис. 1.20. Разбиение треугольного множества точек

Для реализации такого распределения перейдем от дискретного множества номеров итераций $(i, j)$ к непрерывному множеству точек треугольника с единичной стороной (рис. 1.20). Разобьем треугольник на $N$ частей так, чтобы площади полученных частей $s_k, k = 0, \ldots, N-1$ совпадали.

На основе соотношения частичных сумм площадей $s_k$ с площадью всего треугольника получаем границы такого разбиения $b_k$:

$$\sum_{m=0}^{k} s_m \equiv \frac{b_{k+1}^2}{2} = \frac{k+1}{N} \cdot \frac{1^2}{2} \quad \Rightarrow \quad b_k = \sqrt{\frac{k}{N}}.$$

Возвращаясь обратно к дискретному множеству номеров итераций, получаем, что для сбалансированного их выполнения на каждый $k$ процесс должен быть выделен диапазон номеров $i \in [n_k^l; n_k^r)$ (рис. 1.20):

$$n_k^l \approx n\sqrt{\frac{k}{N}}, \quad n_k^r \approx n\sqrt{\frac{k+1}{N}}, \quad k = 0, \ldots, N-1.$$

С использованием полученных выражений для вычисления границ интервалов модифицируем приведенный выше код для выполнения в среде MPI:

```
enum { MASTER_PROC_ID = 0 };

// количество разбиений
int n = /*...*/;
// сторона треугольного основания призмы
double a = /*...*/;

int rank, size;
// получение количества процессов и номера текущего процесса
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

double h = a / n, locv = 0.0;
```

```
int nleft, nright;
// вычисляем левую границу интервала
nleft = int(sqrt(1.0 * rank / size) * n);
// сдвигаем ее между процессами, получаем правую
MPI_Status status;
MPI_Sendrecv(
 &nleft, 1, MPI_INT, (rank + size - 1) % size, 0,
 &nright, 1, MPI_INT, (rank + 1) % size, 0,
 MPI_COMM_WORLD, &status);
// корректируем правую границу у последнего процесса
if (rank == size - 1)
 nright = n;

for (int i = nleft; i < nright; ++i)
{
 for (int j = 0; j <= i; ++j)
 {
  double x = (n - i - 0.5) * h;
  double y = (j + 0.5) * h;
  double s = h * h * ((j < i) ? 1.0 : 0.5);
  locv += f(x, y) * s;
 };
};

// объединяем суммы
double v = 0.0;
MPI_Reduce(
 &locv, &v, 1, MPI_DOUBLE,
 MPI_SUM, MASTER_PROC_ID, MPI_COMM_WORLD);

// вывод результата в одном процессе
if (rank == MASTER_PROC_ID)
 fprintf(stdout, "%.16f\n", v);
```

На разных узлах возможны какие-либо различия в механизмах вычислений с плавающей запятой, в связи с чем в результате округления в разных процессах в принципе могут быть ошибочно получены различные значения для одной и той же границы. Чтобы защититься от возможных в такой ситуации потерь или, наоборот, дублирования итераций каждый процесс вычисляет лишь одну границу, вторую же получает от процесса-соседа, для чего используется кольцевая рассылка с помощью вызова `MPI_Sendrecv`. В этой ситуации последний процесс получает нулевую правую границу, в связи с чем заменяет ее корректным значением. После выполнения всеми процессами соответствующего интервала итераций полученные значения объемов промежуточных призм суммируются с помощью функции `MPI_Reduce`, результат помещается в главный процесс.

Подобные схемы сбалансированного распределения неравномерных итераций могут быть построены и во многих других ситуациях. Данная же схема разбиения может быть использована также, к примеру, при выполнении параллельного умножения треугольной матрицы на вектор.

### 1.2.4. Некоторые вопросы распределения в неоднородной среде

Иногда может потребоваться разработать программу, которая должна будет работать в неоднородной системе, т.е. системе, в которую входят узлы с разной производительностью.

К примеру, это может быть сеть из нескольких машин с разными характеристиками. В этом случае равномерное по времени выполнения распределение нагрузки на узлы вычислительной системы снова ложится на плечи программиста.

### Статическое распределение

В общем случае задача равномерного распределения в неоднородной среде может быть решена только путем динамического распределения нагрузки, однако во многих частных случаях можно приблизительно оценить производительность каждого узла и распределить нагрузку статически в соответствии с полученными оценками. Такая эвристическая оценка может быть выполнена перед началом вычислений на основе замера времени выполнения всеми узлами некоторой небольшой типичной для всей задачи операции, многократное выполнение которой в ходе вычислений занимает подавляющее время.

К примеру, если у нас стоит задача перемножения двух плотных матриц, перед началом распределения подзадач каждым вычислительным узлом может быть выполнен замер времени выполнения операции умножения матрицы на вектор. Или, к примеру, когда стоит задача умножения матрицы на вектор, нужная оценка может быть выполнена на основе замера времени вычисления каждым узлом скалярного произведения.

Когда результаты замеров на всех узлах получены, может быть выполнено распределение большого количества подзадач по узлам в соответствии с полученными оценками производительности.

Пусть каждый процесс выполнил свою операцию-образец за время $T_k, k = 0, \ldots, N-1$. Тогда скорость выполнения таких операций каждым процессом в единицу времени составляет $1/T_k$, суммарная же скорость их выполнения всеми процессами одновременно равна $\sum_{m=0}^{N-1} 1/T_m$. Если за весь период работы совокупности процессов должно быть выполнено $n$ задач, распределенное на каждый конкретный процесс их количество должно быть равно приблизительно:

$$n_k \approx \frac{1/T_k}{\sum_{m=0}^{N-1} 1/T_m} n.$$

Разумеется, в этой ситуации не избежать неточностей. В результате округлений $n_k$ до целых чисел может возникнуть ситуация, когда $\sum_{k=0}^{N-1} n_k \neq n$. Проблема может быть решена путем вычисления в каком-либо конкретном процессе (к примеру, нулевом) суммы всех $n_k$ и корректировки локальной ширины интервала (значения $n_0$) на величину $n - \sum_{k=0}^{N-1} n_k$.

Правая $n_k^r$ и левая $n_k^l$ границы каждого интервала могут быть вычислены на основе полученных значений $n_k$ в виде их частичной суммы:

$$n_k^r = \sum_{m=0}^{k} n_m, \quad n_k^l = n_k^r - n_k, \quad k = 0, \ldots, N-1.$$

При этом из всех итераций $[0; n)$ на каждый $k$ процесс распределяется интервал итераций $[n_k^l; n_k^r)$. Описанный механизм иллюстрируется кодом, приведенным ниже:

```
#define EPS     0.01
enum { MASTER_PROC_ID = 0 };

int rank, size;
double span, speed, allspeed;
int nloc, allnloc, nright;
```

```
// получение количества процессов и номера текущего процесса
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// вычисление скорости
span = MPI_Wtime();
// ... оценочная итерация
span = MPI_Wtime() - span;
speed = 1.0 / span;
// проверка относительной погрешности
if (MPI_Wtick() / span > EPS)
 fprintf(stderr, "warning: sample iteration is too small\n");

// вычисление nloc
MPI_Allreduce(
 &speed, &allspeed, 1, MPI_DOUBLE,
 MPI_SUM, MPI_COMM_WORLD);
nloc = int(floor(n * (speed / allspeed) + 0.5));
// корректировка nloc для rank == 0, если требуется
MPI_Reduce(
 &nloc, &allnloc, 1, MPI_INT,
 MPI_SUM, MASTER_PROC_ID, MPI_COMM_WORLD);
if (rank == MASTER_PROC_ID && allnloc != n)
 nloc += n - allnloc;
// вычисление правой границы интервала индексов
MPI_Scan(
 &nloc, &nright, 1, MPI_INT,
 MPI_SUM, MPI_COMM_WORLD);

for (int i = nright - nloc; i < nright; ++i)
 // ... выполнение итерации с номером i
```

Очевидно, подобный способ оценки производительности для распределения нагрузки в неоднородных системах является весьма приблизительным, вследствие чего расхождения по полному времени вычислений между процессами все равно будут возникать. К примеру, может подвести выбор элементарной операции для оценки или наличие фоновых процессов. Однако даже в такой ситуации это расхождение в неоднородных системах, как правило, будет гораздо ниже, чем при равномерном распределении. Иные же пути оценки производительности (к примеру, по отношению тактовых частот) могут еще менее соответствовать реальности вследствие различий во внутренней архитектуре процессоров. Оптимальным здесь, вероятно, является определение соотношений вычислительных скоростей опытным путем на реальных задачах (а не тестовых операциях) с последующим сохранением этих величин в качестве конфигурационных параметров для использования в дальнейших схожих вычислениях. Подобная экспериментальная схема оценки сродни использованию методов имитационного моделирования, которые широко применяются на практике, несмотря на потенциальную неточность при малом количестве экспериментов.

### Динамическое распределение

Все рассмотренные нами варианты распределения нагрузки являются статическими, т.е. распределение нагрузки в них происходит один раз на весь период выполнения. На практике зачастую также используется динамическое распределение. В этой ситуации распределение очередных подзадач на процессы производится динамически по мере выпол-

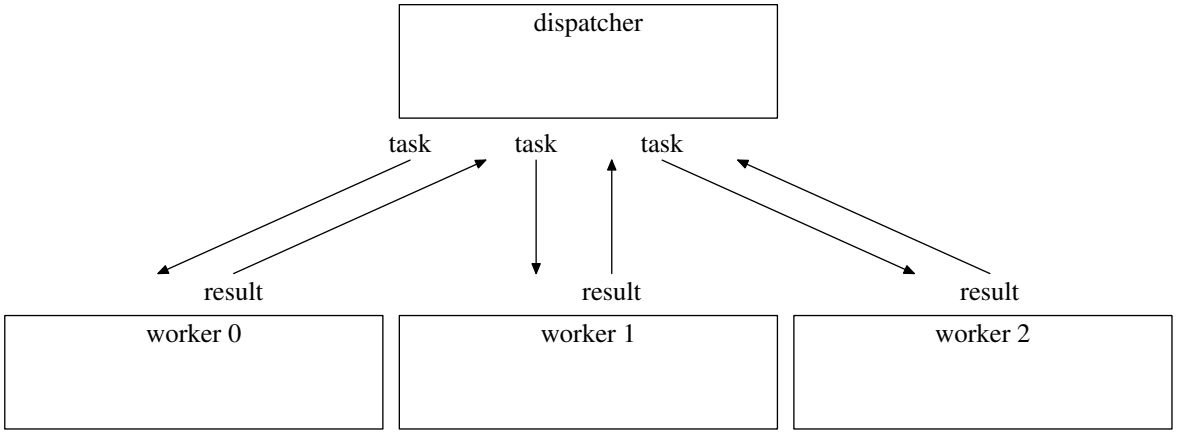



Рис. 1.21. Схема взаимодействия процессов при динамическом распределении подзадач

нения ими предыдущих. Такое распределение наиболее удобно для достижения одновременного завершения выполнения подзадач всеми процессами, особенно в неоднородных системах. Однако динамическое распределение, как правило, требует гораздо больше коммуникаций между процессами и соответствующих потерь времени.

Возникают, порой, и ситуации, когда подзадачи не только не равны, но и недетерминированы по длительности, т.е. мы не знаем наперед, как между собой соотносятся длительности выполнения подзадач. В таких случаях без динамического распределения нагрузки обойтись не получается. При этом, как правило, бывает удобно выделить один процесс, который будет заниматься распределением подзадач на остальные процессы с последующим сбором результатов (рис. 1.21).

Довольно часто бывает так, что некоторому процессу оказывается проще выполнить какую-либо работу самому, чем передавать данные для ее выполнения другому процессу и получать от него результат. К примеру, при наличии не самой быстродействующей сети управляющему процессу может оказаться гораздо проще и быстрее вычислить скалярное произведение двух векторов самостоятельно, нежели поручать его вычисление процессу на другом узле с соответствующей передачей данных в обе стороны.

Вследствие этого возникает такая ситуация, что динамическое распределение нагрузки, при всей своей перспективности с точки зрения возможности управления нагрузкой во время выполнения, во многих ситуациях оказывается менее быстродейственным вариантом по сравнению со статическим распределением. Выходом может являться укрупнение блоков операций, распределяемых динамически, т.е. увеличение размеров решаемой за один заход подзадачи с соответствующим сокращением коммуникаций. Однако это имеет и обратную сторону: чем крупнее подзадача, тем больше потенциальное расхождение во времени завершения выполнения всех подзадач между процессами, т.е. в общем случае снова получаем высокую неравномерность их загрузки. Подробнее о вариантах динамического распределения нагрузки (самопланирующие алгоритмы) можно прочитать в [46].

Представленные в дальнейшем изложении примеры мы для простоты будем приводить исходя из предположения, что система однородна и что нагрузка равномерна (в частности, размер задачи кратен числу процессов). В противном случае они могут быть видоизменены с учетом информации предыдущих разделов.

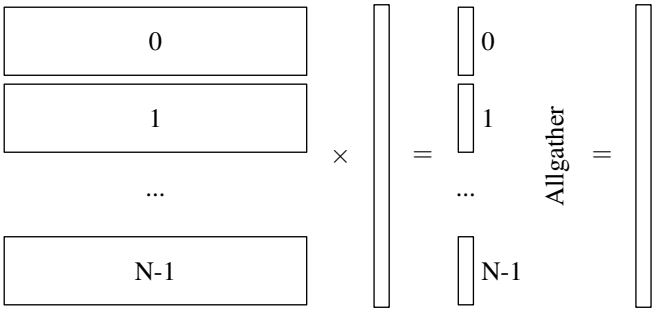



Рис. 1.22. Схема распределенного умножения матрицы на вектор в случае разбиения матрицы горизонтальными блоками

### 1.2.5. Умножение матрицы на вектор

Рассмотрим процедуру, необходимость выполнения которой нередко возникает при решении реальных вычислительных задач, к примеру, при решении итерационными методами систем линейных алгебраических уравнений (СЛАУ).

В таких задачах наиболее ресурсоемкой является операция умножения матрицы на вектор, поэтому именно ее распараллеливание в наибольшей степени повышает скорость вычислений. Помимо этого, когда речь идет о распараллеливании программы в системе с распределенной памятью, бывает необходимо использовать возможность распределения между узлами не только выполнения, но и данных, поскольку зачастую вследствие размеров решаемой задачи ее данные не умещаются в оперативной памяти одной машины. По этим причинам мы распределим по процессам также и хранение умножаемой на вектор матрицы.

Распределенное хранение матрицы возможно множеством вариантов, мы же здесь рассмотрим три наиболее простых из них. Для простоты описания будем предполагать, что размерность матрицы $n$ кратна количеству процессов $N$. Величину $m = n/N$ будем далее называть локальной размерностью.

В первом рассматриваемом нами случае элементы матрицы распределяются по процессам горизонтальными блоками (рис. 1.22). Каждый процесс содержит в своей памяти один горизонтальный блок матрицы и полную копию вектора, на который производится умножение. Процесс выполняет перемножение прямоугольного блока матрицы $m \times n$ и вектора, в результате чего получает вектор локальной размерности $m$, соответствующий части требуемого вектора результата.

После выполнения всеми процессами такой операции все локальные результаты должны быть объединены в полноразмерный вектор результата умножения для дальнейшего использования. В частности, при решении СЛАУ итерационными методами вектор, как правило, должен быть скопирован в память каждого процесса для выполнения дальнейших итераций. Возможность объединить все части вектора и скопировать результат во все процессы предоставляет функция `MPI_Allgather`. Следующий код демонстрирует описанную процедуру:

```
int nloc = n / size;
// матрица (локальная часть)
matrix_type<double> aloc(nloc, n);
// вектор-множитель и результат
vector_type<double> u(n), au(n);

// ... инициализация aloc и u
```

```
// умножение
vector_type<double> auloc = aloc * u;

// сборка частей
MPI_Allgather(
 &auloc(0), auloc.vsize(), MPI_DOUBLE,
 &au(0), nloc, MPI_DOUBLE,
 MPI_COMM_WORLD);
```

Здесь и в дальнейшем при описании программ, работающих с матрицами и векторами, используется специально написанный для этого шаблон класса `matrix_type`, а также унаследованный от него шаблон `vector_type`, являющийся частным случаем матрицы с одним столбцом. Программный интерфейс, предоставляемый этими шаблонами, определен следующим образом:

```
// матрица
template <typename e_t>
class matrix_type
{
public:
 typedef e_t element_type;
 // конструктор (инициализация значениями element_type())
 matrix_type(int vsize, int hsize);
 // размеры по вертикали и горизонтали
 int vsize(void) const;
 int hsize(void) const;
 // обращение к элементам по индексам (нумерация с нуля)
 const element_type & operator ()(int i, int j) const;
 element_type & operator ()(int i, int j);
 // прибавление матрицы
 matrix_type & operator +=(const matrix_type &src);
 // вычитание матрицы
 matrix_type & operator -=(const matrix_type &src);
 // сумма двух матриц
 friend
 matrix_type operator +(
  const matrix_type &src1, const matrix_type &src2);
 // разность двух матриц
 friend
 matrix_type operator -(
  const matrix_type &src1, const matrix_type &src2);
 // перемножение двух матриц
 friend
 matrix_type operator *(
  const matrix_type &src1, const matrix_type &src2);
};

// вектор - матрица в один столбец
template <typename e_t>
class vector_type: public matrix_type<e_t>
{
public:
 typedef matrix_type<e_t> base_type;
 typedef typename base_type::element_type element_type;
 // конструктор (инициализация значениями element_type())
```

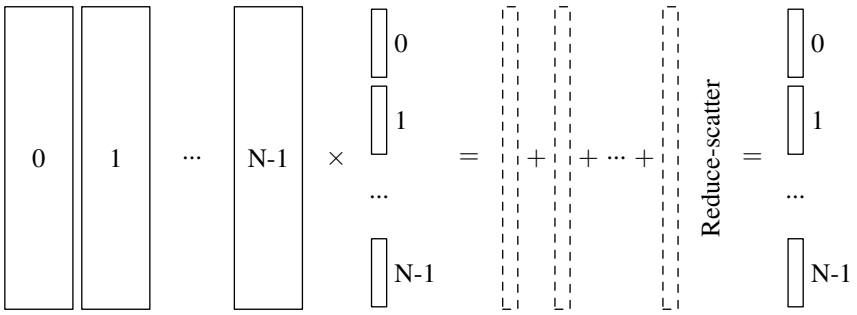



Рис. 1.23. Распределенное умножение матрицы на вектор с разбиением матрицы вертикальными блоками

```
 vector_type(int vsize);
 // конструктор - преобразование типа
 vector_type(const base_type &src);
 // обращение к элементам по индексу (нумерация с нуля)
 const element_type & operator ()(int i) const;
 element_type & operator ()(int i);
 // присваивание матрицы в один столбец
 vector_type & operator =(const base_type &src);
};
```

Помимо функций, описанных явным образом, присутствует также требование наличия корректно работающих конструктора копирования и оператора присваивания. Основное требование к реализации, необходимое нам для обмена данными между процессами, — построчное хранение элементов в непрерывном массиве. Это требуется для обеспечения возможности обмена данными с использованием функций MPI. По тем же причинам операция чтения элемента по индексу возвращает не значение элемента, а ссылку непосредственно на хранимый элемент. Индексация элементов матриц и векторов в нашем случае выполняется с нуля. Это идет вразрез правилами, с принятыми в математике, однако отвечает идиоматическому стилю языков C и C++, в связи с чем во многих случаях позволяет не нагромождать код приращениями индексов и сделать его более читабельным. Тот факт, что индексация элементов в программе выполняется с нуля, следует иметь в виду в последующем при сопоставлении индексов в формулах и в программах. Полный текст примера реализации таких шаблонов можно найти в приложении А.

При вызове функции `MPI_Allgather` мы должны передать адреса массивов отправляемых и принимаемых данных. В приведенном ранее фрагменте используется условие хранения элементов матрицы в непрерывном массиве. В обоих случаях мы передаем адрес первого элемента вектора (с нулевым индексом), который является адресом начала соответствующего массива. Такой способ передачи данных матриц и векторов между процессами будет использоваться нами и в дальнейшем.

Теперь рассмотрим случай распределения элементов матрицы по процессам вертикальными блоками (рис. 1.23). Каждому процессу при выполнении умножения требуется наличие одного локального блока матрицы размерности $n \times m$ и одного локального блока вектора, на который производится умножение. Каждый процесс выполняет перемножение обеих своих локальных частей, после чего для получения требуемого результата полученные полноразмерные векторы из всех процессов должны быть просуммированы.

Для выполнения суммирования между процессами предусмотрена функция `MPI_Reduce`, однако, если результат должен попасть во все процессы, правильнее ис-

пользовать функцию `MPI_Allreduce`. В случае же, когда для дальнейших вычислений каждому процессу необходим не весь вектор результата, а лишь его часть (как зачастую и бывает), мы можем сократить количество передаваемых данных, использовав функцию `MPI_Reduce_scatter`. Во время выполнения этой коллективной операции все полноразмерные векторы из каждого процесса суммируются, после чего результат распределяется частями по всем процессам. Такую процедуру умножения матрицы на вектор иллюстрирует следующий код:

```
int nloc = n / size;
// матрица (локальная часть)
matrix_type<double> aloc(n, nloc);
// вектор–множитель и результат (локальные части)
vector_type<double> uloc(nloc), auloc(nloc);

// ... инициализация aloc и uloc

// умножение
vector_type<double> aupart = aloc * uloc;

// инициализация размеров областей для рассеивания
vector_type<int> nlocs(size);
for (int i = 0; i < nlocs.vsize(); ++i)
 nlocs(i) = nloc;
// суммирование всех aupart и рассеивание результата
MPI_Reduce_scatter(
 &aupart(0), &auloc(0), &nlocs(0),
 MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Реализованное описанными путями параллельное умножение матрицы на вектор может быть использовано при решении СЛАУ итерационными методами. К примеру, последовательное решение СЛАУ $\bar{u} - A\bar{u} = \bar{f}$ с неизвестным вектором $\bar{u}$ методом простой итерации реализуется следующей программой:

```
// матрица
matrix_type<double> a(n, n);
for (int i = 0; i < a.vsize(); ++i)
 for (int j = 0; j < a.hsize(); ++j)
  a(i, j) = /* ... инициализация матрицы */;
// вектор свободных коэффициентов
vector_type<double> f(n);
for (int i = 0; i < f.vsize(); ++i)
 f(i) = /* ... инициализация вектора правой части */;

// вектор текущего приближения
vector_type<double> u = f;
// выполнение нескольких простых итераций
for (int i = 0; i < NUM_ITER; ++i)
 u = a * u + f;
```

Здесь для наглядности выполняется фиксированное количество итераций. Вообще же обычно используют оценку точности текущего приближения $\bar{u}_n$ по величине относительной невязки $\|u_n - A\bar{u}_n - \bar{f}\|/\|\bar{f}\|$.

Для распараллеливания такой программы одним из приведенных выше способов достаточно объявить вместо полноразмерных матрицы и вектора лишь необходимые локальные блоки, а также произвести умножение с использованием одного из приведенных выше

фрагментов. К примеру, для первого рассмотренного нами случая, т.е. для случая хранения матрицы горизонтальными блоками, имеем:

```
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int nloc = n / size;

// матрица (локальная часть)
matrix_type<double> aloc(nloc, n);
// ... инициализация матрицы

// вектор свободных коэффициентов
vector_type<double> f(n);
// ... инициализация вектора правой части

// вектор текущего приближения
vector_type<double> u = f;
// выполнение нескольких простых итераций
for (int i = 0; i < NUM_ITER; ++i)
{
 vector_type<double> uloc = aloc * u;
 MPI_Allgather(
  &uloc(0), uloc.vsize(), MPI_DOUBLE,
  &u(0), nloc, MPI_DOUBLE,
  MPI_COMM_WORLD);
 u += f;
};
```

Можно заметить, что в приведенном фрагменте каждый процесс хранит больше данных и выполняет больше вычислений, чем необходимо. Путем изменения порядка выполнения операций умножения и объединения вектора можно сократить как требуемую для хранения вектора память, так и количество выполняемых операций сложения. В результате процедура принимает следующий вид:

```
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int nloc = n / size;

// матрица (локальная часть)
matrix_type<double> aloc(nloc, n);
// ... инициализация матрицы

// вектор свободных коэффициентов (локальная часть)
vector_type<double> floc(nloc);
// ... инициализация вектора правой части

// вектор текущего приближения (локальная часть)
vector_type<double> uloc = floc;
// выполнение нескольких простых итераций
for (int i = 0; i < NUM_ITER; ++i)
{
 vector_type<double> u(n);
```

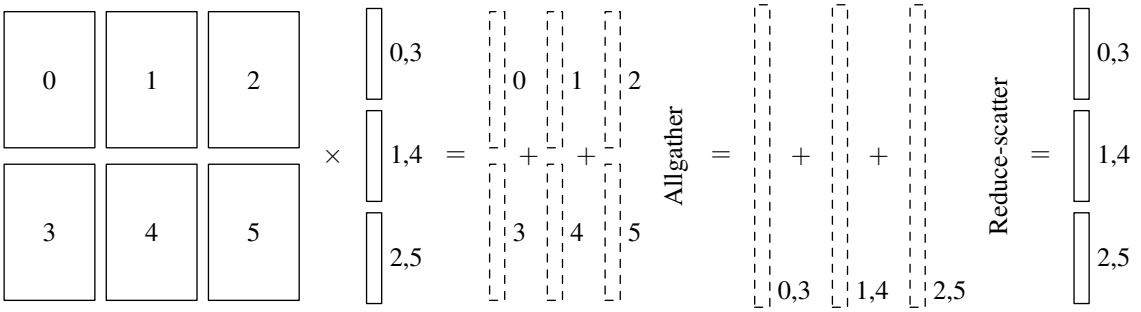



Рис. 1.24. Пример умножения матрицы на вектор в случае двумерной решетки с размерами $N^v = 2$ по вертикали и $N^h = 3$ по горизонтали

```
 MPI_Allgather(
  &uloc(0), uloc.vsize(), MPI_DOUBLE,
  &u(0), nloc, MPI_DOUBLE,
  MPI_COMM_WORLD);
 uloc = aloc * u + floc;
};
```

В этом случае каждый процесс хранит лишь локальные части вектора свободных коэффициентов и вектора текущего приближения. Объединение частей последнего выполняется непосредственно перед умножением. После выполнения заданного количества итераций полученный результат может быть собран в каком-либо процессе функцией `MPI_Gather` для дальнейшего использования.

Помимо приведенных способов распределенного хранения матрицы, разумеется, возможны и другие. В частности, в силу каких-либо обстоятельств матрицу может оказаться удобно разбить в обоих направлениях — по горизонтали и по вертикали. Пример такой ситуации будет приведен ниже при описании распределенного перемножения матриц. Множество из $N$ процессов образует в этом случае двумерную решетку, которая может быть разбита по вертикали на $N^v$ строк процессов или по горизонтали на $N^h$ столбцов процессов ($N = N^v N^h$). Тогда схема умножения распределенной матрицы на вектор может быть получена путем комбинирования описанных приемов. На рис. 1.24 приведен пример такого умножения для случая решетки из $N = 6$ процессов.

Умножаемый вектор разбит на части по количеству столбцов в решетке процессов $N^h$ и распределен между процессами так, что копия соответствующей части вектора присутствует во всех $N^v$ процессах соответствующего столбца. После выполнения умножения локальных частей матрицы и вектора каждый процесс содержит локальную часть одного из векторов размерности $n$, которые в сумме формируют вектор результата. В рамках каждого столбца решетки все $N^v$ процессов объединяют свои части, тем самым формируя в памяти каждого из них полноразмерный вектор. Полученные $N^h$ полноразмерных векторов суммируются между процессами каждой строки двумерной решетки, после чего результат разбивается на части и распределяется между ними же. В результате выполнения всей этой процедуры мы снова имеем вектор, распределенный частями между столбцами решетки, при этом каждый процесс в столбце имеет копию этой части.

Ниже приведен код решения упомянутой ранее СЛАУ методом простой итерации с использованием описанной схемы.

```
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
// задаем размеры двумерной сетки
int dims[2] = {0}, &vsize = dims[0], &hsize = dims[1];
MPI_Dims_create(size, 2, dims);

// локальные размерности по вертикали и горизонтали
int vnloc = n / vsize;
int hnloc = n / hsize;
// вертикальный и горизонтальный номера текущего процесса
int vrank = rank / hsize;
int hrank = rank % hsize;

// коммуникаторы строк процессов и столбцов процессов
MPI_Comm vcomm, hcomm;
MPI_Comm_split(MPI_COMM_WORLD, hrank, rank, &vcomm);
MPI_Comm_split(MPI_COMM_WORLD, vrank, rank, &hcomm);

// матрица (локальная часть)
matrix_type<double> aloc(vnloc, hnloc);
// ... инициализация матрицы

// вектор свободных коэффициентов (локальная часть)
vector_type<double> floc(hnloc);
// ... инициализация вектора правой части

// инициализация размеров областей для рассеивания
vector_type<int> hnlocs(hsize);
for (int i = 0; i < hnlocs.vsize(); ++i)
 hnlocs(i) = hnloc;

// вектор текущего приближения (локальная часть)
vector_type<double> uloc = floc;
// выполнение нескольких простых итераций
for (int i = 0; i < NUM_ITER; ++i)
{
 vector_type<double> upartloc = aloc * uloc;
 vector_type<double> upart(n);
 MPI_Allgather(
  &upartloc(0), upartloc.vsize(), MPI_DOUBLE,
  &upart(0), vnloc, MPI_DOUBLE,
  vcomm);
 MPI_Reduce_scatter(
  &upart(0), &uloc(0), &hnlocs(0), MPI_DOUBLE,
  MPI_SUM, hcomm);
 uloc += floc;
};

// ... использование вектора результата

// уничтожение созданных коммуникаторов
MPI_Comm_free(&hcomm);
MPI_Comm_free(&vcomm);
```

В приведенном коде средствами MPI задаются размеры двумерной решетки $N^v$ и $N^h$, после чего вычисляются локальные размерности блоков матриц по вертикали и горизон-

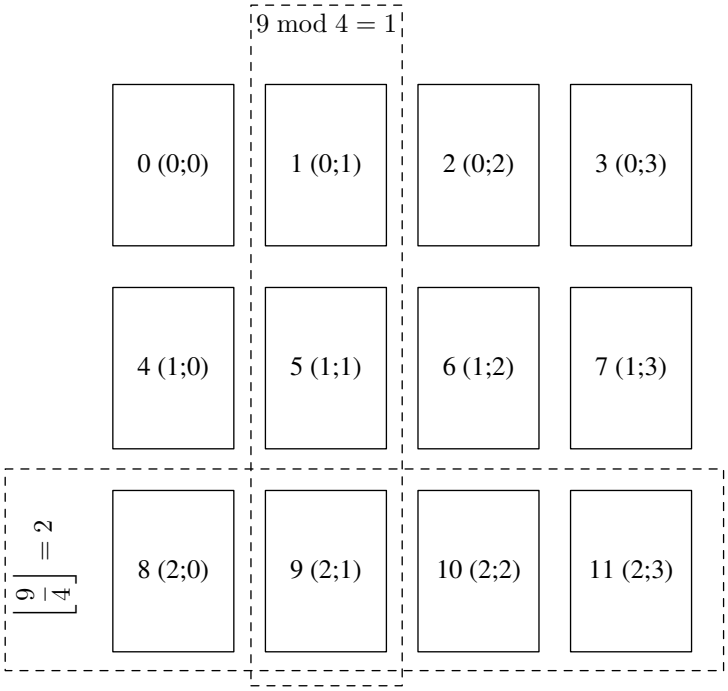



Рис. 1.25. Вычисление координат процесса в двумерной решетке

тали: $m^v = n/N^v$ и $m^h = n/N^h$ соответственно. На основе целочисленного деления номера процесса на длину строки $N^h$ вычисляются координаты каждого процесса в двумерной решетке (рис. 1.25). Вертикальная и горизонтальная координаты получаются в виде целой части результата деления и остатка соответственно.

Далее с помощью вызовов функции `MPI_Comm_split` выполняется разбиение коммуникатора `MPI_COMM_WORLD` на строки и столбцы. В коммуникаторы, объединяющие столбцы процессов, попадают процессы с одинаковыми координатами по горизонтали, и наоборот. Для задания и разбиения двумерной решетки процессов можно (и, возможно, правильнее) было бы использовать функции MPI для работы с декартовой топологией. Однако в данном довольно простом случае это породило бы больше усложнений кода, чем удобств, в связи с чем мы ограничились разбиением «по цвету», в качестве которого выступили вычисленные координаты по горизонтали и вертикали. Созданные коммуникаторы должны быть освобождены после их использования, что и выполняется в конце приведенного фрагмента. Цикл итераций решения СЛАУ соответствует описанию распределенного умножения матрицы на вектор, приведенному выше (рис. 1.24).

При рассмотрении приведенных примеров мы отталкивались от предположения, что размерность матрицы $n$ кратна количеству процессов $N$ (явно описанная проверка в коде опущена здесь для краткости). Разумеется, это не означает, что при некратных размерностях поставленная задача не решается. Подобные случаи лишь требуют модификации кода для использования областей данных неодинаковых размеров, т.е. использования функции `MPI_Allgatherv` вместо `MPI_Allgather` и указания соответствующих размеров областей в массиве, передаваемом функции `MPI_Reduce_scatter`. Размеры областей данных для каждого процесса могут быть вычислены на основе формул приведенного выше сбалансированного блочного распределения.

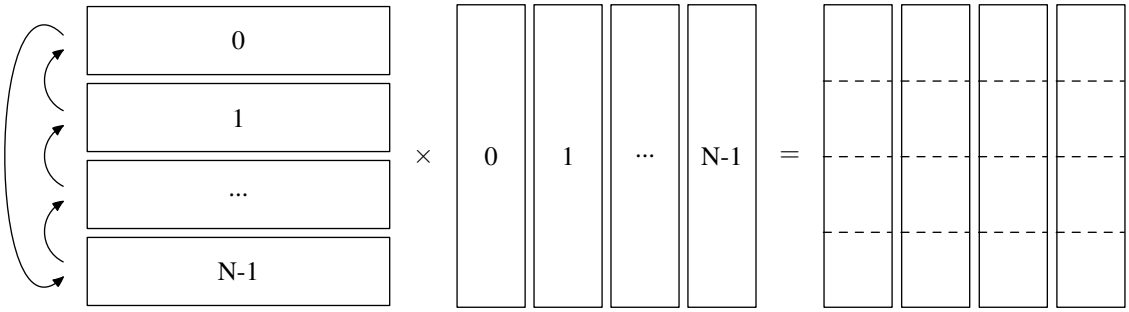



Рис. 1.26. Схема распределенного перемножения двух матриц

### 1.2.6. Перемножение матриц

Еще одной часто возникающей вычислительной задачей, легко поддающейся распараллеливанию, является перемножение матриц. Мы здесь рассмотрим процесс перемножения двух квадратных матриц размерности $n$.

Возможен такой вариант перемножения, когда между машинами распределяется хранение лишь одной матрицы, копия же второй присутствует в памяти каждого процесса целиком. Такая программа, разумеется, будет работать очень эффективно с точки зрения производительности, но потребует большого количества памяти. Этот случай мы здесь не будем рассматривать, поскольку такая программа может быть легко построена на основе программ перемножения матрицы и вектора, приведенных выше. Кроме того, такая программа в принципе представляет мало интереса, поскольку увеличение количества узлов не обеспечит возможности увеличения размеров решаемой задачи.

Будем рассматривать распределенное хранение обеих матриц, поскольку именно такие случаи наиболее интересны при выполнении вычислительных задач в распределенных системах. Однако следует осознавать, что такой подход потребует повышенного обмена информацией между узлами.

Для начала рассмотрим наиболее простой вариант (рис. 1.26). Допустим, требуется умножить слева матрицу $R$ на матрицу $L$ и получить матрицу результата умножения $M$:

$$M = L \cdot R. \tag{1.4}$$

Будем хранить матрицу $L$ горизонтальными блоками, $R$ — вертикальными. Для простоты снова будем считать, что размерность перемножаемых матриц $n$ кратна количеству процессов $N$. Высоту блока левой матрицы и ширину блока правой матрицы ($m = n/N$), как и ранее, будем снова называть локальной размерностью. Каждый процесс содержит один блок $n \times m$ матрицы $R$ и один блок $m \times n$ матрицы $L$. Помимо этого, каждый процесс выделяет область памяти для хранения блока матрицы результата перемножения $M$. Этот блок может быть как вертикальным, так и горизонтальным, в нашем случае он будет вертикальным.

Весь процесс перемножения матриц является пошаговым с количеством шагов, равным количеству процессов $N$. На каждом шаге выполняется перемножение двух локальных прямоугольных блоков, результатом которого является квадратный блок $m \times m$. Элементы этого блока помещаются в локальный прямоугольный блок матрицы $M$, после чего осуществляется циклический сдвиг блоков левой матрицы $L$ между процессами. После сдвига выполняется следующий шаг, на котором прямоугольный блок матрицы результата $M$ пополняется еще одним квадратным блоком. На рис. 1.27 изображен порядок заполне-

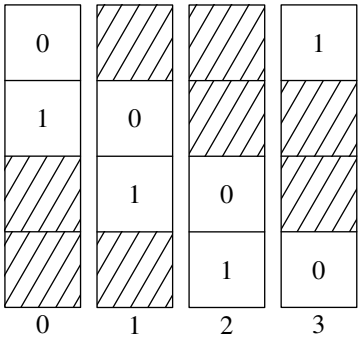



Рис. 1.27. Заполнение блоков матрицы результата после двух шагов вычислений

ния блоков результата после первых двух шагов между четырьмя процессами. Каждый вертикальный блок отражает блок результата, хранимый в памяти соответствующего процесса. Квадратные блоки пронумерованы в соответствии с номером шага, на котором они получены, заштрихованные блоки еще не заполнены.

Описанную процедуру иллюстрирует следующий код:

```
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int nloc = n / size;
// правая матрица (локальная часть)
matrix_type<double> rloc(n, nloc);
// ... инициализация правой матрицы

// левая матрица (локальная часть)
matrix_type<double> lloc(nloc, n);
// ... инициализация левой матрицы

// матрица - результат умножения (локальная часть)
matrix_type<double> mloc(n, nloc);

// вычисление рангов следующего и предыдущего процессов
int next = (rank + 1) % size;
int prev = (rank + size - 1) % size;
// выполнение умножения R на L слева (M = L * R)
for (int s = 0; s < size; ++s)
{
 // умножить вертикальный rloc на горизонтальный lloc
 matrix_type<double> res = lloc * rloc;
 // сохранить квадратный блок результата в mloc
 copy(
  &res(0, 0),
  &res(0, 0) + res.vsize() * res.hsize(),
  &mloc(((rank + s) % size) * nloc, 0));
 // сдвинуть lloc между процессами
 MPI_Status status;
 MPI_Sendrecv_replace(
  &lloc(0, 0), lloc.vsize() * lloc.hsize(), MPI_DOUBLE,
   prev, (rank + s) % size,
```

```
  next, (rank + s + 1) % size,
  MPI_COMM_WORLD, &status);
};
```

После выполнения на каждом шаге перемножения двух прямоугольных блоков осуществляется копирование полученного результата в блок матрицы $M$. Эта операция здесь выполняется путем прямого копирования области памяти с помощью алгоритма `std::copy`. Такое выполнение копирования возможно по той причине, что в соответствии с заявленными нами требованиями к реализации `matrix_type` элементы матрицы хранятся в непрерывном массиве построчно. В противном случае потребовался бы двойной цикл поэлементного копирования с обращением к элементам с помощью `operator()`.

После помещения полученного квадратного блока в блок результата выполняется циклический сдвиг блоков матрицы $L$ между процессами по кольцевой топологии (рис. 1.26). Сдвиг осуществляется однократной отправкой блока из каждого процесса предыдущему и соответствующего приема от следующего в ту же область памяти с помощью функции `MPI_Sendrecv_replace`. Номера следующего и предыдущего процессов вычисляются на основе номера текущего с использованием операции получения остатка от деления. Такой подход для обработки ситуаций выхода за границы закольцованного диапазона является более короткой альтернативой явной обработке условий.

Приведенный код призван проиллюстрировать описанную процедуру наглядно, однако он далеко не во всех отношениях оптимален. В частности, операция копирования здесь, также как и само формирование дополнительной области памяти для хранения квадратного блока, является, по сути, накладными расходами для обеспечения наглядности. В целях экономии памяти можно было, миновав использование определенного для объекта матрицы оператора умножения, осуществить умножение блоков путем явного использования тройного цикла, сохраняя результат сразу в нужные позиции блока матрицы $M$:

```
// ...
// выполнение умножения R на L слева (M = L * R)
for (int s = 0; s < size; ++s)
{
 // умножить вертикальный rloc на горизонтальный lloc,
 // сохраняя результат сразу в mloc
 for (int i = 0; i < lloc.vsize(); ++i)
 {
  for (int j = 0; j < rloc.hsize(); ++j)
  {
   double sum = 0.0;
   for (int k = 0; k < lloc.hsize(); ++k)
    sum += lloc(i, k) * rloc(k, j);
   mloc(((rank + s) % size) * nloc + i, j) = sum;
  };
 };

 // сдвинуть lloc между процессами
 MPI_Status status;
 MPI_Sendrecv_replace(
  &lloc(0, 0), lloc.vsize() * lloc.hsize(), MPI_DOUBLE,
  prev, (rank + s) % size,
  next, (rank + s + 1) % size,
  MPI_COMM_WORLD, &status);
};
```

Если программа работает на пределе использования памяти, во время циклического

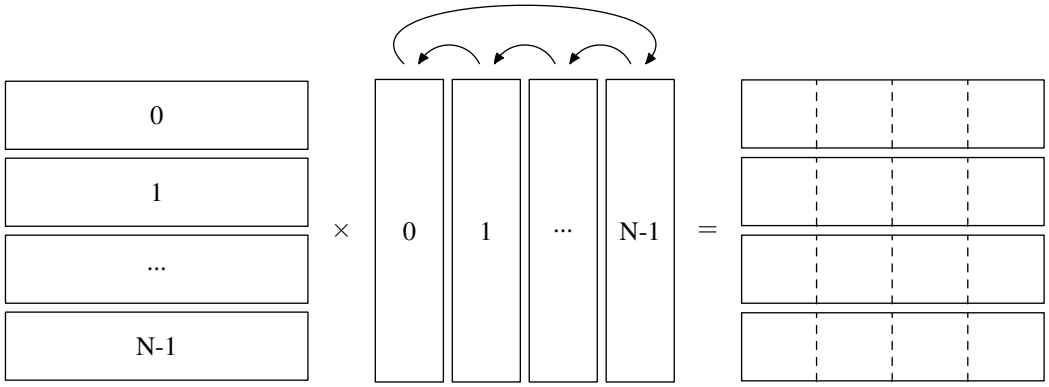



Рис. 1.28. Альтернативная схема для случая хранения результата горизонтальными блоками

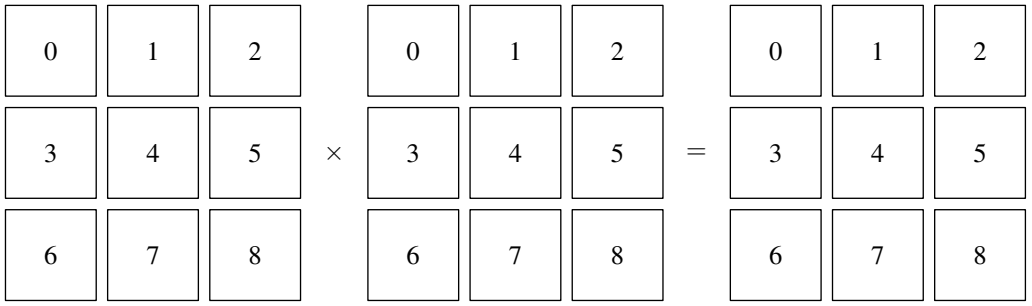



Рис. 1.29. Распределение блоков перемножаемых матриц и матрицы результата по двумерной решетке процессов

сдвига можно вместо отправки целого блока осуществлять многократную постепенную отправку более мелкими блоками, к примеру, построчно. Разумеется, это скажется на снижении производительности из-за латентности сети. Однако такой подход потребовался бы в случае явного использования функций `MPI_Send` и `MPI_Irecv`, в нашем же случае об этих аспектах должна заботиться внутренняя реализация функции `MPI_Sendrecv_replace`.

В результате выполнения полного цикла умножения каждый из $N$ процессов будет содержать один из $N$ вертикальных блоков матрицы результата. В процессе выполнения цикла производится $N$ пересылок областей памяти.

Как уже упоминалось выше, возможен и другой вариант хранения матрицы результата — горизонтальными блоками (рис. 1.28). Основное отличие этого варианта в том, что циклически между процессами сдвигаются блоки не левой, а правой матрицы. Мы не будем приводить реализацию этого подхода, поскольку он практически идентичен предыдущему.

Наконец, рассмотрим еще один вариант умножения. Он немногим более сложен для реализации, однако позволяет сократить количество пересылок данных. На этот раз будем предполагать, что количество процессов $N$ является квадратом некоторого целого числа, а размерность перемножаемых матриц $n$ кратна этому числу. Разобьем обе перемножаемые матрицы двумерной сеткой на квадратные блоки. На рис. 1.29 приведен пример такого распределения по процессам блоков матриц $L$, $R$ и $M$.

На этот раз локальной размерностью будем называть ширину и высоту локального квадратного блока, т.е. величину $m = n/\sqrt{N}$. Каждый из $N$ процессов хранит по одному квадратному блоку $m \times m$ из обеих перемножаемых матриц и один блок матрицы резуль-

тата. При перемножении производится $N' = \sqrt{N}$ шагов, на каждом из которых процесс выполняет умножение очередных двух блоков из матриц $L$ и $R$. Полученные процессом результаты всех шагов суммируются, в результате чего после выполнения полного цикла в каждом процессе с номером $rank = 0, \ldots, N-1$ хранится квадратный блок $m \times m$ матрицы результата $M_{ij}$. Координаты $i$ и $j$ в двумерной решетке процессов вычисляются как целая часть от деления и остаток, приращенные на единицу:

$$\begin{gathered} M_{ij} = \sum_{k=1}^{N'} L_{ik} \cdot R_{kj}, \qquad i,j = 1, \ldots, N'; \\ i = \left\lfloor \frac{rank}{N'} \right\rfloor + 1, \qquad j = rank - \left\lfloor \frac{rank}{N'} \right\rfloor N' + 1. \end{gathered} \tag{1.5}$$

Изначально процесс хранит в своей памяти локальные блоки обеих матриц $L_{ij}$ и $R_{ij}$ в соответствии со своими координатами. Разумеется, на каждом шаге оба перемножаемых процессом блока должны меняться. Для смены одного из блоков, к примеру $R_{ij}$, мы можем, как и в предыдущем случае, использовать циклический сдвиг (в вертикальном направлении). Для смены же блока $L_{ij}$ мы используем рассылку блока от одного из процессов в рамках каждой строки двумерной решетки. Поясним это подробнее.

Распишем суммы $M_{ij}$ из (1.5), представляющие результат полного умножения в каждом процессе:

$$\begin{gathered} M_{11} = L_{11}R_{11} + L_{12}R_{21} + \cdots + L_{1N'}R_{N'1}; \\ M_{12} = L_{11}R_{12} + L_{12}R_{22} + \cdots + L_{1N'}R_{N'2}; \\ \cdots \\ M_{1N'} = L_{11}R_{1N'} + L_{12}R_{2N'} + \cdots + L_{1N'}R_{N'N'}; \\ M_{21} = L_{22}R_{21} + \cdots + L_{2N'}R_{N'1} + L_{21}R_{11}; \\ \cdots \\ M_{2N'} = L_{22}R_{2N'} + \cdots + L_{2N'}R_{N'N'} + L_{21}R_{1N'}; \\ \cdots \\ M_{N'1} = L_{N'N'}R_{N'1} + L_{N'1}R_{11} + \cdots + L_{N'N'-1}R_{N'-11}; \\ \cdots \\ M_{N'N'} = L_{N'N'}R_{N'N'} + L_{N'1}R_{1N'} + \cdots + L_{N'N'-1}R_{N'-1N'}. \end{gathered}$$

В приведенных выражениях изменен порядок следования слагаемых с целью учета начального сдвига по процессам локальных блоков правой матрицы $R_{ij}$. Члены суммы выписаны в том порядке, в каком будет выполняться их вычисление при формировании $M_{ij}$, исходя из того, какой блок правой матрицы доступен процессу на соответствующем шаге. Смена блоков $R_{kj}$ при вычислении каждой суммы происходит посредством циклического сдвига по столбцам решетки процессов в вертикальном направлении (рис. 1.30).

Из приведенных выражений видно, что для всех $M_{ij}$ с одинаковым значением $i$, т.е. в пределах каждой строки решетки процессов, на каждом шаге блоки левой матрицы $L_{ik}$ совпадают между собой. При этом номер требуемого блока $k$ в текущей строке на каждом шаге $s = 0, \ldots, N' - 1$ равен:

$$k = \begin{cases} i + s, & i + s \leqslant N'; \\ i + s - N', & i + s > N'. \end{cases} \tag{1.6}$$

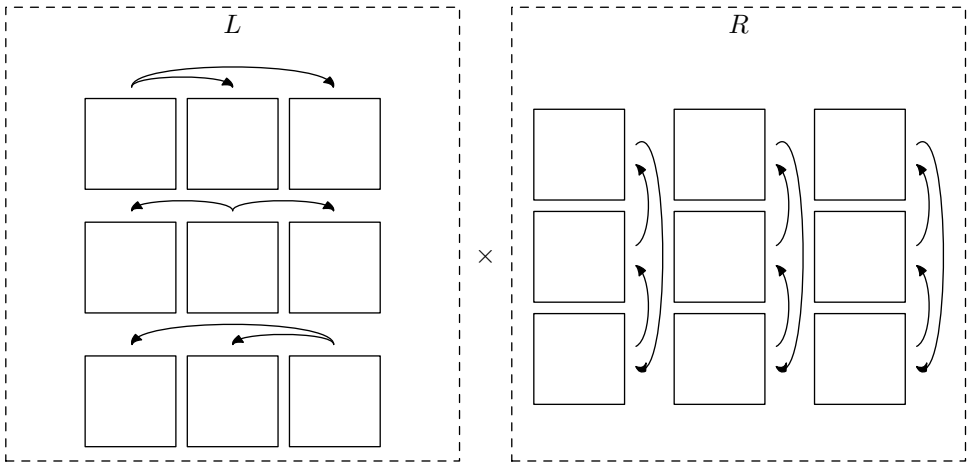



Рис. 1.30. Пересылки данных, производимые в рамках одного шага вычислений

Блок левой матрицы с вычисленным номером рассылается всем процессам, находящимся в одной строке двумерной решетки, непосредственно перед выполнением перемножения блоков $L_{ik}$ и $R_{kj}$ (рис. 1.30). После перемножения осуществляется сдвиг блоков правой матрицы и переход к следующему шагу.

Описанный процесс иллюстрируется следующим кодом:

```
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int sqrtsize = int(floor(sqrt(1.0 * size) + 0.5));
assert(sqrtsize * sqrtsize == size && n % sqrtsize == 0);

int nloc = n / sqrtsize;
const int ndims = 2;

MPI_Comm cart, subcart;
// создаем коммуникатор с двумерной декартовой топологией
int dim[ndims] = {sqrtsize, sqrtsize};
int period[ndims] = {true, true};
MPI_Cart_create(MPI_COMM_WORLD, ndims, dim, period, false, &cart);
// получаем двумерные координаты нашего процесса
int coords[ndims], &icoord = coords[0], &jcoord = coords[1];
MPI_Cart_coords(cart, rank, ndims, coords);
// разбиваем декартов коммуникатор по строкам
int split[ndims] = {false, true};
MPI_Cart_sub(cart, split, &subcart);
// получим группы (они нужны для трансляции рангов)
MPI_Group gcart, gsubcart;
MPI_Comm_group(cart, &gcart);
MPI_Comm_group(subcart, &gsubcart);

// правая матрица (локальная часть)
matrix_type<double> rloc(nloc, nloc);
// ... инициализация правой матрицы
```

```
// левая матрица (локальная часть)
matrix_type<double> lloc(nloc, nloc);
// ... инициализация левой матрицы

// матрица - результат умножения (локальная часть)
// конструктор инициализирует нулевыми значениями
matrix_type<double> mloc(nloc, nloc);

// выполнение умножения R на L слева (M = L * R)
for (int s = 0; s < sqrtsize; ++s)
{
 // вычислим ранг процесса-источника для рассылки
 // блока левой матрицы среди строки процессов
 int src, dst, subroot;
 MPI_Cart_shift(cart, 1, jcoord - (icoord + s), &src, &dst);
 MPI_Group_translate_ranks(gcart, 1, &src, gsubcart, &subroot);

 // разошлем блок левой матрицы всем процессам строки
 matrix_type<double> lcopy(lloc.vsize(), lloc.hsize());
 if (rank == src)
  lcopy = lloc;
 MPI_Bcast(
  &lcopy(0, 0),
  lloc.vsize() * lloc.hsize(), MPI_DOUBLE,
  subroot,
  subcart);

 // перемножим локальные блоки левой и правой матрицы
 matrix_type<double> res = lcopy * rloc;
 // сохраним квадратный блок результата
 mloc += res;

 // вычислим следующий и предыдущий ранги
 // для сдвига по столбцу процессов
 MPI_Cart_shift(cart, 0, -1, &src, &dst);
 // сдвинем rloc между процессами в столбце
 MPI_Status status;
 MPI_Sendrecv_replace(
  &rloc(0, 0),
  rloc.vsize() * rloc.hsize(), MPI_DOUBLE,
  dst, 0,
  src, 0,
  cart, &status);
};

MPI_Group_free(&gsubcart);
MPI_Group_free(&gcart);
MPI_Comm_free(&subcart);
MPI_Comm_free(&cart);
```

Для удобства оперирования номерами процессов первым делом создается коммуникатор `cart` с декартовой топологией, после чего определяются координаты в нем текущего процесса. Созданная топология является двумерным тором, т.е. обе координаты обладают периодичностью. Следующим этапом с помощью функции `MPI_Cart_sub` вся решетка раз-

бивается по строкам. Полученный коммуникатор `subcart` используется позже для широковещательной рассылки блока левой матрицы. Наконец, в цикле выполняется $\sqrt{N}$ шагов операции умножения.

На каждом шаге первым этапом с помощью функции `MPI_Cart_shift` выполняется определение ранга процесса, от которого будет производиться рассылка блока левой матрицы в контексте текущего коммуникатора `subcart`. Второй параметр этой функции сообщает номер измерения, в котором производится смещение (0 — по вертикали, 1 — по горизонтали). Смещение производится одновременно в обоих направлениях относительно текущего процесса на величину, задаваемую третьим параметром — величиной сдвига, при этом в данном случае используется только одно из двух полученных значений. Величина сдвига определяется по формуле (1.6) на основе номера шага и номера строки процессов `icoord`, при этом, поскольку функция `MPI_Cart_shift` возвращает значение ранга на основе заданного сдвига относительно текущего процесса, используется разница между координатой `jcoord` и значением (1.6).

Поскольку мы получаем ранг процесса в группе коммуникатора `cart`, нам требуется его трансляция в значение ранга того же процесса в группе коммуникатора `subcart`, для чего используется функция `MPI_Group_translate_ranks`. Эта функция транслирует массив рангов, в нашем случае его размер равен единице. После определения номера процесса-источника (он во всех процессах группы текущего коммуникатора `subcart` будет совпадать) производится широковещательная рассылка хранимого им блока левой матрицы во все остальные процессы группы коммуникатора `subcart`. Для этого во всех процессах используется временный объект `lcopy`, в который процесс-источник перед осуществлением рассылки копирует хранимый им блок $L_{ij}$.

Распространенный функцией `MPI_Bcast` блок левой матрицы перемножается с текущим блоком правой матрицы, после чего текущее содержимое блока $M_{ij}$ инициализируется результатом (на первом шаге) или пополняется им (на остальных).

Наконец, в конце выполнения каждого этапа умножения циклически сдвигается блок правой матрицы в вертикальном направлении с использованием функции `MPI_Sendrecv_replace`. Для определения номеров процесса-источника и процесса-приемника при сдвиге снова используется функция `MPI_Cart_shift`. Созданная нами декартова топология избавляет нас от необходимости «вручную» вычислять номера процессов источника и приемника и обрабатывать выход за пределы диапазона, что выполнялось нами в предыдущих примерах. По завершении цикла производится очистка созданных коммуникаторов и групп.

На каждом шаге производится по две пересылки данных (функциями `MPI_Bcast` и `MPI_Sendrecv_replace`), поэтому во время такой операции перемножения матриц будет выполнено всего $2\sqrt{N}$ пересылок блоков памяти. Для количества процессов $N > 4$ такой способ, несомненно, оказывается гораздо более эффективным по сравнению с предыдущим, поскольку позволяет существенно сократить накладные расходы на коммуникации.

По завершении умножения блоки распределенной по процессам матрицы результата $M_{ij}$ могут быть использованы в соответствии с их предназначением. К примеру, может быть выполнено перемножение с еще одной матрицей или умножение полученной матрицы на вектор. Пример умножения разбитой таким образом матрицы на вектор был рассмотрен выше.

# Глава 2.

# Ярусно-параллельная форма программы

В этой главе мы расскажем об одном из самых естественных и просто реализуемых подходов к распараллеливанию некоторой комплексной задачи. Именно таким образом во многих средах программирования выстраивается последовательность запуска подзадач при параллельной сборке большого количества проектов с зависимостями между ними. Похожий подход используется и при решении задач, описанных в терминах модели потоков данных (dataflow), а также в языках программирования, использующих понятие будущих значений (futures).

Такое обилие областей применения объясняется тем простым фактом, что представление в ярусно-параллельной форме лежит в основе параллельного выполнения практически любой задачи, обладающей логическим параллелизмом. Напомним, что логическим параллелизмом обладает такая задача, в которой присутствуют некоторые подмножества информационно независимых друг от друга подзадач, процесс выполнения каждой из которых не связан прямо или косвенно с остальными подзадачами того же подмножества. Подзадачи такого подмножества в принципе могут быть выполнены параллельно. При этом каждая подзадача, возможно, по своим входным данным находится в зависимости от результатов выполнения каких-либо других подзадач, не входящих в указанное подмножество.

Порядок решения подобного рода комплексных задач тесно связан с методами сетевого планирования и управления [7, 19]. В соответствии с принятой в этой области терминологией, будем называть в дальнейшем подзадачи — работами, а всю распараллеливаемую задачу — комплексом работ.

## 2.1. Цель и механизм построения

Графическое представление схемы зависимостей работ друг от друга называется сетевым графиком работ. Сетевой график работ представляется направленным ациклическим графом, т.е. ни одна работа не должна прямо либо косвенно зависеть от самой себя. К примеру, на рис. 2.1 можно увидеть сетевой график некоторого комплекса, состоящего из восьми работ. Прямоугольниками здесь обозначены работы, стрелками — зависимости между ними (входными данными одной работы являются выходные данные другой).

Наличие циклической зависимости (петли или замкнутого контура) в сетевом графике работ говорит о том, что некоторая работа, прежде чем выполниться, должна так или иначе дождаться результатов своего же выполнения. Разумеется, подобное описание комплекса работ лишено смысла. Попытка выполнения такого комплекса приводит к возникновению

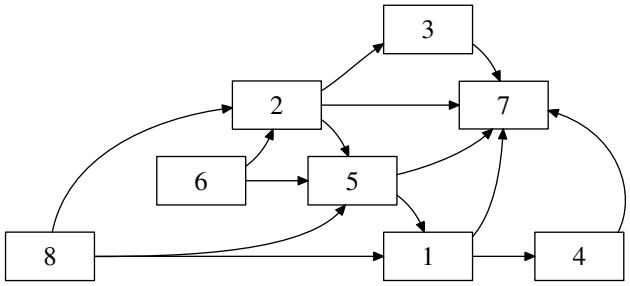



Рис. 2.1. Сетевой график некоторого комплекса работ

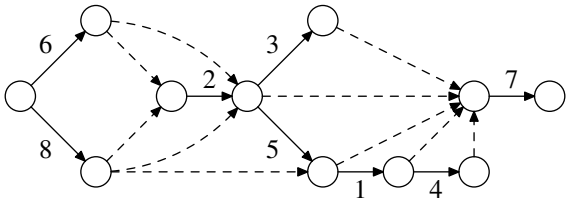



Рис. 2.2. Альтернативное представление сетевого графика работ

взаимоблокировки (deadlock), в связи с чем следует следить, чтобы соответствующий граф был ациклическим.

В сетевом планировании используются два способа представления сетевого графика работ. В первом случае, как на рис. 2.1, работы представляются вершинами графа, зависимости — дугами. Во втором случае работы представляются дугами, вершинами же представляются события их завершения. Общие правила построения таких сетевых графиков приведены в [19].

Пример представления вторым способом комплекса работ, изображенного на рис. 2.1, приведен на рис. 2.2. Здесь пунктирными стрелками обозначены так называемые фиктивные работы, не занимающие ресурсов и характеризующие лишь логическую связь. Разумеется, на этом графике можно опустить фиктивные работы, характеризующие связи, которые неявно вытекают из других, в результате чего получаем заключительное представление сетевого графика работ (рис. 2.3).

Оба способа представления имеют свои достоинства в тех или иных условиях. Второй способ удобен для анализа временных характеристик работ и не обременен лишними связями, поэтому широко используется в сетевом планировании. Первый способ не нуждается в использовании фиктивных работ и обеспечивает более высокую гибкость при добавлении неучтенных зависимостей. Поскольку мы рассматриваем случай, когда анализ временных характеристик не актуален, мы будем пользоваться первым способом представления сетевого графика.

В качестве альтернативы графическому представлению комплекс работ может быть представлен структурной таблицей комплекса работ, в которой указываются работы, включенные в комплекс, а также зависимости каждой работы от других (табл. 2.1).

Решение задач сетевого планирования и управления сводится к определению оптимального распределения ресурсов с целью получения минимального времени выполнения всего комплекса (или же минимального количества ресурсов при фиксированном времени выполнения). Подобная оптимизация может быть осуществлена на основе известных зависимостей времени выполнения отдельных работ от количества выделенных им ресурсов.

При выполнении параллельных вычислений в такой постановке рассматривать задачу

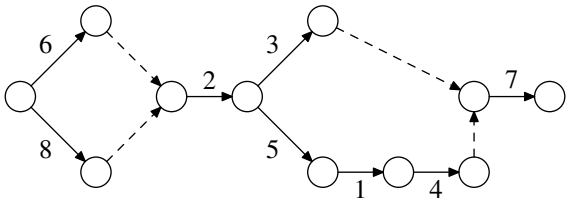



Рис. 2.3. Сетевой график без избыточных фиктивных работ

Таблица 2.1. Структурная таблица комплекса работ

| Работа | Зависимость |
|---|---|
| 1 | 5, 8 |
| 2 | 6, 8 |
| 3 | 2 |
| 4 | 1 |
| 5 | 2, 6, 8 |
| 6 | — |
| 7 | 1, 2, 3, 4, 5 |
| 8 | — |

оказывается затруднительно и зачастую бессмысленно, поскольку в однородной системе нечем управлять (мы не можем повлиять на скорость выполнения конкретной подзадачи), а в неоднородной оказывается сложно задать зависимость времени выполнения каждой подзадачи.

Нам интересен лишь первый этап решения задачи поиска критического пути — упорядочение заданного сетевого графика работ и построения ярусно-параллельной формы программы [1, 7, 19].

Процедура построения ярусно-параллельной формы (далее — ЯПФ) достаточно проста и заключается в представлении сетевого графика в форме последовательных ярусов работ, в рамках каждого из которых работы не связаны между собой и потому могут выполняться независимо друг от друга. Построение ЯПФ легко осуществляется на основе структурной таблицы комплекса работ и сводится к добавлению столбца с присвоенными номерами ярусов. Работами нулевого яруса считаются все работы, выполнение которых не зависит от результатов выполнения каких-либо других работ. Работы первого яруса — те, которые опираются только на работы нулевого яруса. Работы каждого яруса из последующих должны опираться только на работы более ранних по отношению к ним ярусов, среди которых должна быть как минимум одна работа непосредственно предшествующего яруса. В табл. 2.2 можно увидеть распределение по ярусам работ комплекса, изображенного на рис. 2.1.

После выполнения распределения по ярусам всех работ комплекса сетевой график может быть изображен с учетом разбиения по ярусам (рис. 2.4).

Такая форма организации программы носит название ЯПФ программы. Преимущество приведения программы к ЯПФ заключается в возможности параллельного выполнения в рамках одного яруса всех включенных в него работ. Количество работ в каждом ярусе называется шириной яруса, полное количество ярусов — высотой ЯПФ, максимальная ширина яруса — шириной ЯПФ программы.

Эти характеристики при некоторых допущениях позволяют нам судить о свойствах по-

Таблица 2.2. Распределение работ по ярусам

| Работа | Зависимость | Ярус |
|---|---|---|
| 1 | 5, 8 | 3 |
| 2 | 6, 8 | 1 |
| 3 | 2 | 2 |
| 4 | 1 | 4 |
| 5 | 2, 6, 8 | 2 |
| 6 | — | 0 |
| 7 | 1, 2, 3, 4, 5 | 5 |
| 8 | — | 0 |

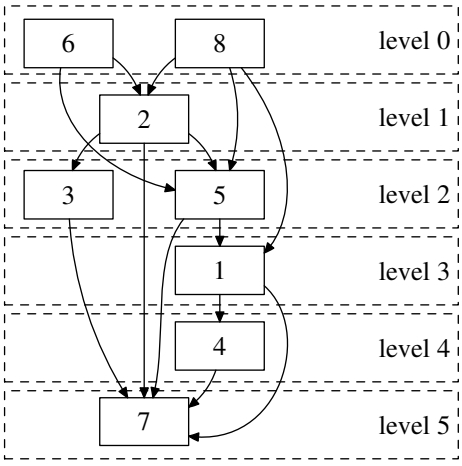



Рис. 2.4. Ярусно-параллельное представление комплекса работ

строенной таким образом программы. В частности, при условии, что все работы приблизительно одинаковы по длительности, ширина ЯПФ характеризует максимальное количество требуемых параллельных ресурсов во время выполнения. В случае наличия необходимого количества параллельных ресурсов время выполнения комплексной работы будет приблизительно равно произведению времени выполнения одной работы на высоту ЯПФ.

Порой, встречаются утверждения, что распараллеливание программ с использованием ЯПФ плохо согласовано с конструкциями существующих языков программирования, чем обусловлена сложность его выполнения и соответствующее крайне редкое использование [1]. Действительно, большинство популярных сегодня языков программирования изначально предназначены для последовательных вычислений и потому плохо приспособлены для любого распараллеливания. Однако следует учитывать, что написание сложной программы (а параллельная программа, как правило, является сложной) всегда требует усилий, и поиск простых путей в большинстве случаев отрицательно сказывается на результате. В связи с этим мы предложим далеко не новый подход, заключающийся в отделении специфики конкретной задачи от функциональности используемой схемы распараллеливания, а именно построения ЯПФ и осуществления как такового параллельного выполнения. В этом случае написание кода, отвечающего за выполнение комплекса работ, разумеется, будет не легче, однако может быть выполнено лишь единожды, в то время как использован такой код может быть многократно при решении широкого круга задач.

## 2.2. Варианты реализации механизма

Прежде чем переходить к описанию предлагаемых вариантов реализации следует оговорить следующий момент. Зачастую встречаются утверждения, что, вследствие того, что любая работа текущего яруса находится в зависимости от хотя бы одной работы предыдущего яруса, она не может быть начата раньше, чем завершится выполнение всех работ предыдущего яруса [1].

Стоит отметить, что это не совсем верно, однако рассмотрение с такой точки зрения имеет и свои плюсы. При подобной реализации нам в любой момент известно необходимое количество параллельных ресурсов, требуемых для параллельного выполнения работ, поскольку оно равно ширине выполняющегося в текущий момент яруса.

Следует также отметить, что такой подход может быть достаточно эффективным и простым для реализации при условии, что в пределах каждого яруса все работы приблизительно одинаковы по длительности. В противном же случае, когда длительности существенно различаются, а зависимостей между работами не очень много, могут возникнуть существенные потери времени на ожидание некоторой работой момента завершения какой-либо работы предыдущего яруса, от которой она не зависит.

Поскольку в реальности встречаются разные ситуации, мы рассмотрим оба варианта. Вначале рассмотрим реализацию более простого случая, когда все работы выполняются строго поярусно, т.е. ни одна работа следующего яруса не начинается до тех пор, пока не закончат выполнение все работы текущего яруса.

### 2.2.1. Поярусное выполнение комплекса работ

Для простоты и наглядности отбросим необходимость передачи входных и выходных данных между работами и сконцентрируемся на распределении их выполнения между параллельными ресурсами. Будем считать, что данные передаются и принимаются через разделяемые ресурсы внутри работ, при этом готовность этих данных к использованию регулируется зависимостями работ.

#### Последовательная реализация с директивами OpenMP

В приложении Б приведен код предлагаемых классов, использующий распараллеливание средствами интерфейса OpenMP. Среди них, прежде всего, абстрактный класс работы:

```
// абстрактный интерфейс работы
class job_abstract_type
{
public:
 // выполнение работы
 // получение исходных данных и вывод результата
 // выполняются внутри через разделяемые ресурсы
 virtual
 void run(void) = 0;
};
```

Все работы в клиентском коде наследуются от класса `job_abstract_type` и переопределяют функцию `run`, внутри которой осуществляется как таковое выполнение работы.

Также определен класс комплекса работ `jobcomplex_type`. Поскольку комплекс работ, в сущности, также является работой, класс `jobcomplex_type` унаследован от класса `job_abstract_type` и, в принципе, может быть использован в качестве работы при формировании комплекса работ более высокого уровня сложности.

В классе `jobcomplex_type` объявляются необходимые структуры данных для заполнения комплекса работами и зависимостями. Указатели на внесенные работы хранятся вместе с отображением на их номера. Номера работам присваиваются в соответствии с порядком их добавления. Зависимости хранятся в виде контейнера `std::set` из пар номеров, что позволяет избежать повторного внесения зависимости. Для добавления работ и зависимостей клиентскому коду предоставляются функции `add_job` и `add_dependence` соответственно.

Реализация функции выполнения комплекса работ `run` начинается с формирования списка работ и таблицы зависимостей на основе заполненных ранее структур данных с помощью функций `get_joblist` и `get_deplist`. После этого вызывается функция `build`, которая осуществляет построение ЯПФ на основе таблицы зависимостей работ. После построения ЯПФ в цикле по номерам ярусов выполняются все работы. Работы на каждом ярусе выполняются параллельно, для чего используется директива OpenMP, при этом до момента завершения всех работ текущего яруса следующий не начинается.

Функция `build`, осуществляющая построение ЯПФ, начинается с определения ярусов всех работ. С этой целью вводится множество работ, для которых ярусы еще не определены, изначально содержащее номера всех работ. Далее до момента опустошения этого множества выполняется цикл, в теле которого на основе таблицы зависимостей производится попытка определить ярусы каких-либо работ из оставшихся. Номера работ, ярус которых был определен, исключаются из множества неопределенных, после чего осуществляется переход к определению работ следующего яруса. Исключение номеров выполняется с помощью алгоритма `std::set_difference`, возможность использования которого обеспечивается тем фактом, что оба исходных множества номеров являются упорядоченными по возрастанию.

При попытке определить номер яруса работы осуществляется проход по всем ее зависимостям. Если какая-либо работа, от которой зависит текущая, все еще в списке неопределенных, то и текущая работа остается неопределенной. В противном случае среди всех зависимостей определяется максимальный номер яруса, после чего полученное значение с увеличением на единицу присваивается в качестве номера яруса текущей работе.

После определения номеров ярусов всех работ на основе них строится как таковая структура ЯПФ. Программно она представляется в виде контейнера контейнеров, каждый контейнер в котором характеризует ярус работ, а его элементы — номера работ соответствующего яруса.

Использование приведенных классов оказывается довольно простым и наглядным. К примеру, фрагмент программы, реализующий выполнение комплекса работ, приведенного на рис. 2.1, может выглядеть следующим образом:

```
// создание объектов работ
// j1, j2, j3, j4, j5, j6, j7, j8
// ...

// создание и наполнение комплекса работ
jobcomplex_type jobcomplex;
// добавление списка работ
jobcomplex.add_job(j1);
jobcomplex.add_job(j2);
jobcomplex.add_job(j3);
jobcomplex.add_job(j4);
jobcomplex.add_job(j5);
jobcomplex.add_job(j6);
jobcomplex.add_job(j7);
jobcomplex.add_job(j8);
// добавление списков зависимостей
jobcomplex.add_dependence(j1, j5);
```

```
jobcomplex.add_dependence(j1, j8);
jobcomplex.add_dependence(j2, j6);
jobcomplex.add_dependence(j2, j8);
jobcomplex.add_dependence(j3, j2);
jobcomplex.add_dependence(j4, j1);
jobcomplex.add_dependence(j5, j2);
jobcomplex.add_dependence(j5, j6);
jobcomplex.add_dependence(j5, j8);
jobcomplex.add_dependence(j7, j1);
jobcomplex.add_dependence(j7, j2);
jobcomplex.add_dependence(j7, j3);
jobcomplex.add_dependence(j7, j4);
jobcomplex.add_dependence(j7, j5);

// выполнение комплекса работ
jobcomplex.run();
```

Каждая работа представляется объектом некоторого класса, унаследованного от `job_abstract_type`. После их создания производится создание и заполнение объекта комплекса работ, т.е. добавление работ и зависимостей между ними в соответствии со структурной таблицей комплекса. Наконец, производится выполнение комплекса работ.

### Реализация, распараллеленная с помощью MPI

Опишем кратко вариант модификации кода, приведенного в приложении Б, для распределенного выполнения с использованием интерфейса MPI. В сущности, изменениям подвергается лишь функция `jobcomplex_type::run`:

```
class jobcomplex_type: public job_abstract_type
{
 // ...

 // выполнение всего комплекса работ
 void run(void)
 {
  // получим список подлежащих выполнению работ
  joblist_type joblist = get_joblist();
  // построим ярусно-параллельную структуру номеров работ
  multilevel_type multilevel = build(get_deplist());

  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  multilevel_type::iterator it;
  // выполним по очереди каждый ярус работ
  for (it = multilevel.begin(); it != multilevel.end(); ++it)
  {
   // жестко отделим выполнение ярусов друг от друга
   MPI_Barrier(MPI_COMM_WORLD);
   // распределим каждый ярус работ циклически
   int width = it->size();
   for (int i = rank; i < width; i += size)
    joblist[(*it)[i]]->run();
  };
```

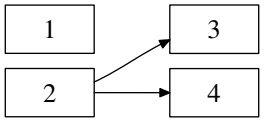



Рис. 2.5. Сетевой график простого комплекса работ

```
  // в конце дождемся завершения всех работ
  MPI_Barrier(MPI_COMM_WORLD);
 }
};
```

Для распределенного выполнения работ внесена явная барьерная синхронизация между ярусами, и циклически распределено выполнение работ каждого яруса по процессам коммуникатора `MPI_COMM_WORLD`. Также добавлена барьерная синхронизация в конце функции, поскольку дальнейшее выполнение клиентского кода обычно полагается на факт завершения всех работ.

Для использования модифицированного таким образом набора классов приведенный ранее фрагмент клиентского кода должен быть дополнен лишь вызовами `MPI_Init` и `MPI_Finalize`.

### 2.2.2. Учет индивидуальных зависимостей работ

Теперь рассмотрим случай, когда длительности работ существенно отличаются друг от друга. Очевидно, что, если в такой ситуации в конце каждого яруса выполнять барьерную синхронизацию между всеми параллельными ресурсами, обычным явлением будет простой тех ресурсов, которые закончили выполнение своей работы раньше. В случае если при этом некоторая работа следующего яруса зависит лишь от тех работ текущего, которые уже завершились, она может быть уже начата и, возможно, даже завершена, однако вместо этого она вынуждена ожидать завершения всех остальных выполняющихся работ текущего яруса. Вследствие этого могут возникать существенные задержки общего времени выполнения комплекса.

В этой связи снимем условие необходимости для запуска работы завершения всех работ предыдущего яруса и вернемся к изначальному условию необходимости завершения лишь тех работ, от которых данная работа непосредственно зависит. В этой ситуации могут сократиться задержки по времени, однако это не всегда будет «бесплатно». Зачастую в таких условиях требуемое количество параллельных ресурсов может превышать ширину ЯПФ.

Поясним это на простом примере. На рис. 2.5 приведен пример комплекса работ, ширина ЯПФ которого равна двум. В случае если вторая работа окажется гораздо короче первой, для наискорейшего выполнения комплекса работ потребуются три параллельных ресурса — для одновременного выполнения первой, третьей и четвертой работ. На рис. 2.6 показано, как такой комплекс может выполняться во времени строго по ярусам с барьером между ними (слева) и при уплотнении с учетом индивидуальных зависимостей (справа).

Таким образом, при индивидуальном учете зависимостей количество требуемых параллельных ресурсов в общем случае может существенно превышать ширину ЯПФ. В худшем случае количество одновременно выполняющихся работ может достигать числа $n - h + 1$, где $n$ — общее количество работ, $h$ — высота ЯПФ. Пример такой ситуации приведен на рис. 2.7. Предполагается, что заштрихованные работы, лежащие в основе всех зависимостей комплекса, оказались кратковременными и уже завершены.

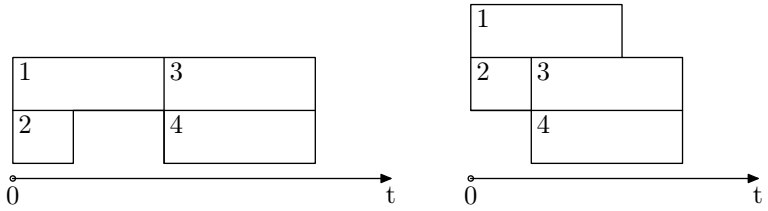



Рис. 2.6. Варианты параллельного выполнения работ во времени

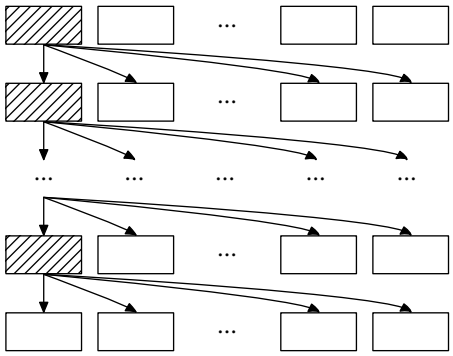


Рис. 2.7. Пример комплекса с большим количеством одновременно выполняющихся работ

Стоит отметить, что при такой последовательности выполнения становится практически бессмысленно говорить о ЯПФ программы, поскольку информация о составе ярусов в данном случае почти не используется. Подобный механизм напоминает, скорее, порядок выполнения в рамках модели потоков данных, где запуск подзадачи осуществляется в момент поступления для нее исходных данных. О такой реализации можно говорить как об альтернативе ярусно-параллельному выполнению программы, в некоторых ситуациях обладающей гораздо более высокой эффективностью и в нашем случае не требующей изменения клиентского кода.

В связи с тем, что необходимое количество параллельных ресурсов заранее неизвестно, реализация такого подхода с использованием MPI оказывается затруднительной вследствие статического существования группы процессов. Помимо этого, для подобной реализации необходимы довольно тонкие механизмы синхронизации, вследствие чего использование интерфейса OpenMP также оказывается неестественным. В этом случае может подойти какой-либо более низкоуровневый программный интерфейс, к примеру, для многопоточной реализации могут быть использованы интерфейсы Windows API или pthreads. Мы приведем вариант многопоточной реализации с использованием интерфейса Windows API.

Прежде всего, оговорим функциональность предлагаемого кода. Каждая работа выполняется в своем потоке. При этом вместо ожидания завершения яруса работ перед началом следующего внесем перед выполнением каждой работы ожидание завершения тех работ, от которых она зависит. Такие ожидания вынесем в начало каждого созданного для выполнения работы потока и поместим непосредственно перед выполнением работы.

Поскольку в таких условиях к моменту завершения всех работ, от которых текущая работа зависит, поток может уже существовать и находиться в ожидании, представляется удобным создание сразу всех потоков для всех работ с тем, чтобы они находились в состоянии ожидания до тех пор, пока не станет допустимо выполнение работы. Однако следует

учитывать, что одновременное создание большого количества потоков может чрезмерно нагрузить операционную систему. При этом такая нагрузка может оказаться бесполезной, если, к примеру, задачи окажутся по длительности близкими друг к другу, и такое количество одновременно существующих потоков не будет необходимым вследствие того, что большинство из них будет пребывать в ожидании.

В связи с этим примем во внимание тот факт, что ни одна работа следующего яруса не может быть начата прежде, чем завершится хотя бы одна работа текущего. В соответствии с таким положением будем создавать потоки поэтапно по одному ярусу, при этом переход к созданию потоков следующего яруса осуществляется только в момент завершения какой-либо работы текущего.

Как и ранее, будем использовать за основу код классов, приведенных в приложении Б. Внешний интерфейс классов снова не изменяется, следовательно, клиентский код использующей их программы может оставаться прежним. Более того, как и в предыдущем случае, практически не меняется большинство реализующего классы кода. Помимо изменений функции `jobcomplex_type::run`, которые будут описаны ниже, в код класса `jobcomplex_type` добавились несколько объявлений типов, функция потока и две функции ожидания:

```
class jobcomplex_type: public job_abstract_type
{
 // ...

private:
 // набор хэндлов потоков
 typedef std::vector<HANDLE> handles_type;
 // структура параметров для каждого потока
 struct threadparam_type
 {
  // указатель на набор хэндлов всех потоков
  const handles_type *phdls;
  // указатель на зависимости текущей работы
  const jobnums_type *pdeps;
  // указатель на работу для выполнения
  job_abstract_type *pjob;
 };

 // ожидание завершения любого потока из переданного набора
 static
 int wait_any(const handles_type &hdls)
 {
  assert(!hdls.empty());
  DWORD dw = ::WaitForMultipleObjects(
    hdls.size(), &hdls.front(), FALSE, INFINITE);
  assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdls.size());
  return dw - WAIT_OBJECT_0;
 }

 // ожидание завершения всех потоков из переданного набора
 static
 void wait_all(const handles_type &hdls)
 {
  if (hdls.size())
  {
   DWORD dw = ::WaitForMultipleObjects(
```

```
    hdls.size(), &hdls.front(), TRUE, INFINITE);
   assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdls.size());
  };
 }

 // функция, выполняемая потоками
 static
DWORD WINAPI thr_proc(LPVOID param)
 {
  threadparam_type &thrp = *static_cast<threadparam_type *>(param);

  // формирование списка хэндлов для ожидания
  // на основе зависимостей текущей работы
  handles_type hdls;
  jobnums_type::const_iterator it;
  for (it = thrp.pdeps->begin(); it != thrp.pdeps->end(); ++it)
   hdls.push_back((*thrp.phdls)[*it]);

  // ожидание завершения работ
  wait_all(hdls);

  // выполнение текущей работы
  thrp.pjob->run();

  return 0;
 }

 // ...
};
```

Структура параметров потока `threadparam_type`, помимо указателя на подлежащую выполнению работу, содержит также указатели на контейнер хэндлов всех потоков и контейнер зависимостей текущей работы. Функция потока на основе этих данных формирует набор хэндлов потоков, завершения которых текущий поток должен дождаться. После ожидания вызывается функция выполнения работы, и поток завершается.

Внутри функций ожидания в качестве передаваемого адреса массива хэндлов используется адрес первого элемента контейнера `std::vector`. Такой подход для передачи массива допустим по той причине, что, в соответствии со спецификацией STL, во-первых, все элементы контейнера `std::vector` обязаны храниться в виде массива, во-вторых, функция `std::front` возвращает ссылку на элемент этого массива.

Наконец, изменению снова подверглась функция `jobcomplex_type::run`:

```
class jobcomplex_type: public job_abstract_type
{
 // ...

 // выполнение всего комплекса работ
 void run(void)
 {
  // получим список подлежащих выполнению работ
  joblist_type joblist = get_joblist();
  // получим список зависимостей
  deplist_type deplist = get_deplist();
  // построим ярусно-параллельную структуру номеров работ
  multilevel_type multilevel = build(deplist);
```

```
  // список хэндлов всех потоков
  handles_type allhdl(joblist.size());
  // список всех структур параметров
  std::vector<threadparam_type> thrparam(joblist.size());

  multilevel_type::iterator it;
  // запустим по очереди каждый ярус работ
  for (it = multilevel.begin(); it != multilevel.end(); ++it)
  {
   handles_type lasthdl;
   // создадим все потоки текущего яруса
   int width = it->size();
   for (int i = 0; i < width; ++i)
   {
    // номер текущей работы
    int idx = (*it)[i];
    // заполняем структуру параметров потока
    thrparam[idx].phdls = &allhdl;
    thrparam[idx].pdeps = &deplist[idx];
    thrparam[idx].pjob = joblist[idx];

    // создание потока
    DWORD dwId;
    allhdl[idx] = ::CreateThread(
     NULL, 0,
     thr_proc, &thrparam[idx],
     0, &dwId);
    // добавляем в список хэндлов текущего яруса
    lasthdl.push_back(allhdl[idx]);
   };
   // прежде чем переходить к следующему ярусу,
   // дождемся момента, пока завершится
   // хотя бы один поток текущего яруса
   wait_any(lasthdl);
  };

  // ждем завершения всех оставшихся потоков
  wait_all(allhdl);
  // и закрываем хэндлы
  handles_type::iterator ht;
  for (ht = allhdl.begin(); ht != allhdl.end(); ++ht)
   ::CloseHandle(*ht);
 }
};
```

После построения необходимых структур создаются набор хэндлов всех потоков и набор структур параметров, передаваемых каждому потоку при создании. Набор хэндлов потоков необходим для ожидания их завершения. Он заполняется по мере создания потоков, при этом в силу поэтапности (поярусности) их создания нет риска, что какой-либо поток попытается ожидать завершения других потоков по незаполненным хэндлам. Если бы мы выбрали вариант одновременного создания всех потоков (что, как говорилось выше, менее предпочтительно), пришлось бы их создавать «спящими» (suspended), и «будить» их лишь после заполнения всего набора хэндлов, поскольку в противном случае они могли

Таблица 2.3. Таблица истинности некоторой логической функции

| $X_1$ | $X_2$ | $X_3$ | $Y_1$ | $Y_2$ | $Y_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

начать ожидание завершения других потоков по хэндлам, значения которых еще не определены. Полный набор структур параметров мы храним для того, чтобы быть уверенными, что каждый поток успеет прочитать из него свои параметры прежде, чем структура будет уничтожена.

В теле цикла каждого яруса заполняется соответствующая структура параметров потока, после чего создается поток и сохраняется его хэндл. После создания всех потоков текущего яруса, перед переходом к следующему ярусу выполняется ожидание завершения любого из них.

В конце функции выполняется ожидание завершения всех созданных потоков и закрытие хэндлов.

## 2.3. Симуляция выполнения логических схем

В качестве наглядного примера использования ЯПФ рассмотрим следующую задачу. Допустим, дана таблица истинности некоторой логической векторной функции от векторного аргумента (табл. 2.3). Как известно, по таблице истинности легко может быть построена соответствующая схема, состоящая из базовых вентилей «И», «ИЛИ» и «НЕ». Требуется реализовать симуляцию выполнения такой схемы, осуществляемую с учетом ее ярусно-параллельной структуры.

На основе, к примеру, представления в совершенной дизъюнктивной нормальной форме (СДНФ) значения рассматриваемой функции в зависимости от значений аргументов могут быть выражены следующим образом:

$$Y_1 = \neg X_1 \neg X_2 X_3 \vee X_1 \neg X_2 \neg X_3 \vee X_1 X_2 X_3 = (\neg X_1 \neg X_2 \vee X_1 X_2) X_3 \vee X_1 \neg X_2 \neg X_3;$$
$$Y_2 = \neg X_1 X_2 \neg X_3 \vee X_1 X_2 \neg X_3 = (\neg X_1 \vee X_1) X_2 \neg X_3 = X_2 \neg X_3;$$
$$Y_3 = \neg X_1 \neg X_2 \neg X_3 \vee \neg X_1 X_2 X_3 \vee X_1 X_2 \neg X_3 = (\neg X_1 \neg X_2 \vee X_1 X_2) \neg X_3 \vee \neg X_1 X_2 X_3.$$

Мы не затрагиваем вопросы минимизации логических функций, вследствие чего здесь произведены лишь некоторые очевидные упрощения. Одна из возможных схем, реализующих такую логическую функцию, приведена на рис. 2.8.

Приведенная схема не оптимальна, но довольно наглядна. Она основана на разложении процесса вычисления значений рассматриваемой функции в следующую последовательность элементарных операций:

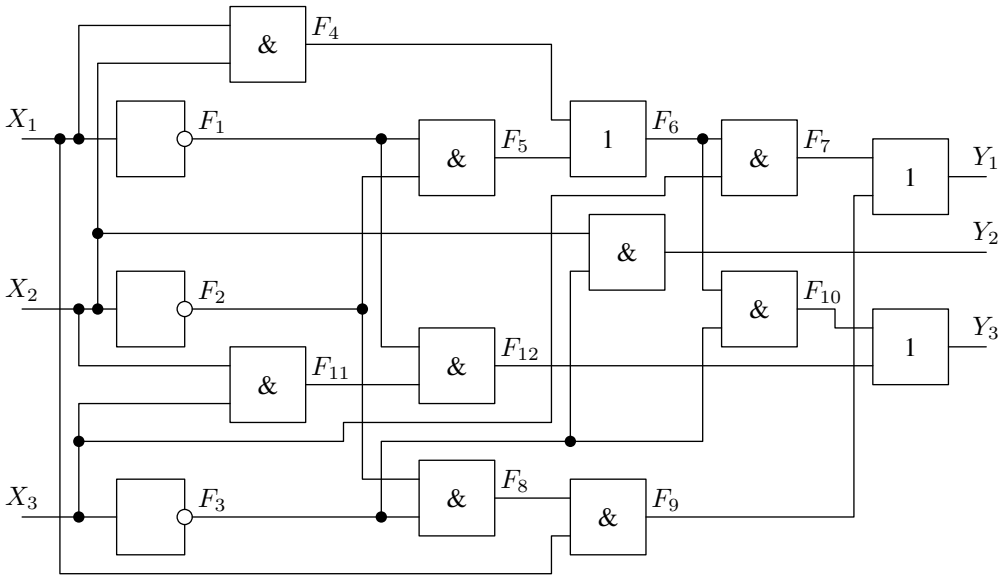



Рис. 2.8. Схема соединения логических вентилей

$$F_1 = \neg X_1; \quad F_2 = \neg X_2; \quad F_3 = \neg X_3;$$
$$F_4 = X_1 X_2; \quad F_5 = F_1 F_2; \quad F_6 = F_4 \vee F_5;$$
$$F_7 = F_6 X_3; \quad F_8 = F_2 F_3; \quad F_9 = X_1 F_8;$$
$$F_{10} = F_6 F_3; \quad F_{11} = X_2 X_3; \quad F_{12} = F_1 F_{11};$$
$$Y_1 = F_7 \vee F_9; \quad Y_2 = X_2 F_3; \quad Y_3 = F_{10} \vee F_{12}.$$

Каждой логической операции сопоставляется выполнение отдельной работы. После построения структурной таблицы зависимостей и получения номеров ярусов операций становится ясно, что ширина и высота соответствующей ЯПФ равны пяти (табл. 2.4). Это означает, что при наличии пяти параллельных ресурсов все выходные значения функции будут вычислены на основе входных значений за пять шагов, в то время как в последовательном варианте потребовалось бы пятнадцать.

Похожим образом может быть реализована симуляция параллельного выполнения произвольной схемы вентилей, если она не содержит циклических зависимостей. Приведем вариант такой реализации на основе рассмотренных выше классов выполнения комплекса работ.

Составная логическая схема реализуется отдельным классом, объект которого хранит весь набор вентилей и связей между ними. Каждый вентиль наследуется от класса `job_abstract_type` и выполняет соответствующую ему логическую операцию в рамках реализации функции `run`. Выходные значения всех сработавших вентилей содержатся в едином хранилище, доступ к которому предоставляется специальным объектом типа `storage_type`. Также объект составной схемы содержит объект класса `jobcomplex_type`, который осуществляет непосредственно выполнение всего набора вентилей. Ниже приведена реализация такого класса составной схемы:

```
// составная логическая схема
class circuit_type
{
public:
```

Таблица 2.4. Зависимости и ярусы логических операций

| Операция | Зависимость | Ярус |
|---|---|---|
| $F_1$ | — | 0 |
| $F_2$ | — | 0 |
| $F_3$ | — | 0 |
| $F_4$ | — | 0 |
| $F_5$ | $F_1$, $F_2$ | 1 |
| $F_6$ | $F_4$, $F_5$ | 2 |
| $F_7$ | $F_6$ | 3 |
| $F_8$ | $F_2$, $F_3$ | 1 |
| $F_9$ | $F_8$ | 2 |
| $F_{10}$ | $F_3$, $F_6$ | 3 |
| $F_{11}$ | — | 0 |
| $F_{12}$ | $F_1$, $F_{11}$ | 1 |
| $Y_1$ | $F_7$, $F_9$ | 4 |
| $Y_2$ | $F_3$ | 1 |
| $Y_3$ | $F_{10}$, $F_{12}$ | 4 |

```
 // уникальный идентификатор логического элемента
 typedef int id_type;
 // значения на выходе соответствующих элементов
 typedef std::map<id_type, bool> valueset_type;

private:
 // множество идентификаторов
 typedef std::set<id_type> idset_type;

 // хранилище промежуточных логических значений
 class storage_type
 {
 public:
  // внесение очередного значения
  void put_value(id_type id, bool value)
  {
   // ...
  }
  // получение некоторого значения
  bool get_value(id_type id) const
  {
   // ...
  }
 };

 // логический вентиль
 class gate_type: public job_abstract_type
 {
 public:
  // выполняемая вентилем логическая операция
  enum op_type { AND, OR, NOT };
 private:
```

```
  // идентификаторы вентиля и его входов
  const id_type m_id, m_in1, m_in2;
  // идентификатор операции
  const op_type m_op;
  // разделяемое хранилище данных
  storage_type &m_storage;
public:
  // конструктор
  gate_type(
   id_type id, id_type in1, id_type in2, op_type op,
   storage_type &storage):
   m_id(id), m_in1(in1), m_in2(in2), m_op(op),
   m_storage(storage)
  {}
  // выполнение работы
  void run(void)
  {
   switch (m_op)
   {
   case AND:
   case OR:
    // логические операции И, ИЛИ
    {
     bool in1 = m_storage.get_value(m_in1);
     bool in2 = m_storage.get_value(m_in2);
     bool out = (m_op == AND) ? (in1 && in2) : (in1 || in2);
     m_storage.put_value(m_id, out);
    };
    break;
   case NOT:
    // логическая операция НЕ
    m_storage.put_value(m_id, !m_storage.get_value(m_in1));
    break;
   };
  }
};
typedef std::map<id_type, gate_type> gates_type;


// множество логических вентилей
gates_type m_gates;
// идентификаторы входов и выходов
idset_type m_inids, m_outids;
// комплекс работ
jobcomplex_type m_jobcomplex;
// хранилище промежуточных значений
storage_type m_storage;


// внесение нового бинарного или унарного вентиля
void add_gate(
 id_type id, id_type in1, id_type in2,
 gate_type::op_type op)
{
 // id не должен быть в списках уже внесенных
```

```
  assert(m_inids.find(id) == m_inids.end());
  assert(m_gates.find(id) == m_gates.end());
  // обе зависимости, наоборот, должны
  assert(
   m_inids.find(in1) != m_inids.end() ||
   m_gates.find(in1) != m_gates.end());
  assert(
   m_inids.find(in2) != m_inids.end() ||
   m_gates.find(in2) != m_gates.end());

  // создаем новый логический элемент
  gates_type::iterator it = m_gates.insert(
   gates_type::value_type(
    id,
    gate_type(id, in1, in2, op, m_storage))
  ).first;

  gates_type::iterator in1it = m_gates.find(in1);
  gates_type::iterator in2it = m_gates.find(in2);
  // добавляем новую работу в комплекс
  m_jobcomplex.add_job(it->second);
  // если зависимость найдена среди работ, вносим ее в комплекс
  if (in1it != m_gates.end())
   m_jobcomplex.add_dependence(it->second, in1it->second);
  if (in2it != m_gates.end())
   m_jobcomplex.add_dependence(it->second, in2it->second);
 }

public:
 // внесение идентификатора входа схемы
 void add_input(id_type id)
 {
  assert(m_inids.find(id) == m_inids.end());
  assert(m_gates.find(id) == m_gates.end());
  m_inids.insert(id);
 }
 // внесение идентификатора выхода схемы
 void add_output(id_type id)
 {
  assert(
   m_inids.find(id) != m_inids.end() ||
   m_gates.find(id) != m_gates.end());
  m_outids.insert(id);
 }
 // внесение элемента И
 void add_and(id_type id, id_type in1, id_type in2)
 {
  add_gate(id, in1, in2, gate_type::AND);
 }
 // внесение элемента ИЛИ
 void add_or(id_type id, id_type in1, id_type in2)
 {
  add_gate(id, in1, in2, gate_type::OR);
 }
 // внесение элемента НЕ
```

```
 void add_not(id_type id, id_type in)
 {
  add_gate(id, in, in, gate_type::NOT);
 }

 // выполнение схемы при заданных входных значениях
 valueset_type execute(const valueset_type &input)
 {
  assert(input.size() == m_inids.size());
  idset_type::iterator it;
  // заполнение хранилища данных входными значениями
  for (it = m_inids.begin(); it != m_inids.end(); ++it)
  {
   valueset_type::const_iterator f = input.find(*it);
   assert(f != input.end());
   m_storage.put_value(f->first, f->second);
  };

  // параллельное выполнение комплекса работ
  m_jobcomplex.run();

  valueset_type output;
  // изъятие выходных значений из хранилища данных
  for (it = m_outids.begin(); it != m_outids.end(); ++it)
   output[*it] = m_storage.get_value(*it);
  return output;
 }
};
```

Вложенный класс `gate_type` представляет произвольный вентиль схемы. В зависимости от заданной при создании объекта константы он выполняет одну из трех базовых операций. В рамках выполнения работы вентиль обращается к объекту хранилища типа `storage_type` для получения входных значений и для помещения выходного.

В составной схеме каждому вентилю сопоставлен целочисленный идентификатор. Эти идентификаторы используются при указании входных зависимостей вентилей, а также для хранения промежуточных и выходных значений в хранилище.

Клиентскому коду предоставляются функции добавления вентилей (`add_and`, `add_or`, `add_not`) и функции указания идентификаторов входов и выходов схемы (`add_input`, `add_output`). Идентификаторы входов используются при проверке зависимостей и при начальном заполнении хранилища. Идентификаторы выходов требуются после выполнения схемы для изъятия из хранилища лишь необходимых значений.

Внесение вентилей в схему производится внутренней функцией `add_gate`. В ней осуществляется создание объекта `gate_type` и сохранение его во внутреннем контейнере. Также в ней осуществляется добавление новой работы и соответствующих зависимостей в комплекс. Поскольку для хранения вентилей выбран контейнер `std::map`, при добавлении новых вентилей размещение в памяти ранее внесенных не меняется, в связи с чем указатели на работы, сохраненные в объекте комплекса работ, остаются валидными.

Функция `execute` осуществляет как таковое выполнение схемы. На входе принимается множество входных значений, которое перед выполнением схемы помещается в хранилище промежуточных данных. После выполнения комплекса работ из хранилища изымаются помещенные туда во время выполнения вентилей выходные значения схемы, множество которых возвращается функцией `execute`.

В приведенном выше коде не было определено содержимое вложенного класса `storage_type` по причине зависимости его реализации от механизма распараллеливания. Поскольку объект этого класса предоставляет совместный доступ к одним данным различным параллельным ресурсам, в некоторых случаях оказывается необходимым наличие разграничения доступа. К примеру, в случае реализации распараллеливания на основе OpenMP такое разграничение может быть осуществлено на основе директивы `critical`:

```
class circuit_type
{
 // ...

 // хранилище промежуточных логических значений
 class storage_type
 {
 private:
  // внесенные логические значения
  valueset_type m_values;
 public:
  // внесение очередного значения
  void put_value(id_type id, bool value)
  {
   #pragma omp critical (storage)
   m_values[id] = value;
  }
  // получение некоторого значения
  bool get_value(id_type id) const
  {
   bool rc;
   #pragma omp critical (storage)
   {
    valueset_type::const_iterator it = m_values.find(id);
    assert(it != m_values.end());
    rc = it->second;
   };
   return rc;
  }
 };

 // ...
};
```

Все внесенные в хранилище значения содержатся в контейнере, доступ к которому осуществляется атомарно. В случае распараллеливания на основе потоков Windows API аналогичный механизм может быть реализован с помощью предоставляемых этим интерфейсом функций для работы с критическими секциями. Наконец, при распараллеливании между различными процессами общие данные могут содержаться, к примеру, в каких-либо разделяемых областях памяти или файлах.

Написание клиентского кода для приведенного класса составной схемы оказывается столь же простым, как и в случае реализации комплекса работ. К примеру, выполнение схемы, изображенной на рис. 2.8, реализуется следующим кодом:

```
// идентификаторы элементов схемы
enum {
 IN1, IN2, IN3,
 F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12,
```

```
 OUT1, OUT2, OUT3
};

// схема
circuit_type circuit;

// входные идентификаторы
circuit.add_input(IN1);
circuit.add_input(IN2);
circuit.add_input(IN3);
// логические элементы
circuit.add_not(F1, IN1);
circuit.add_not(F2, IN2);
circuit.add_not(F3, IN3);
circuit.add_and(F4, IN1, IN2);
circuit.add_and(F5, F1, F2);
circuit.add_or(F6, F4, F5);
circuit.add_and(F7, IN3, F6);
circuit.add_and(F8, F2, F3);
circuit.add_and(F9, IN1, F8);
circuit.add_and(F10, F3, F6);
circuit.add_and(F11, IN2, IN3);
circuit.add_and(F12, F1, F11);
circuit.add_or(OUT1, F9, F7);
circuit.add_and(OUT2, IN2, F3);
circuit.add_or(OUT3, F12, F10);
// выходные идентификаторы
circuit.add_output(OUT1);
circuit.add_output(OUT2);
circuit.add_output(OUT3);

circuit_type::valueset_type input, output;
// формирование множества входных значений
input[IN1] = /* ... */;
input[IN2] = /* ... */;
input[IN3] = /* ... */;
// выполнение схемы
output = circuit.execute(input);
// чтение выходных значений
/* ... */ = output[OUT1];
/* ... */ = output[OUT2];
/* ... */ = output[OUT3];
```

При заполнении схемы каждому вентилю дается уникальный идентификатор. Помимо самих вентилей, в схему заносится информация об идентификаторах входов, а также об идентификаторах вентилей, выходные значения которых будут являться выходными значениями схемы.

# Глава 3.

# Сети конечных автоматов

В настоящей главе мы рассмотрим схемы создания параллельных программ на основе сетей конечных автоматов. Основа параллелизма такого подхода заключается в том, что каждый автомат в сети является автономным элементом и не связан с другими автоматами иначе, кроме как входными и выходными значениями (сигналами). Таким образом, на протяжении каждого такта все автоматы сети могут работать параллельно. Такая схема в какой-то степени напоминает рассмотренное в главе 2 выполнение яруса работ, но добавляет возможность введения обратной связи (циклических зависимостей).

Разумеется, при реализации такого подхода придется прибегнуть к методам автоматного программирования (Automata-Based Programming, State-Based Programming) для преобразования алгоритма программы к автоматному виду. Существует немало работ, посвященных этой теме, в частности, рассмотрению SWITCH-технологии [39, 40, 44], КА-технологии [27] и преобразованию алгоритмов в автоматные [43, 45], поэтому мы не будем останавливаться на ней подробно. Отметим, что есть немало областей программирования, в которых автоматный подход может существенно облегчить проектирование и реализацию.

## 3.1. Программирование конечных автоматов

Изложим вкратце основы программирования с использованием конечных автоматов. Описания конечного автомата (Finite-State Machine, Finite-State Automaton) разнятся в деталях от издания к изданию в зависимости от области применения, в частности, от ориентации на программную или аппаратную реализацию. Более того, даже при ориентации лишь на программную реализацию описания автомата различаются, особенно среди сравнительно недавних публикаций [13, 20].

Мы здесь приведем далеко не новое очень обобщенное описание конечного автомата [32], достаточное для понимания принципов создания сетей конечных автоматов и применения их в параллельном программировании. Конечный автомат характеризуется следующими параметрами:

- конечное множество состояний $\{S_0, S_1, \ldots, S_N\}$ (внутренний алфавит);
- конечное множество совокупностей входных значений (входной алфавит);
- конечное множество совокупностей выходных значений (выходной алфавит);
- начальное состояние $S_0$ (в котором автомат находится перед началом работы);

Таблица 3.1. Таблица переходов конечного автомата

| | A | B | C |
|---|---|---|---|
| 00 | B | C | C |
| 01 | B | A | B |
| 10 | C | C | A |
| 11 | B | B | B |

Таблица 3.2. Таблица выходов конечного автомата

| | A | B | C |
|---|---|---|---|
| 00 | a | a | d |
| 01 | b | c | e |
| 10 | d | d | a |
| 11 | c | b | c |

- конечное состояние (или множество таковых), т.е. такое состояние, по достижении которого работа автомата считается завершенной (зачастую удобно использовать начальное состояние $S_0$);

- таблица переходов (функция переходов, зависимость следующего состояния от текущего состояния и текущей совокупности входных значений);

- таблица выходов (функция выходов, зависимость совокупности текущих выходных значений от текущего состояния и текущей совокупности входных значений).

Конечность множества состояний говорит о том, что автомат (именно поэтому он называется конечным) обладает ограниченной памятью. В некоторых приложениях используются автоматы с неограниченной памятью (к примеру, в синтаксических анализаторах). Удобство совпадения начального и конечного состояний заключается в возможности повторного использования: после завершения работы автомата он снова готов к запуску.

Таблицы, указанные в последних двух пунктах, при некоторых допущениях могут полностью описывать все остальные перечисленные характеристики автомата. К примеру, в табл. 3.1 и 3.2 мы видим таблицы переходов и выходов некоторого автомата. Из таблиц видно множество состояний $\{A, B, C\}$, множество входов $\{00, 01, 10, 11\}$ и множество выходов $\{a, b, c, d, e\}$. Если оговориться, что в первом столбце таблицы всегда описывается начальное/конечное (выключенное) состояние, известно и оно: $S_0 = A$. Входом такого автомата может быть, к примеру, двубитное целое число или два логических значения, выход — символьный.

Помимо таблиц, описание автомата также может быть представлено диаграммой состояний (графом переходов). Вершины такого графа представляют весь набор возможных состояний автомата; направленные дуги, характеризующие переходы, помечаются входными данными, соответствующими переходу, и выходными данными автомата в момент такого перехода (в программно реализованных автоматах часто интерпретируются соответственно как локальное событие и действие на переходе [20]). К примеру, на рис. 3.1 представлен граф переходов конечного автомата, описанного табл. 3.1 и 3.2.

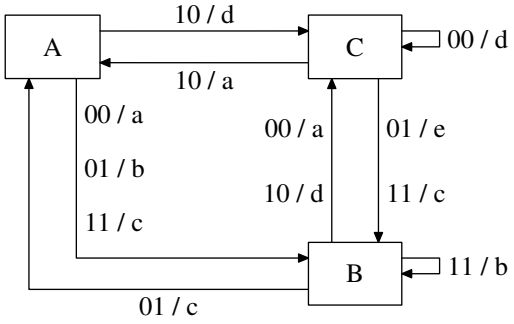



Рис. 3.1. Граф переходов конечного автомата

Сказанное выше касалось скорее математического понятия конечного автомата. В условиях программной реализации жизненный цикл автомата представляется программным циклом со сменой состояний от начального до конечного. В теле цикла на каждом такте происходит проверка текущего состояния и значений входных сигналов, после чего осуществляется возможное изменение состояния и генерация значений выходных сигналов. В программе, разумеется, могут быть выполнены дополнительные действия, характерные для предметной области. К примеру, в управляющей программе могут быть выполнены вызовы соответствующих функций управления оборудованием.

В качестве иллюстрации можно привести простой вариант реализации программного автомата, диаграмма состояний которого изображена на рис. 3.1.

```
char state = 'A';
do
{
 bool in1, in2;
 char out;
 get_input(in1, in2);
 switch (state)
 {
 case 'A':
  out = in1 ? (in2 ? 'c' : 'd') : (in2 ? 'b' : 'a');
  state = (in1 && !in2) ? 'C' : 'B';
  break;
 case 'B':
  out = in1 ? (in2 ? 'b' : 'd') : (in2 ? 'c' : 'a');
  state = !in2 ? 'C' : (in1 ? 'B' : 'A');
  break;
 case 'C':
  out = in1 ? (in2 ? 'c' : 'a') : (in2 ? 'e' : 'd');
  state = in2 ? 'B' : (in1 ? 'A' : 'C');
  break;
 };
 put_output(out);
} while (state != 'A');
```

Работа автомата представлена циклом, условие завершения которого — достижение начального состояния. Текущее состояние автомата хранится в его внутренней переменной. В теле цикла на каждой итерации (каждом такте) производится чтение входных сигналов, после чего на основе текущего состояния и считанных значений формируются и выводятся значения выходных сигналов автомата.

Автоматное программирование обычно не во всех отношениях следует модели конечных автоматов. Зачастую используются упрощения, диктуемые традиционным программированием. Примером тому может служить введение каких-либо переменных, внешне не привязанных к состоянию, но по существу являющихся его частью. В этих ситуациях в течение такта работы автомата выполняются модификации его внутренних переменных, даже когда считается, что сохранено предыдущее состояние. В соответствии с приведенным выше описанием автомата, это является сменой состояния. Однако с целью упрощения программирования и повышения наглядности принципов функционирования программы таким путем производится группировка некоторого количества схожих состояний как одного. К примеру, если автомат, помимо прочей работы, последовательно обрабатывает элементы внутреннего массива, происходит смена состояний «Обработка элемента 0», «Обработка элемента 1», «Обработка элемента 2» и т.д. Однако в программе мы можем себе позволить хранение текущего индекса обрабатываемого элемента отдельно от переменной состояния, объединив все перечисленные состояния в единое состояние «Обработка массива», тем самым и упростив программирование, и повысив читабельность программы.

Автоматное программирование оказывается удобным при реализации широкого круга задач, включающего программы логического управления, компиляторы, игры. Однако существуют задачи, для которых автоматное программирование может лишь утяжелить процесс реализации. Такой подход оказывается сродни попытке воплотить какой-нибудь сложный вычислительный алгоритм на машине Тьюринга (хотя это довольно грубое сравнение). Написание вычислительной процедуры классическим путем, включая отладку, может оказаться на порядок менее трудоемким, нежели преобразование алгоритма в автоматный, при этом сохранится большее соответствие между исходным описанием алгоритма математическими формулами и программным представлением. Хорошим примером тому является приведенная в этой главе автоматная реализация алгоритма суммирования сдваиванием. Любые среда/средство/язык программирования, предназначенные для реализации какого-либо класса задач, должны, прежде всего, предоставлять простоту и наглядность описания стоящей задачи. Далеко не все алгоритмы описываются автоматами наглядно, несмотря на то, что, в принципе, могут быть ими описаны.

Считается, что автоматные программы практически не требуют отладки, либо требуют ее в гораздо меньшей степени [5, 20, 27]. Отчасти это так. В идеале алгоритм решения задачи должен быть построен еще до начала процесса его реализации в программе. К примеру, возможно построение алгоритма с последующей его верификацией или же с верификацией прямо в процессе построения (так называемое доказательное программирование). Однако этот процесс довольно трудоемок, и в реальности алгоритм решения поставленной задачи зачастую строится «на ходу», т.е. прямо в процессе его реализации, с отладкой до тех пор, пока «не заработает». В результате такого создания алгоритмов последние, как правило, не имеют доказательной основы и оказываются нерабочими при некоторых исходных данных. В случае же автоматного программирования создание алгоритма в принудительном порядке отвязывается от реализации и отодвигается на ранние этапы разработки, поскольку для преобразования в автоматный алгоритм исходный должен уже быть построен и выверен. Разумеется, после трансляции полученного таким образом автоматного алгоритма на формальный язык программа будет содержать гораздо меньше ошибок, нежели при формировании алгоритма в процессе реализации.

Одной из важных особенностей автоматного программирования является возможность визуализации алгоритма с последующей генерацией программного кода без участия программиста. К примеру, если характеристики автомата уже описаны таблицами переходов и выходов или же графически с помощью диаграммы состояний, программный код такого автомата может быть сгенерирован автоматически соответствующими программными сред-

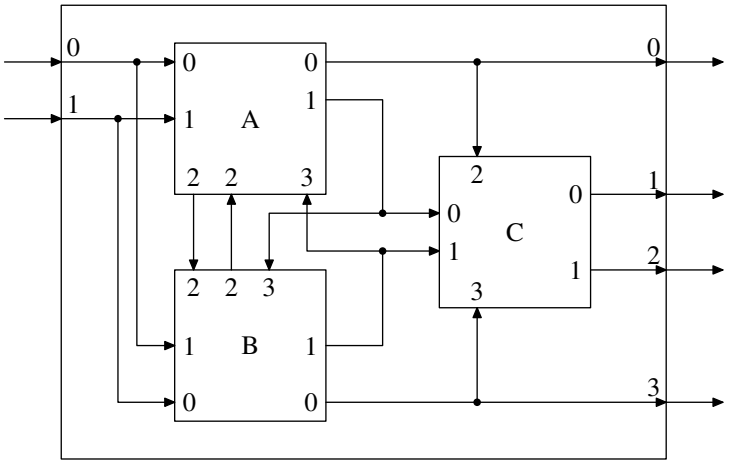



Рис. 3.2. Пример сети конечных автоматов

ствами. На сегодняшний день существует немало подобных средств генерации (к примеру, FSMDesigner, FSMGenerator, компилятор Ragel и т.п.), однако мы на них останавливаться не будем.

## 3.2. Параллелизм сетей конечных автоматов

Сами по себе реализованные программно конечные автоматы являются последовательными и в общем случае потенциального параллелизма в себе не несут. Параллелизм может содержаться при вычислении условий перехода, однако для произвольного автоматного алгоритма далеко не всегда это позволит повысить эффективность вычислений. Зачастую такое распараллеливание может оказаться просто дублированием одних и тех же вычислений во всех параллельных ресурсах. Однако ситуация совершенно иная в отношении сетей конечных автоматов.

Сетью конечных автоматов называют некоторое количество конечных автоматов, связанных между собой входными и выходными сигналами. Вход каждого автомата сети может быть подключен к входу сети или выходу другого автомата. Выход каждого автомата может быть подключен, соответственно, к входу другого автомата или выходу сети. К одному источнику сигнала (выходу автомата или входу сети) может быть подключено более одного приемника сигнала (входа автомата или выхода сети). К одному же приемнику сигнала может быть подключен только один источник. Пример связи автоматов в некоторой автоматной сети можно увидеть на рис. 3.2. Здесь изображена сеть, состоящая из трех автоматов. Входы и выходы автоматов, а также всей автоматной сети, пронумерованы для удобства учета связей, что будет использовано нами позже.

Все автоматы сети работают синхронно, т.е. работа следующего такта в каждом автомате не начинается до тех пор, пока не завершится работа предыдущего такта во всех автоматах. На протяжении каждого такта, т.е. между моментами чтения входных сигналов и записи выходных, все автоматы работают независимо друг от друга, поэтому могут выполнять работу параллельно.

Программно реализованная автоматная сеть, в отличие от аппаратной, не нуждается во внешней генерации синхроимпульсов, поэтому длительность такта обычно не является постоянной величиной. Естественным способом синхронизации параллельно работающих автоматов является установка барьера. В такой ситуации длительность такта работы сети

определяется максимальной длительностью такта среди всех автоматов сети. Иными словами, такт работы сети завершается в тот момент, когда закончит обработку такта последний автомат сети.

Реализация параллельной программы сетью конечных автоматов имеет смысл лишь в том случае, когда такт занимает достаточно много времени по сравнению с накладными расходами на передачу данных между автоматами и обеспечение их синхронизации. В частности, представленный ниже алгоритм суммирования сдваиванием носит лишь иллюстративный характер, поскольку накладные расходы на действия, осуществляемые для обеспечения работы всех вычисляющих автоматов на одном такте, существенно выше длительности полезного действия, выполняемого каждым таким автоматом на одном такте (суммирование двух чисел).

С точки зрения функционирования сеть автоматов также представляет собой некий сложный автомат, состояние которого на каждом такте определяется совокупностью состояний внутренних автоматов сети (а также значений сигналов, передаваемых между автоматами). Таким образом, возможно построение автоматных сетей с многоуровневым вложением. При программной реализации такое вложение может иметь смысл по различным причинам. К примеру, такой подход может позволить создание сложных распараллеленных базовых элементов на основе элементарных. Еще одна причина целесообразности такого вложения будет указана ниже при рассмотрении примера программной реализации.

Для более подробного ознакомления с теорией конечных автоматов рекомендуется обратиться к специально посвященной этой теме литературе [11, 32]. Помимо учебников, доступно много довольно современных статей [5, 20, 25, 26, 40, 42, 44], немало публикаций можно найти в сети Internet.

## 3.3. Пример программной реализации

Здесь для иллюстрации мы рассмотрим один из вариантов программной реализации автоматов в некотором обобщенном виде, пригодном для использования при реализации достаточно широкого класса конечно-автоматных сетей. Прежде чем перейти к обсуждению исходного текста, необходимо оговорить некоторые требующие учета при описании предлагаемой реализации моменты.

Прежде всего, в автоматной сети содержится общая промежуточная область «выходов», т.е. единая общая область памяти, куда попадают все значения выходных сигналов автоматов после каждого такта, и откуда эти значения будут считаны на следующем такте. Эта область должна содержать именно «выходы» автоматов, а не «входы», вследствие возможности наличия связей типа «один выход — много входов». В противном случае возникают необходимость дублирования информации и дополнительные сложности реализации.

Во избежание перезаписи непрочитанных данных чтение входных данных всеми автоматами на каждом такте должно быть отделено во времени от записи ими выходных данных. В нашем случае действия автомата разделены на три функции: чтение входных данных, как таковые внутренние действия автомата и запись выходных данных. Интерпретация входных данных, анализ локальных событий, действия на переходе и сам переход объединены в одну функцию между чтением и записью данных. На интервале между чтением входов и записью выходов необходим лишь один барьер, поэтому функция действия может быть выполнена без барьера как сразу после чтения, так и непосредственно перед записью.

Будем предполагать в нашем примере, что все входные и выходные данные автоматов имеют один тип — целочисленный. Общая область памяти для значений выходов автоматов строится программно на основе описаний связей автоматов между собой. При этом может

быть осуществлена проверка корректности связей и соответствия их набору автоматов. Каждая связь в наборе может быть представлена в одной из трех форм:

- от одного автомата к другому: $(FSM_{num}, index) \rightarrow (FSM_{num}, index)$;
- от входа сети к автомату: $(NET^{IN}, index) \rightarrow (FSM_{num}, index)$;
- от автомата к выходу сети: $(FSM_{num}, index) \rightarrow (NET^{OUT}, index)$.

Здесь первый элемент каждой пары в скобках характеризует вход сети, выход сети либо конкретный автомат сети. Второй элемент каждой пары характеризует индекс конкретного сигнала (входа или выхода) для объекта, указанного первым элементом пары. В нашей реализации сигналы будут нумероваться с нуля. При проверке корректности связей должны учитываться следующие факторы:

- одному входу автомата или выходу сети соответствует только один выход автомата или вход сети;
- вход сети может быть только в левой части связи;
- выход сети может быть только в правой части связи.

При проверке соответствия набору автоматов может быть осуществлена проверка границ номеров входов и выходов (они должны соответствовать их количеству для каждого автомата или всей сети), а также может быть выполнена проверка отсутствия «висящих концов» (выходов, не присоединенных ни к одному входу, и наоборот).

Однако в случае реализации распределенной автоматной сети выполнение проверки соответствия набору автоматов оказывается неэффективным вследствие рассредоточенности автоматов между узлами и, как следствие, сравнительно высоких затратах на получение информации о количестве входов и выходов. Поскольку такая проверка требуется, как правило, лишь на этапе отладки автоматного алгоритма, ее можно реализовать в рамках последовательной программы.

При формировании общей области выходных данных для распределенной автоматной сети удобнее всего ориентироваться только на набор связей, поскольку при условии отсутствия «висящих концов» его вполне достаточно для вычленения информации о количестве выходов и входов автоматов и, соответственно, построения общей области памяти. Это позволит исключить межпроцессные коммуникации для определения количества входов и выходов каждого автомата.

Во избежание ошибок при указании связей удобнее всего их генерировать автоматически путем интерпретации графического представления сети. Для этой цели, как и для генерации программного кода конкретного автомата, могут быть созданы специальные программные средства, подобные описанным в [5].

Вложение сетей, помимо очевидных преимуществ модульного построения, имеет смысл также для локализации связей. При универсальной (не привязанной к структуре конкретной автоматной сети) реализации параллельно работающей сети в системах с распределенной памятью после каждого такта потребуется рассылка выходных данных каждого автомата всем остальным автоматам сети. Зачастую это нецелесообразно, поскольку редко бывает так, что каждому автомату нужны результаты работы всех других автоматов. Выходы конкретного автомата могут требоваться на входе лишь небольшому количеству других автоматов сети, рассылка же остальным в этом случае — лишь напрасная растрата ресурсов. В таких ситуациях бывает удобно объединить некоторые группы автоматов в отдельные подсети, не затрагивая при этом универсальность реализации. Каждый автомат

такой группы обменивается данными с другими автоматами своей группы, не производя ненужного обмена с остальными автоматами всей сети.

Изначально автомат находится в выключенном состоянии, которое является начальным и конечным. При завершении работы автомат должен перейти в начальное состояние. Будем считать, что сеть завершила свою работу, когда завершили работу, т.е. перешли в начальное состояние, все автоматы сети.

Также будем считать, что если автомат завершил работу, он не выдает на выходе никаких данных. Это дополнение позволит избежать потерь данных при проверке завершения работы сети автоматов. Иначе возможна ситуация, когда один автомат сети, будучи уже выключенным, передает данные другому автомату, еще не включенному, при этом работа сети завершается, хотя алгоритм еще не отработал.

В момент старта все автоматы пребывают в выключенном состоянии, поэтому сигналы внутри сети между выходами одних автоматов и входами других отсутствуют.

В некоторых случаях в распределенных вычислениях проверять факт завершения работы всеми автоматами сети оказывается чересчур накладно, поэтому может оказаться удобным выполнять проверку завершения работы сети более простыми средствами, нежели путем коммуникации всех процессов. К примеру, проверка завершения может осуществляться путем анализа одного из выходных сигналов сети, предназначенного для указания наличия на выходе финального результата работы сети. Здесь следует оговориться, что, при введенных выше условиях, таким путем может быть выявлен не факт завершения работы сети, а лишь тот факт, что сеть завершит работу на следующем такте, поскольку после завершения работы сеть не выдает никаких выходных сигналов.

Далее рассмотрим различные варианты реализации классов для построения автоматных сетей с использованием описанного подхода. Ниже будут приведены примеры некоторых автоматных сетей на основе приведенных классов.

### 3.3.1. Реализация с использованием OpenMP

В приложении В приведен исходный текст реализации базовых классов конечного автомата и автоматной сети. Выполнение тактов в классе автоматной сети распараллелено с помощью директив OpenMP. Оба этих класса унаследованы от абстрактного класса конечного автомата (`fsm_abstract_type`), объявляющего следующий интерфейс:

```
// абстрактный тип автомата
class fsm_abstract_type
{
public:
 // тип состояния автомата
 typedef int state_type;
 // тип входных и выходных данных
 typedef int signal_type;
 // полный набор входных или выходных сигналов
 typedef std::vector<signal_type> signals_type;

 // количество входов
 virtual
 int number_input(void) const = 0;
 // количество выходов
 virtual
 int number_output(void) const = 0;
 // передать автомату вектор входных данных
 virtual
```

```
  void put_input(const signals_type &input) = 0;
  // выполнить такт работы автомата
  virtual
  void do_work(void) = 0;
  // получить от автомата вектор выходных данных
  virtual
  signals_type get_output(void) const = 0;
  // проверка, находится ли автомат в начальном состоянии
  virtual
  bool is_off(void) const = 0;
};
```

Первые две функции возвращают информацию о количестве входных и выходных сигналов автомата. Следующие три функции выполняют соответственно работу по передаче автомату входных данных, выполнению автоматом внутренних действий и получению от автомата выходных данных. Наконец, последняя функция возвращает информацию о том, находится ли автомат в настоящий момент в выключенном (начальном/конечном) состоянии.

На основе абстрактного типа автомата создается класс элементарного конечного автомата `fsm_type`. Функции этого класса определяют в общем виде работу простого последовательного конечного автомата (за исключением выполнения конкретных действий). Помимо возврата основных характеристик и хранения входных и выходных значений автомата, этот класс инкапсулирует хранение и использование таблицы обработчиков состояний.

Обработчик состояния — функция-член класса конкретного автомата, который будет унаследован от класса `fsm_type`. Унаследованный класс объявляет набор состояний, каждому из них сопоставляет некий обработчик и добавляет его в таблицу обработчиков с помощью функции `add_handler`.

Во время каждого такта в зависимости от текущего состояния вызывается тот или иной обработчик. Один обработчик может быть подключен более чем к одному состоянию. Обычно для выбора обработчиков рекомендуют использовать оператор `switch` или его аналоги в других языках [5, 20, 42]. Однако такая конструкция нам не подходит, поскольку она работает лишь с предопределенными константами и неудобна при программном формировании автомата, к примеру, на основе каких-либо конфигурационных данных. В этой ситуации приходится хранить информацию о соответствии обработчиков состояниям в виде таблицы. Мы воспользовались для этих целей стандартным шаблоном `std::map`, который обеспечивает логарифмическое от количества состояний время поиска. Если после поиска и выполнения обработчика в функции `do_work` состояние автомата изменилось на выключенное, производится обнуление его выходных сигналов.

Наконец, на основе абстрактного типа автомата объявляется класс сложного автомата — автоматной сети (`fsmnet_type`).

Конструктор и деструктор этого класса осуществляют соответственно создание и уничтожение внутренних автоматов средствами так называемой «фабрики сети». Объект класса фабрики сети, объявляемого клиентским кодом на основе абстрактного типа `factory_abstract_type`, передается конструктору класса `fsmnet_type` и содержит всю необходимую информацию о создаваемой автоматной сети.

Два слова о фабрике сети. Этот класс введен по той причине, что автоматы в явном виде должны создаваться клиентским кодом, поскольку только ему известны класс автомата и сигнатура соответствующего конструктора. В то же время список созданных автоматов не должен передаваться конструктору сети готовым, так как в целях удобства распараллеливания и единообразия клиентского кода сеть сама должна принимать решение о создании

тех или иных автоматов. Помимо этого, класс фабрики сети выполняет еще одну роль, о которой подробнее скажем чуть позже. Сейчас лишь оговоримся, что этот класс должен быть «легким», т.е. как таковое создание объекта этого класса не должно требовать больших затрат ресурсов.

Класс `fsmnet_type` содержит описание вложенного класса `shared_area_type`, инкапсулирующего выделение и использование общей области памяти автоматной сети для промежуточного хранения значений выходных сигналов автоматов. Выделение области памяти сопровождается также созданием списков индексов для чтения входных данных и для записи выходных данных каждым автоматом. Все данные для работы с общей областью памяти создаются в конструкторе класса `shared_area_type` на основе списка связей, полученного от фабрики сети.

Список связей формируется в фабрике сети клиентским кодом с помощью вспомогательного класса `links_type`, который введен для простоты добавления связей.

Для создания автоматной сети в клиентском коде объявляется набор классов автоматов, унаследованных от `fsm_type`, а также класс фабрики сети, унаследованный от `factory_abstract_type`. Во время выполнения клиентским кодом создается объект фабрики сети, который должен существовать на протяжении всего ее жизненного цикла, после чего он передается конструктору объекта `fsmnet_type`. Объекты конкретных автоматов из объявленного набора классов создаются и уничтожаются в рамках выполнения функций фабрики сети `create_fsm` и `destroy_fsm` соответственно. Этим функциям передается номер автомата из интервала от 0 до $N-1$, где $N$ — количество автоматов в сети, возвращаемое функцией фабрики `number_fsm`.

К примеру, объявление автоматной сети, изображенной на рис. 3.2, может выглядеть следующим образом:

```
// класс автомата A
class fsm_a_type: public fsm_type
{
public:
 enum { INPUT0, INPUT1, INPUT2, INPUT3, INPUT_NUMBER };
 enum { OUTPUT0, OUTPUT1, OUTPUT2, OUTPUT_NUMBER };
 enum { STATE0, STATE1 /* ... прочие состояния */ };
private:
 state_type do_something(state_type state)
 {
  if (state == STATE_OFF && m_input[INPUT0] == /* ... */)
  {
   state = STATE0;
   m_output[OUTPUT0] = /* ... */;
   // ... заполнение остальных выходов
  };
  // ... обработка прочих условий
  return state;
 }
public:
 fsm_a_type(void):
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&fsm_a_type::do_something));
  // ... прочие обработчики
 }
};
```

```
// класс автомата B
class fsm_b_type: public fsm_type
{
 // ...
};

// класс автомата C
class fsm_c_type: public fsm_type
{
 // ...
};

// сеть из трех автоматов
class fsmnet_abc_type: public fsmnet_type
{
public:
 enum { FSM_A, FSM_B, FSM_C, FSM_NUMBER };
 enum { INPUT0, INPUT1, INPUT_NUMBER };
 enum { OUTPUT0, OUTPUT1, OUTPUT2, OUTPUT3, OUTPUT_NUMBER };

 // фабрика сети
 class factory_type: public factory_abstract_type
 {
 public:
  int number_fsm(void) const { return FSM_NUMBER; }
  int number_input(void) const { return INPUT_NUMBER; }
  int number_output(void) const { return OUTPUT_NUMBER; }
  links_type links(void) const
  {
   links_type links;
   // связи входов сети с входами автоматов
   links.input_to_fsm(INPUT0, FSM_A, fsm_a_type::INPUT0);
   links.input_to_fsm(INPUT1, FSM_A, fsm_a_type::INPUT1);
   links.input_to_fsm(INPUT0, FSM_B, fsm_b_type::INPUT1);
   links.input_to_fsm(INPUT1, FSM_B, fsm_b_type::INPUT0);
   // связи автоматов между собой
   links.fsm_to_fsm(FSM_A, fsm_a_type::OUTPUT0, FSM_C, fsm_c_type::INPUT2);
   links.fsm_to_fsm(FSM_A, fsm_a_type::OUTPUT1, FSM_B, fsm_b_type::INPUT3);
   links.fsm_to_fsm(FSM_A, fsm_a_type::OUTPUT1, FSM_C, fsm_c_type::INPUT0);
   links.fsm_to_fsm(FSM_A, fsm_a_type::OUTPUT2, FSM_B, fsm_b_type::INPUT2);
   links.fsm_to_fsm(FSM_B, fsm_b_type::OUTPUT0, FSM_C, fsm_c_type::INPUT3);
   links.fsm_to_fsm(FSM_B, fsm_b_type::OUTPUT1, FSM_A, fsm_a_type::INPUT3);
   links.fsm_to_fsm(FSM_B, fsm_b_type::OUTPUT1, FSM_C, fsm_c_type::INPUT1);
   links.fsm_to_fsm(FSM_B, fsm_b_type::OUTPUT2, FSM_A, fsm_a_type::INPUT2);
   // связи выходов автоматов с выходами сети
   links.fsm_to_output(FSM_A, fsm_a_type::OUTPUT0, OUTPUT0);
   links.fsm_to_output(FSM_C, fsm_c_type::OUTPUT0, OUTPUT1);
   links.fsm_to_output(FSM_C, fsm_c_type::OUTPUT1, OUTPUT2);
   links.fsm_to_output(FSM_B, fsm_b_type::OUTPUT0, OUTPUT3);
   return links;
  }
  fsm_abstract_type *create_fsm(int id)
  {
   if (id == FSM_A)
    return new fsm_a_type();
```

```
    else if (id == FSM_B)
     return new fsm_b_type();
    else
     return new fsm_c_type();
   }
   void destroy_fsm(int id, fsm_abstract_type *pfsm)
   {
    if (id == FSM_A)
     delete dynamic_cast<fsm_a_type *>(pfsm);
    else if (id == FSM_B)
     delete dynamic_cast<fsm_b_type *>(pfsm);
    else
     delete dynamic_cast<fsm_c_type *>(pfsm);
   }
 };

 fsmnet_abc_type(factory_type &factory):
  fsmnet_type(factory)
 {}
};
```

Работа такой сети должна выполняться внешним циклом так, как будто она является обычным конечным автоматом. В частности, она может быть реализована следующим образом:

```
// цикл работы конечного автомата
// на входе: автомат, интервал наборов входных сигналов
// и начало интервала наборов выходных сигналов
template <typename InIt, typename OutIt>
OutIt run_fsm(fsm_abstract_type &fsm, InIt ib, InIt ie, OutIt ob)
{
 // цикл работы автомата
 do
 {
  // подать входные сигналы, если есть, или обнуленные
  fsm.put_input((ib != ie) ?
   *ib++ :
   fsm_type::signals_type(fsm.number_input(), 0));
  // выполнить такт работы автомата
  fsm.do_work();
  // получить очередные выходные сигналы
  *ob++ = fsm.get_output();
  // проверить, выключен ли автомат
 } while (!fsm.is_off());
 return ob;
}

// ...

// контейнер наборов входных сигналов
vector<fsm_type::signals_type> allins;
// ... инициализация входных сигналов

// контейнер всех наборов выходных сигналов
vector<fsm_type::signals_type> allouts;
```

```
// фабрика автоматной сети и сама сеть
fsmnet_abc_type::factory_type factory;
fsmnet_abc_type fsmnet(factory);

// выполнение автоматной сети
run_fsm(fsmnet,
 allins.begin(), allins.end(),
 back_inserter(allouts));
// по завершении работы сети allouts содержит все наборы
// выходных сигналов сети, включая последний (обнуленный)
```

В приведенном примере на вход сети подается последовательность векторов входных сигналов `allins`. После выполнения цикла последовательность `allouts` содержит по порядку все векторы выходных сигналов, включая вектор последнего такта. Значения последнего вектора как результат работы автомата рассматриваться не должны, поскольку после последнего такта сеть пребывает в выключенном состоянии, следовательно, полезные сигналы в ней, в т.ч. выходные, отсутствуют.

### 3.3.2. Простая реализация с использованием MPI

Приведенный в приложении В набор базовых классов сделан, по возможности, универсальным с точки зрения инкапсуляции создания и работы автоматов для того, чтобы использующий эти классы клиентский код подвергался минимальным изменениям при смене инструмента распараллеливания. Как видно, этот код был выполнен последовательным, после чего выполнение тактов автоматов на каждом такте работы сети было распараллелено с помощью директив OpenMP. Теперь рассмотрим, как приведенный набор классов может быть распараллелен с использованием интерфейса MPI для выполнения в системах с распределенной памятью. Для краткости, мы приведем лишь те фрагменты кода, которые подверглись изменению относительно приведенного в приложении В.

Для начала рассмотрим простейший способ реализации, требующий минимальных изменений клиентского кода, но который, однако, будет работать только в случае отсутствия вложенных сетей, т.е. когда вся сеть состоит лишь из элементарных автоматов.

Предположим, что количество процессов в группе коммуникатора `MPI_COMM_WORLD` соответствует количеству автоматов в сети, и на каждый автомат приходится свой процесс. Тогда в каждом процессе может быть создан объект сети, содержащий один автомат. В этой ситуации объекту сети в каждом процессе, прежде всего, необходимо знать номер конкретного автомата, им реализуемого. Этот номер хранится в переменной-члене класса `fsmnet_type`:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
private:
 // номер автомата текущей сети, реализуемого нашим процессом
 int m_fsmid;
 // ...
};
```

Присвоение номера автомата производится путем получения ранга текущего процесса в коммуникаторе `MPI_COMM_WORLD` в теле конструктора `fsmnet_type`. Там же выполняется создание одного автомата с текущим номером. Деструктор также меняется, поскольку выполняется уничтожение лишь одного автомата:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
public:
 fsmnet_type(factory_abstract_type &factory):
  m_factory(factory), m_shared(m_factory.links().get())
 {
  int size;
  MPI_Comm_rank(MPI_COMM_WORLD, &m_fsmid);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  assert(size == m_factory.number_fsm());
  // создание автомата
  m_pfsm.push_back(m_factory.create_fsm(m_fsmid));
 }

 ~fsmnet_type(void)
 {
  // уничтожение автомата
  m_factory.destroy_fsm(m_fsmid, m_pfsm.back());
 }
 // ...
};
```

Поскольку автомат в текущем процессе один, изменяется код функции `do_work`, выполняющей работу сети на одном такте. Эта работа сокращается до соответствующего последовательного вызова трех функций автомата:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 void do_work(void)
 {
  // прочитать входные данные, выполнить действия
  m_pfsm.back()->put_input(m_shared.get_input(m_fsmid));
  m_pfsm.back()->do_work();
  // записать выходные (запись должна быть отделена от чтения)
  m_shared.put_output(m_fsmid, m_pfsm.back()->get_output());
 }
 // ...
};
```

Наконец, поскольку автоматы находятся в разных процессах, в функцию проверки выключения сети вносится вызов редукционной операции логического «И»:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 bool is_off(void) const
 {
  // сеть в выключенном состоянии, когда все автоматы выключены
  int isoff = m_pfsm.back()->is_off();
  int allisoff;
  MPI_Allreduce(&isoff, &allisoff, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);
  return allisoff;
 }
 // ...
```

```
};
```

Перечисленные изменения коснулись самого класса сети `fsmnet_type`. Однако для взаимодействия автоматов сети между собой также требуется изменение вложенного класса `shared_area_type`. Поскольку после записи выходных данных каждого автомата требуется их рассылка остальным процессам, в конец функции `put_output` добавлен вызов `MPI_Allgatherv`, выполняющий сбор областей выходных данных разных размеров от всех автоматов в одну область и копирование полученного результата в адресное пространство всех процессов коммуникатора `MPI_COMM_WORLD`. Помимо этого, для предотвращения возможности записи выходов до выполнения чтения входов в начало функции добавлена барьерная синхронизация всех процессов коммуникатора:

```
class shared_area_type
{
 // ...
public:
 // сохранение выходных данных конкретного автомата
 void put_output(int i, const signals_type &output)
 {
  // синхронизация всех автоматов (для отделения от чтения входов)
  MPI_Barrier(MPI_COMM_WORLD);
  assert(size_t(m_outsize[i]) == output.size());
  std::copy(
   output.begin(), output.end(),
   m_data.begin() + m_outpos[i]);
  // выполним полный обмен - раскидаем все выходы всем процессам
  // в конце массивов размеров и смещений есть по одному
  // лишнему элементу, но это не мешает - они не учитываются
  MPI_Allgatherv(
   &m_data[m_outpos[i]], m_outsize[i], MPI_INT,
   &m_data[0], &m_outsize.front(), &m_outpos.front(), MPI_INT,
   MPI_COMM_WORLD);
 }
 // ...
};
```

Модифицированный таким образом набор классов потребует минимальных изменений в использующем их клиентском коде. Необходимой из них является лишь вставка вызовов `MPI_Init` и `MPI_Finalize`.

### 3.3.3. Реализация с поддержкой вложенных сетей

Описанная только что модификация базовых классов с использованием MPI, как было оговорено выше, позволяет создавать лишь одноуровневые автоматные сети, состоящие из элементарных автоматов (которые не являются вложенными автоматными сетями). При реализации на ее основе большой сети могут возникнуть неоправданно высокие накладные расходы на зачастую ненужную передачу выходных данных между всеми процессами. Реализация же полноценного параллелизма на основе MPI для многоуровневых автоматных сетей требует внесения небольших изменений в используемый описанными классами программный интерфейс. Как и в предыдущем случае, будем приводить лишь подвергнувшиеся изменению фрагменты кода из приложения В.

Необходимость изменения предоставляемого классами интерфейса диктуется следующими обстоятельствами. Каждой автоматной сети для выполнения обмена данными меж-

ду ее внутренними автоматами должен быть сопоставлен свой коммуникатор. Поскольку рассматриваемая версия MPI предполагает статическое существование групп процессов (отсутствие возможности изменения группы после создания), перед созданием коммуникатора необходимо знать, сколько процессов он должен содержать. По этой причине к интерфейсу фабрики сети добавляется функция `size_fsm`:

```
// абстрактный тип фабрики автоматной сети
class factory_abstract_type
{
public:
 // ...
 // полное количество параллелей в автомате
 virtual
 int size_fsm(int id) const = 0;
 // ...
 // создание очередного автомата в сети
 virtual
 fsm_abstract_type *create_fsm(int id, MPI_Comm commfull) = 0;
 // ...
};
```

Дополнительный параметр в функции создания автомата нужен для передачи коммуникатора конструктору `fsmnet_type` или `fsm_type`, о чем подробнее будет сказано ниже. Новая функция `size_fsm` возвращает требуемое количество параллельных ресурсов для каждого автомата сети. Будем в дальнейшем для определенности называть это число размером автомата. Поскольку автомат может быть автоматной сетью, количество выполняющих его работу процессов может быть больше одного. Кроме того, любой автомат, даже не являясь сетью, может требовать более одного процесса. К примеру, у какого-либо автомата на одном такте работы может выполняться полноценный цикл работы некоторой другой, не связанной с данной, автоматной сети, или каким-либо другим образом организованная параллельная работа.

На первый взгляд, дополнительную функцию, возвращающую размер автомата, логичнее было бы внести не в интерфейс фабрики сети `factory_abstract_type`, а в интерфейс автомата `fsm_abstract_type`. Однако в таком случае для выяснения размера конкретного автомата потребовалось бы его создание, причем далеко не только в тех процессах, в которых он действительно должен быть создан для последующего функционирования.

Итак, информация о требуемом размере автомата должна быть известна фабрике сети. Поскольку ей изначально известно соответствие номеров конкретным автоматам, это не составляет проблему. При создании вложенной сети ее конструктору также должен быть передан объект фабрики вложенной сети. Он может содержаться внутри объекта фабрики объемлющей сети, и тогда информация о размере вложенной сети также будет доступна фабрике объемлющей. Таким образом, объект фабрики сети верхнего уровня удобно представляется «деревом» фабрик сети, т.е. содержит фабрики всех нижележащих сетей (рис. 3.3). Именно по этой причине, поскольку такой объект дерева фабрик должен содержаться в каждом процессе, все объекты фабрик должны быть «легкими» для создания. Существенные затраты вычислительных ресурсов допускаются лишь при вызове функций создания/уничтожения конкретного автомата.

На основе полученной от фабрики сети информации о размерах внутренних автоматов конструктор `fsmnet_type` выполняет создание необходимых коммуникаторов:

```
class fsmnet_type: public fsm_abstract_type
{
```

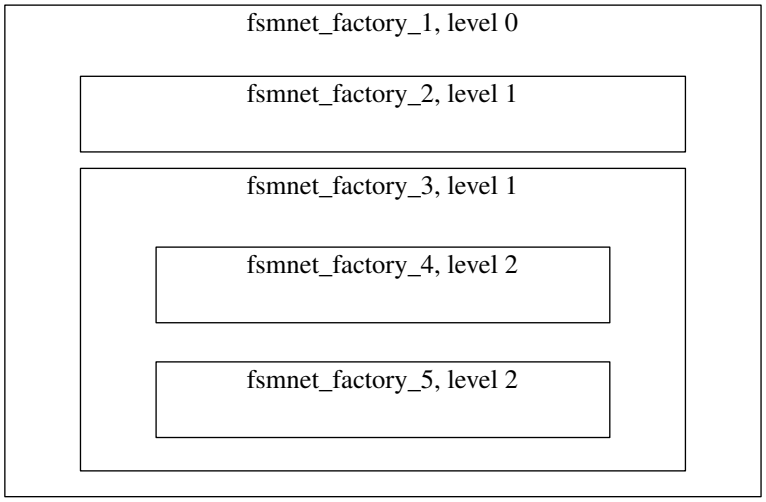



Рис. 3.3. Вложенность фабрик автоматных сетей

```
 // ...
private:
 // номер автомата текущей сети, реализуемого нашим процессом
 int m_fsmid;
 // полный коммуникатор по всем процессам сети
 MPI_Comm m_commfull;
 // коммуникатор локальной связи по сети автоматов
 MPI_Comm m_commlocal;
 // коммуникатор, переданный нижней сети/автомату
 MPI_Comm m_commlower;

public:
 fsmnet_type(factory_abstract_type &factory, MPI_Comm commfull):
  m_factory(factory), m_shared(m_factory.links().get(), *this),
  m_commfull(commfull)
 {
  // заполним массив локальных лидеров (нижележащих нулевых)
  std::vector<int> loc;
  // и локализуем свою принадлежность конкретному нижнему автомату
  int rank, size;
  MPI_Comm_rank(m_commfull, &rank);
  MPI_Comm_size(m_commfull, &size);
  m_fsmid = -1;
  for (int i = 0, cur = 0; i < m_factory.number_fsm(); ++i)
  {
   // положим индекс первого процесса очередного автомата
   loc.push_back(cur);
   // перейдем к следующему автомату
   cur += m_factory.size_fsm(i);
   assert(i + 1 < m_factory.number_fsm() || cur == size);
   // запомним индекс последнего автомата, если он наш
   if (m_fsmid < 0 && rank < cur)
    m_fsmid = i;
  };
  assert(m_fsmid != -1);
```

```
  // создадим коммуникатор локальных лидеров сети
  MPI_Group group, newgroup;
  MPI_Comm_group(m_commfull, &group);
  MPI_Group_incl(group, loc.size(), &loc.front(), &newgroup);
  MPI_Comm_create(m_commfull, newgroup, &m_commlocal);
  MPI_Group_free(&newgroup);
  MPI_Group_free(&group);

  // создадим нижний коммуникатор (для подсети или автомата)
  MPI_Comm_split(m_commfull, m_fsmid, rank, &m_commlower);

  // создадим нижний автомат или подсеть
  m_pfsm.push_back(m_factory.create_fsm(m_fsmid, m_commlower));
 }

 ~fsmnet_type(void)
 {
  // уничтожение автомата
  m_factory.destroy_fsm(m_fsmid, m_pfsm.back());
  // зачистка созданных коммуникаторов
  MPI_Comm_free(&m_commlower);
  if (m_commlocal != MPI_COMM_NULL)
   MPI_Comm_free(&m_commlocal);
 }
 // ...
};
```

К конструктору `fsmnet_type` добавляется дополнительный параметр, характеризующий коммуникатор, в рамках которого будут работать все нижележащие автоматы текущей сети. Конструктор сети самого верхнего уровня вызывается с каким-либо определенным вне сети коммуникатором с количеством процессов, равным полному размеру всей сети. К примеру, это может быть коммуникатор `MPI_COMM_WORLD` соответствующего размера.

Чтобы понять, какую работу в приведенном коде выполняет конструктор, следует оговорить, как происходит распределение автоматов по процессам. Тот коммуникатор текущей сети, в контексте которого будут обмениваться выходными данными ее абстрактные автоматы (элементарные автоматы или вложенные подсети), должен включать ровно столько процессов, сколько абстрактных автоматов она непосредственно содержит, т.е. сколько возвращается функцией `number_fsm` фабрики сети. В то же время, каждому абстрактному автомату должен быть передан свой коммуникатор, включающий количество процессов, равное размеру этого автомата, возвращаемому функцией `size_fsm`. С этой целью все процессы переданного коммуникатора `m_commfull` разделяются на группы заданных размеров, каждая из которых соответствует одному из абстрактных автоматов сети. Для выполнения этой операции путем опроса фабрики сети на предмет размеров ее автоматов все процессы предварительно вычисляют номер `m_fsmid` нижележащего абстрактного автомата текущей сети, который будет реализован в рамках данного процесса. После этого коммуникатор `m_commfull` разделяется на коммуникаторы `m_commlower` по значению номера текущего нижележащего автомата `m_fsmid`.

Из каждой созданной группы выделяется нулевой процесс, ответственный за «представительство» нижележащего автомата в текущей сети. Этот процесс (лидер) ответственен за передачу входных данных нижележащего абстрактного автомата всем его процессам и изъятие из него выходных. Все такие процессы текущей сети объединяются в группу так

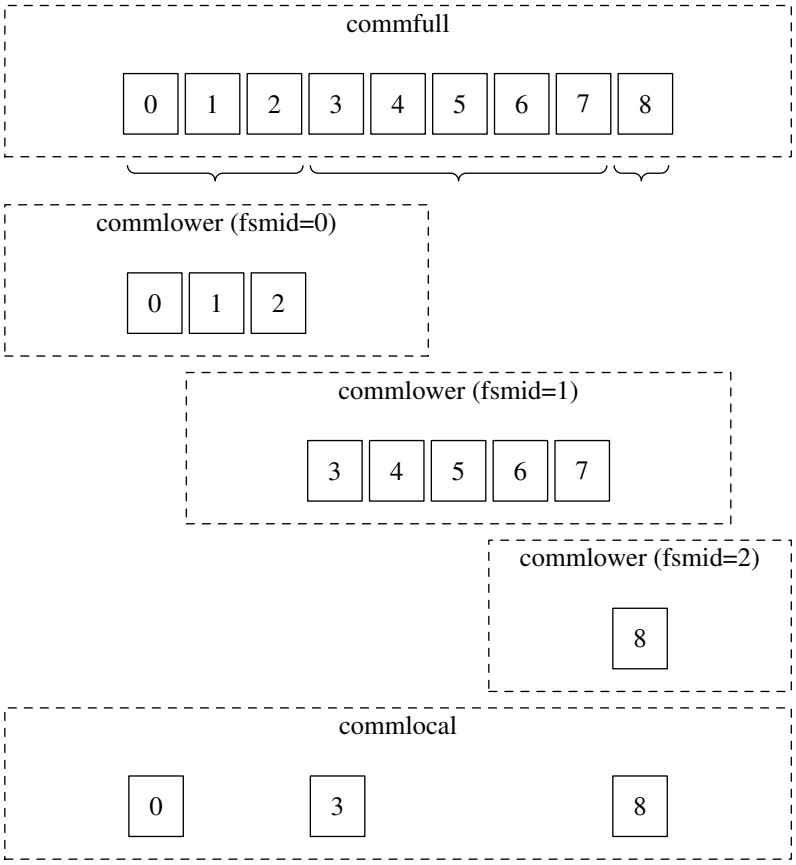



Рис. 3.4. Пример разбиения группы процессов по вложенным группам с выделением их лидеров

называемых локальных лидеров. Список номеров процессов локальных лидеров формируется в процессе опроса размеров автоматов. На основе группы локальных лидеров создается коммуникатор `m_commlocal`, в контексте которого будет производиться взаимодействие в рамках обмена выходными данными между абстрактными автоматами сети текущего уровня.

Пример распределения процессов по коммуникаторам можно увидеть на рис. 3.4. Здесь показано, как девять процессов распределяются на три автомата, размеры которых равны соответственно трем, пяти и одному. Коммуникатор `m_commfull` содержит все девять процессов, коммуникатор `m_commlocal` содержит три процесса (по количеству автоматов). Наконец, коммуникатор `m_commlower` содержит количество процессов, равное размеру соответствующей подсети или элементарного автомата. Всего таких коммуникаторов три (также по количеству автоматов). В группу процессов коммуникатора `m_commlocal` включаются нулевые процессы из всех коммуникаторов `m_commlower`.

В конце работы конструктора `fsmnet_type` вызывается создание нижележащего автомата с только что сформированным коммуникатором `m_commlower` в качестве параметра, который должен быть передан новому объекту класса `fsmnet_type` с целью последующего разбиения на коммуникаторы или объекту класса `fsm_type` с целью использования для внутреннего распараллеливания.

Деструктор `fsmnet_type` выполняет уничтожение созданного в конструкторе нижележащего автомата, после чего производит освобождение созданных там же коммуникаторов.

Коммуникатор `m_commlocal` освобождают только те процессы, для которых он действителен, т.е. только те, которые являются лидерами нижележащих автоматов (имеют нулевой ранг в группе коммуникатора `m_commlower`).

Функция `do_work` класса `fsmnet_type` полностью аналогична той, что была приведена выше для случая распараллеливания с использованием MPI работы одноуровневой автоматной сети. Функция же проверки выключения сети, в отличие от предыдущего случая, опрашивает на предмет выключения лишь локальных лидеров, причем результат получает лишь лидер текущей сети. Поскольку проверка выключения производится во всех процессах сети, полученный результат рассылается лидером в контексте коммуникатора `m_commfull`:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 bool is_off(void) const
 {
  // сеть в выключенном состоянии, когда все автоматы выключены
  int isoff = m_pfsm.back()->is_off();
  // сольем статусы от локальных лидеров в нулевой процесс
  int allisoff;
  if (m_commlocal != MPI_COMM_NULL)
   MPI_Reduce(&isoff, &allisoff, 1, MPI_INT, MPI_LAND, 0, m_commlocal);
  // и распределим результат между всеми
  // по построению нулевой в m_commfull тот же, что и в m_commlocal
  MPI_Bcast(&allisoff, 1, MPI_INT, 0, m_commfull);
  return allisoff;
 }
 // ...
};
```

Наконец, изменения должны коснуться функций помещения выходных данных автомата и входных данных сети в область общих данных сети. В обеих функциях используется коммуникатор локальных лидеров сети, который с этой целью должен быть каким-либо образом передан объекту класса `shared_area_type`:

```
class shared_area_type
{
 // ...
private:
 // ссылка на сеть-владельца
 fsmnet_type &m_fsmnet;

public:
 // ...
 shared_area_type(const linklist_type &linklist, fsmnet_type &fsmnet):
  m_fsmnet(fsmnet)
 {
  // ...
 }
 // сохранение выходных данных конкретного автомата
 void put_output(int i, const signals_type &output)
 {
  // рассмотрению подлежат только локальные лидеры
  if (m_fsmnet.m_commlocal != MPI_COMM_NULL)
  {
```

```
    // синхронизация всех автоматов (для отделения от чтения входов)
    MPI_Barrier(m_fsmnet.m_commlocal);
    assert(size_t(m_outsize[i]) == output.size());
    std::copy(
     output.begin(), output.end(),
     m_data.begin() + m_outpos[i]);
    // выполним полный обмен - раскидаем все выходы всем процессам
    // в конце массивов размеров и смещений есть по одному
    // лишнему элементу, но это не мешает - они не учитываются
    MPI_Allgatherv(
     &m_data[m_outpos[i]], m_outsize[i], MPI_INT,
     &m_data[0], &m_outsize.front(), &m_outpos.front(), MPI_INT,
     m_fsmnet.m_commlocal);
   };
  }
  // сохранение входных данных сети
  void put_input(const signals_type &input)
  {
   // рассмотрению подлежат только локальные лидеры
   if (m_fsmnet.m_commlocal != MPI_COMM_NULL)
   {
    // вход сети принимается только нулевым процессом в сети
    int rank;
    MPI_Comm_rank(m_fsmnet.m_commlocal, &rank);
    if (rank == 0)
    {
     assert(size_t(m_outsize.back()) == input.size());
     std::copy(
      input.begin(), input.end(),
      m_data.begin() + m_outpos.back());
    };
    // и рассылается остальным
    MPI_Bcast(
     &m_data[m_outpos.back()], m_outsize.back(), MPI_INT,
     0, m_fsmnet.m_commlocal);
   };
  }
  // ...
};
```

Обе функции выполняют полезную работу лишь в случае, когда текущий процесс является локальным лидером, т.е. лидером нижележащего автомата. В остальном функция `put_output` отличается от приведенной выше реализации для одноуровневой сети лишь использованным коммуникатором. Функция же `put_input` сохраняет переданные входные данные сети лишь в случае, когда текущий процесс является лидером текущей сети, т.е. нулевым процессом в рамках коммуникатора `m_commfull`. Этот же процесс по построению является нулевым для коммуникатора `m_commlocal`. После сохранения входных данных сети они рассылаются всем процессам локальных лидеров, которые распределят их далее в рамках предстоящего выполнения такта работы сети и вызова `put_input` для соответствующей подсети.

Конструктору `fsm_type` также передается коммуникатор, в рамках которого будет работать соответствующий автомат. В простом случае этот коммуникатор содержит лишь один процесс (эквивалентен `MPI_COMM_SELF`), однако в общем случае может содержать и больше. Объектом автомата, унаследованным от `fsm_type`, переданный коммуникатор может быть

использован по своему усмотрению:

```
class fsm_type: public fsm_abstract_type
{
 // ...
protected:
 // ...
 // коммуникатор автомата (может быть более одного процесса)
 MPI_Comm m_comm;

public:
 fsm_type(int inputnum, int outputnum, MPI_Comm comm):
  m_state(STATE_OFF),
  m_input(inputnum, 0), m_output(outputnum, 0),
  m_comm(comm)
 {}
 // ...
};
```

Приведенное описание модифицированных классов позволяет строить распределенные автоматные сети с многоуровневым вложением, а также, при необходимости, с независимым внутренним распараллеливанием элементарных автоматов.

## 3.4. Примеры сетей автоматов

Рассмотрим реализацию на основе описанных базовых классов решения двух задач. Обе эти задачи имеют довольно отвлеченный характер, но позволяют в общем понять принцип построения сетей автоматов и схему организации их взаимодействия.

### 3.4.1. Параллельный сумматор

Данный пример дает мало преимуществ при решении реальной задачи суммирования, однако хорошо подходит для иллюстрации возможностей построения программных автоматных сетей. При описании параллельных алгоритмов часто приводится алгоритм сдваивания — алгоритм параллельного суммирования последовательности чисел некоторой длины $n$. Суть его заключается в том, что вся последовательность разбивается по парам, после чего все пары суммируются параллельно. Последовательность результатов снова разбивается попарно, цикл повторяется и т.д. Сумма элементов всей последовательности будет получена за количество шагов, равное $\lceil \log_2 n \rceil$ (при достаточном количестве параллельных ресурсов или в условиях неограниченного параллелизма).

При реализации алгоритма суммирования сдваиванием в виде автоматной сети мы будем учитывать реальное отсутствие неограниченного параллелизма и возложим выполнение операции суммирования на конечный набор параллельно работающих автоматов.

Реализуем автоматную сеть в виде, представленном на рис. 3.5. Сеть содержит $N$ автоматов, выполняющих непосредственно вычисление суммы двух чисел, и один автомат, управляющий вычислением суммы всей последовательности.

Каждый вычисляющий автомат (далее — вычислитель) получает на входе пару чисел, на выходе выдает результат их суммирования. Вычислитель пребывает либо в выключенном состоянии, либо в состоянии выдачи результата. В последнее состояние вычислитель переходит после получения на входе ненулевых аргументов и выполнения их суммирования.

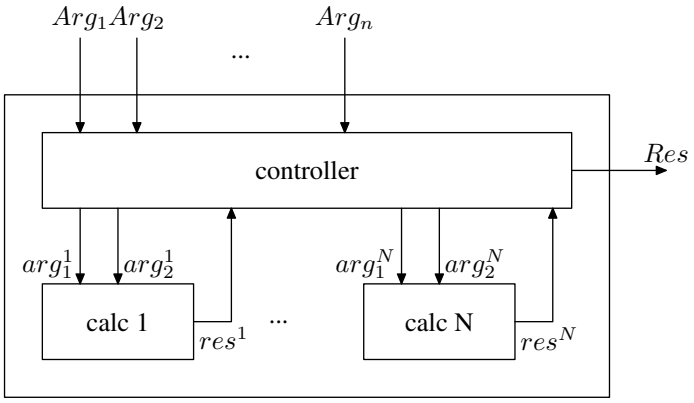



Рис. 3.5. Схема автоматной сети, реализующей сумматор

Управляющий автомат (далее — контроллер) в качестве входных сигналов на первом такте получает всю последовательность суммируемых чисел, на последующих тактах — результаты суммирования с вычислителей. В качестве выхода контроллер подает на каждый вычислитель очередную пару значений, а также в конце выдает на выход сети результат суммирования всей последовательности.

Принцип функционирования сети заключается в следующем. Сеть начинает работу при получении на входе последовательности с ненулевыми элементами. Все элементы попадают в некоторую очередь, хранимую в контроллере и распределяемую попарно контроллером на вычислители. Результаты суммирования с выходов вычислителей также помещаются контроллером в конец очереди. Здесь следует оговориться, что от наличия внутренней очереди автомат контроллера не перестает быть конечным, поскольку максимальная длина очереди ограничена (количество входов сети фиксировано).

На первом такте работы сети полезную работу выполняет только контроллер: принимает входную последовательность и передает пары на входы вычислителей. На втором такте вычислители производят суммирование, контроллер же готовит следующие пары. На этом такте выходы с вычислителей контроллером не учитываются, поскольку после первого такта они были нулевыми. На третьем такте работы контроллера результаты вычислений второго такта добавляются в конец очереди, и готовится третье распределение пар, пока вычислители обрабатывают второе. Когда длина внутренней очереди контроллера становится меньше удвоенного количества вычислителей, те вычислители, на которые не хватает работы, получают обнуленные аргументы и остаются в выключенном состоянии.

Состояние контроллера характеризует количество работающих в текущий момент вычислителей, т.е. количество распределенных в последний раз пар слагаемых. Здесь снова оговоримся, что формально очередь слагаемых также является частью состояния, однако для наглядности программы мы можем отделить одно от другого и использовать переменную состояния для более информативных целей. В начале работы обычное состояние контроллера — «N» (при $n \gg N$). Когда процесс близится к завершению вычислений, состояние меняется на меньшее «N», в соответствии с текущим количеством распределенных пар. По достижении состояния «0», когда в очереди контроллера остался лишь один элемент, состояние меняется на «Вывод результата». На выходе при этом выдается единственный элемент очереди. На следующем такте выключается контроллер и вся сеть.

Класс вычислителя `calc_type` определяет один обработчик на оба состояния:

```
class calc_type: public fsm_type
{
```

```
public:
 enum { INPUT_ARG1, INPUT_ARG2, INPUT_NUMBER };
 enum { OUTPUT_RESULT, OUTPUT_NUMBER };
 enum { STATE_OUT };

private:
 state_type do_calc(state_type state)
 {
  if (m_input[INPUT_ARG1] || m_input[INPUT_ARG2])
  {
   state = STATE_OUT;
   m_output[OUTPUT_RESULT] = m_input[INPUT_ARG1] + m_input[INPUT_ARG2];
  }
  else
   state = STATE_OFF;
  return state;
 }
public:
 calc_type(void):
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&calc_type::do_calc));
  add_handler(STATE_OUT, handler_type(&calc_type::do_calc));
 }
};
```

В случае наличия на входе ненулевых слагаемых, вычислитель производит суммирование и вывод результата. В противном случае переходит в выключенное состояние (или остается в нем), при этом выход его автоматически обнуляется.

Класс контроллера `ctrl_type` определяет три обработчика состояний:

```
class ctrl_type: public fsm_type
{
public:
 enum { OUTPUT_RESULT, OUTPUT_ARG1_0, OUTPUT_ARG2_0 };
 enum { STATE_OUT, STATE_CALC_NUMBER_0 };

private:
 deque<signal_type> m_queue;
 int m_argnum; // количество аргументов сумматора
 int m_calcnum; // количество вычислителей

 // распределить пары данных, сколько есть в наличии, на вычислители
 int dispatch(void)
 {
  // количество задействуемых вычислителей - половина размера очереди
  int num = m_queue.size() / 2;
  // но не более общего количества вычислителей
  num = (num > m_calcnum) ? m_calcnum : num;
  // переносим пары аргументов на входы пит вычислителей
  // остальные вычислители, если есть, не задействуем
  fill(
   copy(
    m_queue.begin(), m_queue.begin() + num * 2,
    m_output.begin() + OUTPUT_ARG1_0),
   m_output.end(), 0);
```

```
  m_queue.erase(m_queue.begin(), m_queue.begin() + num * 2);
  return num;
 }

 state_type do_start(state_type state)
 {
  // сложим все ненулевые входы в очередь
  remove_copy(
   m_input.begin(), m_input.begin() + m_argnum,
   back_inserter(m_queue), 0);
  // если таковые есть, переключимся на обработку
  if (!m_queue.empty())
  {
   state = STATE_CALC_NUMBER_0 + dispatch();
   m_output[OUTPUT_RESULT] = 0;
  }
  else
   // иначе продолжим быть выключенными
   state = STATE_OFF;
  return state;
 }

 state_type do_calc(state_type state)
 {
  // сложить все ненулевые результаты в конец очереди
  remove_copy(
   m_input.begin() + m_argnum, m_input.end(),
   back_inserter(m_queue), 0);
  // выход, если активно 0 вычислителей и в очереди один элемент
  if (state == STATE_CALC_NUMBER_0 && m_queue.size() == 1)
  {
   state = STATE_OUT;
   m_output[OUTPUT_RESULT] = m_queue.front();
   m_queue.pop_front();
  }
  else
  {
   // распределить пары из начала очереди
   state = STATE_CALC_NUMBER_0 + dispatch();
   m_output[OUTPUT_RESULT] = 0;
  };
  return state;
 }

 state_type do_stop(state_type state)
 {
  // перейдем в выключенное состояние
  return STATE_OFF;
 }

public:
 ctrl_type(int argnum, int calcnum):
  fsm_type(
   argnum + calcnum,
   OUTPUT_ARG1_0 + calcnum * calc_type::INPUT_NUMBER),
```

```
  m_argnum(argnum), m_calcnum(calcnum)
 {
  add_handler(STATE_OFF, handler_type(&ctrl_type::do_start));
  for (int i = 0; i <= calcnum; ++i)
   add_handler(STATE_CALC_NUMBER_0 + i, handler_type(&ctrl_type::do_calc));
  add_handler(STATE_OUT, handler_type(&ctrl_type::do_stop));
 }
};
```

Обработчик выключенного состояния `do_start` проверяет наличие на входе контроллера последовательности ненулевых чисел, помещает их, если таковые есть, в очередь `m_queue` и распределяет по вычислителям с помощью функции `dispatch`. Функция `dispatch` возвращает количество задействованных вычислителей, на основе которого задается следующее состояние.

Для всех состояний от «0» до «N» включительно определен обработчик `do_calc`. В начале функции `do_calc` осуществляется сбор результатов вычислений предыдущего такта со всех вычислителей с помощью алгоритма `std::remove_copy`. Его название может несколько сбивать с толку, но в самом деле он просто копирует элементы последовательности, за исключением копий заданного значения (нуля в нашем случае). Далее выполняется проверка факта завершения вычислений, который определяется наличием в очереди лишь одного элемента и отсутствием активных вычислителей. В случае если какое-либо из этих условий не выполняется, производится попытка следующего распределения. Если же вычисления закончены, происходит переход к состоянию «Вывод результата». Его обработчик `do_stop` просто переводит автомат в выключенное состояние.

Автоматная сеть представляется классом `summator_type`, который объявляет константы для обозначения автоматов и выходов сети, а также класс фабрики сети, создающий автоматы и формирующий набор связей между ними в соответствии с рис. 3.5:

```
class summator_type: public fsmnet_type
{
public:
 enum { FSM_CONTROL, FSM_CALCULATOR_0 };
 enum { OUTPUT_RESULT, OUTPUT_NUMBER };

 class factory_type: public factory_abstract_type
 {
 private:
  int m_inputnum, m_outputnum, m_fsmnum;

 public:
  factory_type(int inputnum, int outputnum, int fsmnum):
   m_inputnum(inputnum), m_outputnum(outputnum), m_fsmnum(fsmnum)
  {}
  int number_fsm(void) const { return m_fsmnum; }
  int number_input(void) const { return m_inputnum; }
  int number_output(void) const { return m_outputnum; }
  links_type links(void) const
  {
   links_type links;
   // входы сети прицепим к контроллеру
   for (int i = 0; i < m_inputnum; ++i)
    links.input_to_fsm(i, FSM_CONTROL, i);
   // выходы сети - первые два выхода от контроллера
   for (int i = 0; i < m_outputnum; ++i)
```

```
    links.fsm_to_output(FSM_CONTROL, i, i);
   // все вычислители к контроллеру подцепим
   for (int i = 0; i < m_fsmnum - FSM_CALCULATOR_0; ++i)
   {
    // аргументы вычислителя от контроллера (после выходов сети)
    for (int j = 0; j < calc_type::INPUT_NUMBER; ++j)
     links.fsm_to_fsm(
      FSM_CONTROL, m_outputnum + calc_type::INPUT_NUMBER * i + j,
      FSM_CALCULATOR_0 + i, j);
    // выход вычислителя - к контроллеру, после входов сети
    links.fsm_to_fsm(
     FSM_CALCULATOR_0 + i, calc_type::OUTPUT_RESULT,
     FSM_CONTROL, m_inputnum + i);
   };
   return links;
  }
  fsm_abstract_type *create_fsm(int id)
  {
   fsm_abstract_type *pfsm;
   if (id == FSM_CONTROL)
    pfsm = new ctrl_type(m_inputnum, m_fsmnum - 1);
   else
    pfsm = new calc_type();
   return pfsm;
  }
  void destroy_fsm(int id, fsm_abstract_type *pfsm)
  {
   if (id == FSM_CONTROL)
    delete dynamic_cast<ctrl_type *>(pfsm);
   else
    delete dynamic_cast<calc_type *>(pfsm);
  }
 };

public:
 summator_type(factory_type &factory):
  fsmnet_type(factory)
 {}
};
```

Приведенный набор автоматов может быть использован следующим кодом:

```
// вектор входных сигналов сети
fsm_type::signals_type ins;
// ... инициализация вектора входных сигналов

// количество автоматов: контроллер+вычислители
int fsmnum = 1 + DEF_CALCNUM;
// фабрика сети и сеть
summator_type::factory_type factory(
 ins.size(), summator_type::OUTPUT_NUMBER, fsmnum);
summator_type fsmnet(factory);

// результат суммирования
fsm_type::signal_type sum = 0;
// передача входных сигналов
```

```
fsmnet.put_input(ins);
for (;;)
{
 // такт работы сети
 fsmnet.do_work();
 // проверка выключения сети
 if (fsmnet.is_off())
  break;
 // чтение выходного сигнала
 sum = fsmnet.get_output()[summator_type::OUTPUT_RESULT];
};
```

Количество входов сети определяется на основе длины вектора входных значений. После передачи входной последовательности в сеть выполняется полный цикл работы сети. Сохранение выходного сигнала в теле цикла осуществляется на всех тактах, кроме последнего (когда сеть уже выключена).

### 3.4.2. Прямоугольный бильярд

Следующий пример описывает реализацию не настольной игры, а, скорее, некоторого сильно упрощенного варианта математического бильярда, хотя до этого понятия ему также довольно далеко (слишком прост механизм движения шаров). С другой стороны, эта задача в таком упрощении весьма наглядна в автоматной реализации. Более того, при попытке реализации ее обычным путем, так или иначе, в большинстве случаев будет получен алгоритм, близкий к автоматному.

Понятие математического бильярда предполагает движение точечного шара без трения по столу произвольной формы без луз в условиях абсолютно упругого отражения от границ. Эта, казалось бы, простая задача порождает немало вопросов, связанных с возможными траекториями шара, таких как, к примеру, критерий существования периодических траекторий на столе заданной формы. В некоторых случаях рассматривают системы из нескольких шаров, сталкивающихся в том числе друг с другом (модель газа).

Мы реализуем дискретное движение шаров по некоторому столу и соответствующее отображение его состояния. Шары могут отталкиваться от границ стола и сталкиваться друг с другом, при этом направление их движения меняется соответственно. Информация о границах стола считается известной всем шарам. Помимо нее, шару для принятия решения о направлении дальнейшего движения на каждом шаге необходима информация о своем текущем положении и направлении движения, а также о направлении и координатах других шаров. Дальнейшее направление своего движения каждый шар вычисляет автономно, и именно здесь возникает логический параллелизм задачи, который мы используем, реализуя ее в виде автоматной сети.

Поскольку в наши цели входит лишь иллюстрация использования автоматов и сетей, мы снимем учет множества факторов, чем сильно упростим задачу и сделаем ее реализацию более наглядной. Нами будет реализован случай прямоугольного стола с двумя шарами. Будем считать, что шары двигаются по фиксированным клеткам стола под углом 45 градусов к его границам в одном из четырех возможных в этой ситуации направлений. При столкновении с границей стола каждый шар меняет направление естественным образом. При попадании в угол стола шар меняет направление на противоположное (это допустимо для частного случая прямоугольного бильярда). При столкновении шаров между собой оба продолжают движение в противоположных направлениях (рис. 3.6). Для простоты мы учитываем лишь встречное столкновение шаров. Очевидно, такое упрощение не отвечает некоторым естественным законам (к примеру, не учитывается боковое столкновение). Од-

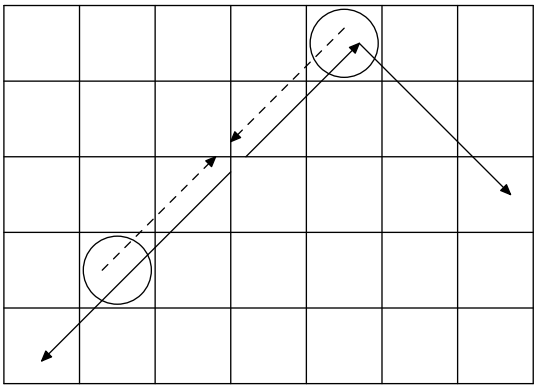


Рис. 3.6. Схема движения шаров

нако, поскольку поведение шара инкапсулируется внутри соответствующего автомата, при желании оно может быть легко доработано. Также путем усложнения лишь этого автомата может быть изменена логика движения шаров (количество возможных направлений движения, скорость и т.п.)

В нашей конечноавтоматной реализации мы в любом случае получаем периодическую последовательность состояний системы. Действительно, всего возможных состояний системы в нашем случае может быть не более, чем квадрат количества возможных состояний одного шара, которое, в свою очередь, равно произведению количества клеток на столе на количество возможных направлений движения шара. Поскольку количество возможных состояний системы конечно, рано или поздно она придет в какое-либо состояние, в котором уже была ранее. Начиная с этого момента повторится вся дальнейшая последовательность состояний системы, поскольку каждое последующее состояние однозначно определяется текущим. В этой ситуации естественно возникает вопрос о длине периода повторения, задачу замера которого мы возложим на реализуемую автоматную сеть.

В общем случае последовательность состояний системы может стать периодической не сразу, а лишь после прошествия некоторой подпоследовательности состояний, в которых система больше не окажется. Так бывает в случаях, когда последующее состояние определяется текущим однозначно, но не взаимно однозначно, т.е. когда для некоторых состояний возможно более одного предшествующего. В то же время, природа задачи подсказывает, что преобразование состояния должно быть взаимно однозначным (поскольку в идеале движение шаров в противоположном направлении должно привести к исходному состоянию системы). Таким образом, мы можем задать преобразование состояний системы взаимно однозначным и считать последовательность периодической сразу. Это даст нам возможность замерить период, начиная с первого шага. Остается лишь добавить условие отсутствия у шаров препятствий движению на первом шаге. В противном случае может возникнуть отсутствие взаимной однозначности в начале последовательности, поскольку движение на втором шаге, к примеру, шара от границы может оказаться следствием изначально заданного движения как от границы, так и, наоборот, к ней.

Реализуемая нами автоматная сеть содержит три автомата (рис. 3.7). Один из них играет роль стола. В его задачи входит получение данных о текущих состояниях шаров и замер периода состояния системы, а также отображение какими-либо средствами текущего положения шаров. На выходе стол выдает сигнал включения для шаров, а также информацию о периоде. Еще два автомата играют роль шаров, каждый из которых принимает инициализационные данные о своих координатах и направлении движения (на первом такте) и

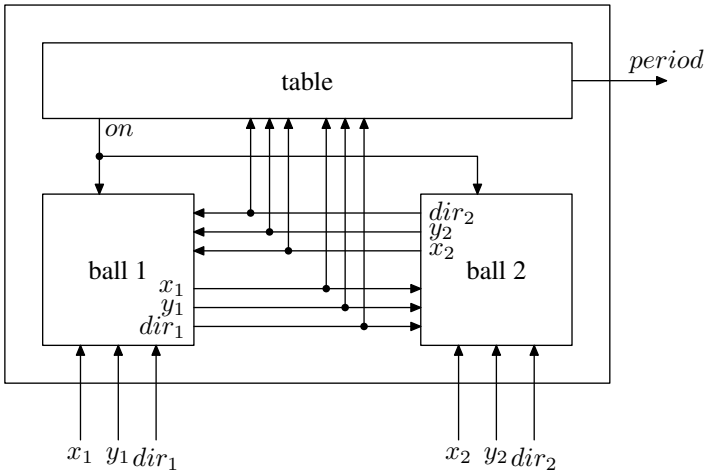



Рис. 3.7. Автоматная сеть, реализующая движение шаров

текущие координаты и направление другого шара (на остальных тактах).

Автомат, отвечающий за поведение стола, по сравнению с автоматом шара, реализуется довольно просто:

```
class table_type: public fsm_type
{
public:
 enum {
  INPUT_X1,
  INPUT_Y1,
  INPUT_DIR1,
  INPUT_X2,
  INPUT_Y2,
  INPUT_DIR2,
  INPUT_NUMBER
 };
 enum { OUTPUT_ON, OUTPUT_PERIOD, OUTPUT_NUMBER };
 enum { STATE_ON, STATE_OUT };

private:
 // размеры стола
 const int m_xsize, m_ysize;
 // количество выполненных шагов движения
 signal_type m_steps;
 // начальное положение и направление движения шаров
 signals_type m_stored;

 state_type do_start(state_type state)
 {
  m_output[OUTPUT_ON] = 1;
  return STATE_ON;
 }

 state_type do_step(state_type state)
 {
  // выполняем какое-либо отображение стола и шаров
  // ...
```

```
  // запоминаем положение шаров (на первом шаге)
  if (m_steps == 0)
   m_stored = m_input;

  // на остальных проверяем совпадение
  if (m_steps > 0 && m_stored == m_input)
  {
   // если совпало, завершаем цикл работы
   state = STATE_OUT;
   m_output[OUTPUT_ON] = 0;
   m_output[OUTPUT_PERIOD] = m_steps;
  }
  else
  {
   // иначе переходим к следующему шагу
   state = STATE_ON;
   m_output[OUTPUT_ON] = 1;
   ++m_steps;
  };
  return state;
 }

 state_type do_stop(state_type state)
 {
  m_steps = 0;
  m_stored.clear();
  return STATE_OFF;
 }

public:
 table_type(int xsize, int ysize):
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER),
  m_xsize(xsize), m_ysize(ysize), m_steps(0)
 {
  add_handler(STATE_OFF, handler_type(&table_type::do_start));
  add_handler(STATE_ON, handler_type(&table_type::do_step));
  add_handler(STATE_OUT, handler_type(&table_type::do_stop));
 }
};
```

Приведенный автомат может пребывать в трех состояниях: выключенном, рабочем и состоянии вывода. В конструкторе класса `table_type` перечисленным состояниям назначаются соответственно обработчики `do_start`, `do_step` и `do_stop`.

В выключенном состоянии счетчик выполненных шагов обнулен. Функция `do_start` передает шарам сигнал включения. Функция `do_step` в первый вызов (при нулевом счетчике) запоминает текущее состояние шаров, в остальных сверяет текущее состояние шаров с сохраненным. В случае совпадения выводится значение периода (значение счетчика шагов) и работа сети завершается. В противном случае наращивается счетчик шагов и работа продолжается.

Далее приведен код класса автомата, реализующего простое поведение шара, описанное выше:

```
class ball_type: public fsm_type
{
```

```
public:
 enum {
  INPUT_ON,
  INPUT_INIT_X,
  INPUT_INIT_Y,
  INPUT_INIT_DIR,
  INPUT_OTHER_X,
  INPUT_OTHER_Y,
  INPUT_OTHER_DIR,
  INPUT_NUMBER
 };
 enum { OUTPUT_X, OUTPUT_Y, OUTPUT_DIR, OUTPUT_NUMBER };
 // направления движения шара
 enum { DIR_NW, DIR_NE, DIR_SW, DIR_SE, DIR_NUMBER };

private:
 // размеры стола
 const int m_xsize, m_ysize;
 // текущие координаты шара
 typedef pair<int, int> coords_type;
 coords_type m_coords;

 // сдвиг координат на одну клетку в заданном направлении
 coords_type update(state_type dir, const coords_type &coords)
 {
  int dx, dy;
  dx = (dir == DIR_NW || dir == DIR_SW) ? -1 : 1;
  dy = (dir == DIR_NW || dir == DIR_NE) ? -1 : 1;
  return coords_type(coords.first + dx, coords.second + dy);
 }
 // обращение направления на противоположное
 state_type invert(state_type dir)
 {
  assert(dir >= 0 && dir < DIR_NUMBER);
  state_type inv[] = { DIR_SE, DIR_SW, DIR_NE, DIR_NW };
  return inv[dir];
 }
 // отражение направления от границ стола (если требуется)
 state_type reflect(state_type dir, const coords_type &coords)
 {
  assert(dir >= 0 && dir < DIR_NUMBER);
  state_type invx[] = { DIR_NE, DIR_NW, DIR_SE, DIR_SW };
  state_type invy[] = { DIR_SW, DIR_SE, DIR_NW, DIR_NE };
  coords_type ncoords = update(dir, coords);
  if (ncoords.first < 0 || ncoords.first >= m_xsize)
   dir = invx[dir];
  if (ncoords.second < 0 || ncoords.second >= m_ysize)
   dir = invy[dir];
  return dir;
 }

 state_type do_start(state_type state)
 {
  // инициализация координат и направления
  m_coords = coords_type(m_input[INPUT_INIT_X], m_input[INPUT_INIT_Y]);
```

```
  state = m_input[INPUT_INIT_DIR];
  // выводим координаты
  m_output[OUTPUT_X] = m_coords.first;
  m_output[OUTPUT_Y] = m_coords.second;
  m_output[OUTPUT_DIR] = state;
  return state;
 }

 state_type do_step(state_type state)
 {
  if (m_input[INPUT_ON])
  {
   coords_type other = coords_type(
    m_input[INPUT_OTHER_X], m_input[INPUT_OTHER_Y]);
   // вычисляем последующее направление шаров
   state_type mydir = reflect(state, m_coords);
   state_type otherdir = reflect(m_input[INPUT_OTHER_DIR], other);
   // проверяем совпадение координат (другой шар
   // может быть в текущей позиции или в следующей)
   if (m_coords == other || update(mydir, m_coords) == other)
   {
    // если направления противоположны
    if (invert(mydir) == otherdir)
    {
     // выполняется столкновение (смена направления)
     mydir = invert(mydir);
     // и снова отражение от возможных границ
     mydir = reflect(mydir, m_coords);
    };
   };
   // меняем текущие координаты
   m_coords = update(mydir, m_coords);
   // выводим их вместе с новым направлением
   m_output[OUTPUT_X] = m_coords.first;
   m_output[OUTPUT_Y] = m_coords.second;
   state = m_output[OUTPUT_DIR] = mydir;
  }
  else
   // закончился сигнал работы - завершаемся
   state = STATE_OFF;
  return state;
 }

public:
 ball_type(int xsize, int ysize):
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER),
  m_xsize(xsize), m_ysize(ysize)
 {
  add_handler(STATE_OFF, handler_type(&ball_type::do_start));
  for (int i = 0; i < DIR_NUMBER; ++i)
   add_handler(i, handler_type(&ball_type::do_step));
 }
};
```

Автомат предусматривает, помимо выключенного, четыре состояния (соответственно

возможным направлениям движения). В конструкторе класса `ball_type` им сопоставляются два обработчика: начало работы и выполнение шага движения (во всех направлениях).

Обработчик выключенного состояния `do_start` выполняет инициализацию шара. Здесь для краткости опущены проверки соответствия входных данных допустимым диапазонам.

Состояниям движения назначается обработчик `do_step`. В его задачи входит (в случае наличия сигнала о включении) вычисление последующих координат и направления движения шара на основе текущих его координат, направления и наличия препятствий на пути следования. С помощью функции `reflect` вычисляется последующее направление движения обоих шаров после возможного столкновения с границами стола. Далее проверяется наличие встречно двигающегося шара в текущей позиции или в последующей. В случае обнаружения такового направление меняется на противоположное (функция `invert`), после чего снова выполняется отражение от возможных границ стола. В конце происходит изменение текущих координат и направления, а также запись выходных сигналов.

Вследствие чрезвычайного упрощения реализуемой схемы возможна такая ситуация, когда между двумя шарами на одной диагонали, двигающихся навстречу друг другу, находится одна пустая клетка. В этом случае в соответствии с описанным механизмом они «не заметят» друг друга вовремя, и оба попадут на эту клетку. Именно поэтому в функции `do_step` при анализе возможного столкновения выполняется проверка наличия встречного шара не только в следующей позиции, но и в текущей.

Наконец, все описанные автоматы должны быть объединены в сеть. Для этого реализуется класс `billiard_type`, описывающий входы и выходы сети, а также объявляющий класс фабрики сети:

```
class billiard_type: public fsmnet_type
{
public:
 enum { FSM_TABLE, FSM_BALL1, FSM_BALL2, FSM_NUMBER };
 enum {
  INPUT_INIT_X1,
  INPUT_INIT_Y1,
  INPUT_INIT_DIR1,
  INPUT_INIT_X2,
  INPUT_INIT_Y2,
  INPUT_INIT_DIR2,
  INPUT_NUMBER
 };
 enum { OUTPUT_PERIOD, OUTPUT_NUMBER };

 class factory_type: public factory_abstract_type
 {
  const int m_xsize, m_ysize;
 public:
  factory_type(int xsize, int ysize):
   m_xsize(xsize), m_ysize(ysize)
  {
   assert(m_xsize > 1);
   assert(m_ysize > 1);
  }
  int number_fsm(void) const { return FSM_NUMBER; }
  int number_input(void) const { return INPUT_NUMBER; }
  int number_output(void) const { return OUTPUT_NUMBER; }
  links_type links(void) const
  {
```

```
  links_type links;
  // передача информации о координатах и направлении шаров
  for (int i = 0; i < ball_type::OUTPUT_NUMBER; ++i)
  {
   // начальные координаты и направление шаров
   links.input_to_fsm(
    INPUT_INIT_X1 + i,
    FSM_BALL1, ball_type::INPUT_INIT_X + i);
   links.input_to_fsm(
    INPUT_INIT_X2 + i,
    FSM_BALL2, ball_type::INPUT_INIT_X + i);
   // информация от шаров на входы стола
   links.fsm_to_fsm(
    FSM_BALL1, i,
    FSM_TABLE, table_type::INPUT_X1 + i);
   links.fsm_to_fsm(
    FSM_BALL2, i,
    FSM_TABLE, table_type::INPUT_X2 + i);
   // информация от шаров друг другу
   links.fsm_to_fsm(
    FSM_BALL1, i,
    FSM_BALL2, ball_type::INPUT_OTHER_X + i);
   links.fsm_to_fsm(
    FSM_BALL2, i,
    FSM_BALL1, ball_type::INPUT_OTHER_X + i);
  };
  // сигнал включения от стола к шарам
  links.fsm_to_fsm(
   FSM_TABLE, table_type::OUTPUT_ON,
   FSM_BALL1, ball_type::INPUT_ON);
  links.fsm_to_fsm(
   FSM_TABLE, table_type::OUTPUT_ON,
   FSM_BALL2, ball_type::INPUT_ON);
  // выход сети - искомая величина периода
  links.fsm_to_output(
   FSM_TABLE, table_type::OUTPUT_PERIOD,
   OUTPUT_PERIOD);
  return links;
}
fsm_abstract_type *create_fsm(int id)
{
 fsm_abstract_type *pfsm;
 if (id == FSM_TABLE)
  pfsm = new table_type(m_xsize, m_ysize);
 else
  pfsm = new ball_type(m_xsize, m_ysize);
 return pfsm;
}
void destroy_fsm(int id, fsm_abstract_type *pfsm)
{
 if (id == FSM_TABLE)
  delete dynamic_cast<table_type *>(pfsm);
 else
  delete dynamic_cast<ball_type *>(pfsm);
}
```

```
 };

public:
 billiard_type(factory_type &factory):
  fsmnet_type(factory)
 {}
};
```

Входами сети являются инициализационные данные для обоих шаров. Функция `links` класса фабрики сети задает связи между автоматами, входами и выходами сети в соответствии с рис. 3.7.

Выполнение цикла работы такой сети может быть осуществлено, к примеру, следующим кодом:

```
billiard_type::factory_type factory(8, 6);
billiard_type fsmnet(factory);

fsm_type::signals_type ins(billiard_type::INPUT_NUMBER);
// на первом шаге рядом с шарами не должно быть препятствий
// (каждый шар не должен примыкать к границе или другому шару)
ins[billiard_type::INPUT_INIT_X1] = 1;
ins[billiard_type::INPUT_INIT_Y1] = 1;
ins[billiard_type::INPUT_INIT_DIR1] = ball_type::DIR_NE;
ins[billiard_type::INPUT_INIT_X2] = 6;
ins[billiard_type::INPUT_INIT_Y2] = 4;
ins[billiard_type::INPUT_INIT_DIR2] = ball_type::DIR_NW;

fsm_type::signal_type period = 0;
fsmnet.put_input(ins);
do
{
 fsmnet.do_work();
 period = max(
  period,
  fsmnet.get_output()[billiard_type::OUTPUT_PERIOD]);
} while (!fsmnet.is_off());
```

После создания объектов фабрики и сети формируется входной вектор сети. Далее производится передача входного вектора непосредственно в сеть и выполняется цикл ее работы. Автоматная сеть устроена так, что входной вектор учитывается лишь на первом такте, вследствие чего запись входных значений внутри цикла не производится.

# Глава 4.

# Сети Петри

Настоящая глава посвящена описанию модели сетей Петри и вопросу создания параллельных программ на их основе. Сети Петри представляют собой аппарат моделирования динамических дискретных систем и являются одним из наиболее удобных способов описания асинхронного выполнения параллельных процессов, в том числе в распределенных системах.

Следует отметить, что, поскольку сети Петри изначально предназначены больше для моделирования систем и анализа их поведения, представление на базе них архитектуры проектируемой программной системы, в том числе с наличием параллелизма, а также последующая ее реализация, для многих разработчиков оказываются сопряженными с некоторыми сложностями. Это является в том числе следствием не слишком высокой согласованности этой модели с популярными на текущий момент методами программирования. Мы же здесь покажем, что, несмотря на это, они являются более мощным средством построения параллельных вычислений, включающим в себя возможности существующих популярных средств. Кроме того, представление вычислительного процесса в виде сети Петри может быть не привязано к самой реализации ее функционирования и может быть построено путем выделения подзадач в отдельные функциональные блоки. На совести сети Петри в этой ситуации остается лишь организация последовательности их выполнения и синхронизация.

## 4.1. Краткое введение в теорию сетей Петри

Мы не будем углубляться в изложение теории сетей Петри, поскольку это довольно широкая тема, а ограничимся лишь поверхностным описанием их функционирования, достаточным для иллюстрации возможностей их использования при построении параллельных программ. В частности, мы не затронем здесь вопросы анализа сетей Петри, поскольку это, скорее, цель моделирования, при программной же реализации предполагается, что этот этап уже пройден. Для более детального ознакомления с теорией сетей Петри следует обратиться к посвященной этой теме литературе [18, 29, 23].

### 4.1.1. Знакомство с сетями Петри

В некотором приближении можно сказать, что сеть Петри обобщает понятие конечного автомата и добавляет ему некоторые свойства, присущие сетям конечных автоматов. В каком-то смысле сеть Петри напоминает диаграмму состояний автомата, который может находиться одновременно в нескольких состояниях. Каждый переход обуславливается каким-либо подмножеством текущих состояний и заменяет его на другое подмножество

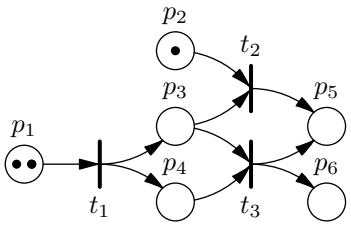



Рис. 4.1. Пример сети Петри

состояний. Позиции и переходы сети Петри в такой интерпретации играют роль состояний автомата и действий на переходах соответственно.

### Состав

Графически сеть Петри представляется в виде двудольного ориентированного графа с двумя типами вершин: позициями и переходами (рис. 4.1). Позиция обозначается кругом, переход — чертой или прямоугольником. Как правило, чертой обозначают простой (мгновенный) переход, прямоугольником — длительный. В позициях могут находиться фишки — некая сущность, наличие которой говорит о том, что условие, соответствующее текущей позиции, выполняется. К примеру, наличие фишки может говорить о наличии входных данных для некоторой процедуры. Наличие нескольких фишек говорит о том, что условие выполнено с многократным запасом. В примере с процедурой это может быть наличие нескольких элементов в очереди ее входных данных. Количество фишек в каждой позиции является целым неотрицательным числом. На графе наличие фишек в позиции обозначается числом либо жирными точками в соответствующем количестве. Совокупность всех фишек, размещенных в позициях сети Петри, называется разметкой сети.

Дуги в сети Петри могут быть направлены лишь от позиций к переходам либо от переходов к позициям. При этом они могут быть кратными, т.е. иметь вес — также целое неотрицательное число (дуга помечается этим числом на графе). Наличие кратной дуги между позицией и переходом эквивалентно наличию между ними соответствующего количества простых дуг в том же направлении. Простая дуга имеет кратность, равную единице. Дуги, направленные от позиций к переходам, называются входными, направленные от переходов к позициям — выходными. Аналогичными терминами обозначаются и соответствующие позиции по отношению к некоторому переходу.

Таким образом, сеть Петри полностью описывается следующим набором параметров:

- множество позиций;
- множество переходов;
- множество входных дуг (дуг от позиций к переходам);
- множество выходных дуг (дуг от переходов к позициям);
- множество фишек, размещенных в позициях изначально (начальная разметка).

### Принципы функционирования

Функционирование сети Петри осуществляется в виде последовательности срабатывания переходов. В каждый момент времени в сети есть некоторое количество разрешенных

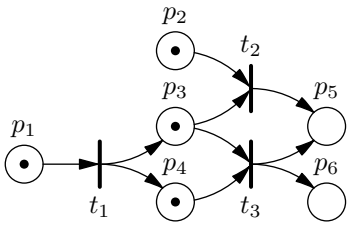



Рис. 4.2. Результат первого срабатывания

переходов, т.е. таких, которые могут сработать. Переход считается разрешенным, если в каждой его входной позиции количество фишек не меньше, чем кратность соответствующей входной дуги. Из всего множества разрешенных переходов сработать может любой. Какой именно сработает, определяется вне сети, также как и момент его срабатывания во времени. Выбор может быть осуществлен на основе ожидания аппаратного события, события завершения длительного процесса, указания пользователем и т.п. При срабатывании простого перехода из каждой его входной позиции изымаются фишки в количестве, равном кратности соединяющей ее с переходом входной дуги, после чего к каждой выходной позиции добавляются фишки в количестве, также равном кратности соответствующей выходной дуги.

К примеру, в сети на рис. 4.1 разрешенным является один переход — $t_1$ (остальным переходам не хватает фишек во входных позициях). При его срабатывании из единственной входной позиции изымается одна фишка, поскольку соответствующая входная дуга — простая, т.е. ее кратность равна единице. В обе выходные позиции помещается также по одной фишке, поскольку соответствующие дуги также являются простыми. В этот момент разрешенными являются все три перехода сети (рис. 4.2).

После каждого срабатывания перехода сети Петри множество разрешенных переходов в общем случае меняется, поскольку меняется текущая разметка сети. Считается, что сеть Петри «жива», если в ней есть разрешенные переходы. Если после срабатывания очередного перехода больше разрешенных переходов в сети не остается, она завершает выполнение.

В случае, когда позиция является входной для двух или более разрешенных переходов, срабатывание любого из них и соответствующее уменьшение количества фишек в позиции может запретить другие переходы. Такой ситуацией моделируются конфликты, когда для выполнения различных операций требуется использование общего ресурса. Пример такой ситуации виден на рис. 4.2. Все переходы сети являются разрешенными, однако переходы $t_2$ и $t_3$ имеют общую входную позицию $p_3$. Если первым сработает, к примеру, переход $t_2$ (рис. 4.3), тем самым он на какое-то время запретит срабатывание перехода $t_3$ (пока снова не сработает $t_1$). И наоборот, срабатывание $t_3$ запретило бы переход $t_2$. Сеть на рис. 4.3 содержит лишь один разрешенный переход $t_1$. После его срабатывания и последующего срабатывания $t_3$ выполнение сети завершается (рис. 4.4).

**Атомарность**

Срабатывание простого перехода считается мгновенным. Поскольку вероятность одновременного происхождения двух мгновенных событий равна нулю, два простых перехода не могут сработать одновременно [29]. Отсюда вытекает довольно парадоксальное свойство сетей Петри: притом, что моделируемые ими процессы в общем случае параллельны, само по себе выполнение сети Петри происходит строго последовательно. Именно невозможностью одновременного срабатывания каких-либо переходов определяется асинхронная природа

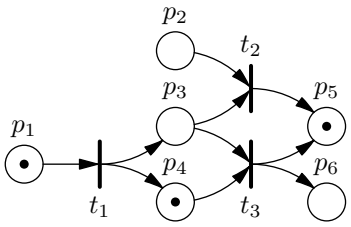



Рис. 4.3. Результат возможного следующего срабатывания

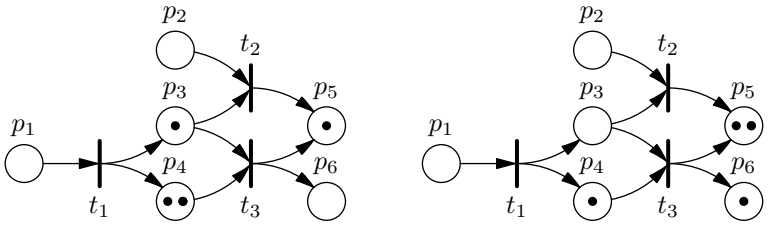



Рис. 4.4. Результат последующих срабатываний и завершение работы

сетей Петри. Параллелизм же, приписываемый им, выражается в том, что в один момент времени разные участки сети могут отражать выполнение разных независимых процессов.

### Движение фишек

Важно понимать, что многократно упоминаемые в тексте «перемещения» фишек между позициями — не более чем интуитивно понятное описание. Формально оно неверно, поскольку, в самом деле, фишки не перемещаются из позиции в позицию, а исчезают в переходе «в никуда» и появляются в нем же «из ниоткуда». По этим причинам в общем случае их количество в сети до и после срабатывания перехода разное, как в примере выше. Случай, когда количество фишек в сети всегда постоянно (и тогда только можно говорить о перемещении фишек), является частным и относится к сетям, обладающим свойством строгого сохранения.

В этом отношении полезно рассмотреть следующий пример. На рис. 4.5 приведены три простых сети, каждая из которых содержит лишь один переход. Во всех трех сетях этот переход разрешен. В сети, содержащей входную позицию, после первого срабатывания перехода единственная фишка исчезает, и сеть завершает выполнение. Поскольку выходных позиций у перехода нет, фишки нигде не появляются. В обеих остальных сетях переходы не имеют входных позиций, в связи с чем разрешены всегда. Сеть, содержащая выходную позицию, неограниченно накапливает в ней фишки. Сеть же, не содержащая позиций вообще, не потребляет и не генерирует фишки, однако переход также обречен срабатывать вечно.

### Расширения

Помимо приведенного описания классических сетей Петри, существуют различные расширенные модели [14]. К примеру, цветные сети Петри дополняются введением типов фишек и помогают избежать многократного дублирования фрагментов сети в сложных системах. Сети Петри с приоритетами добавляют к разрешенным переходам приоритеты и тем самым позволяют снизить недетерминированность срабатываний, ограничивая множество

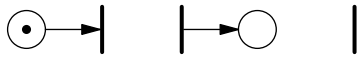


Рис. 4.5. Примеры переходов без входных и/или выходных позиций

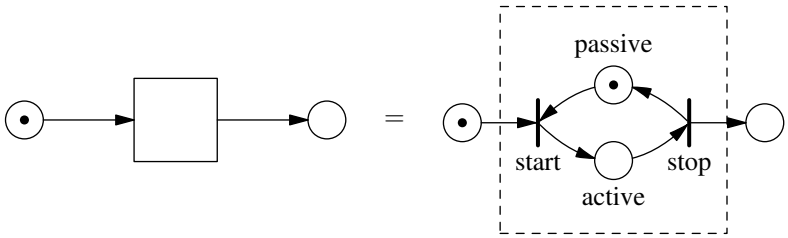



Рис. 4.6. Длительный переход

разрешенных переходов группой переходов с наивысшим приоритетом. Ингибиторные сети вводят понятие запрещающей (сдерживающей) входной дуги, при наличии которой переход считается разрешенным и может сработать лишь в случае отсутствия фишек в соответствующей входной позиции. Существуют и другие расширения, из которых мы рассмотрим строго иерархические сети [18].

### 4.1.2. Строго иерархические сети

Выше мы уже говорили о том, что срабатывание простого перехода в сети Петри мгновенно. Однако зачастую переходами моделируется выполнение каких-либо далеко не мгновенных операций. В таких случаях операции может быть сопоставлен так называемый длительный переход. Для наглядности длительные переходы, в отличие от простых, часто обозначаются прямоугольником (рис. 4.6, слева).

Длительный переход, как и простой, начинает выполнение с изъятия фишек из своих входных позиций и завершает, соответственно, помещая фишки в выходные. Отличие от простого перехода заключается в том, что между этими двумя моментами могут срабатывать другие переходы. Иначе говоря, срабатывание длительного перехода не атомарно. По этой причине выполнение различных длительных переходов может перекрываться во времени, т.е. осуществляться параллельно.

С момента начала выполнения до момента завершения длительный переход считается активным; в случае же, если он в данный момент не выполняется, — пассивным. Длительный переход не может сработать, если он уже активен, даже если он разрешен по наличию фишек во входных позициях.

Выполнение длительного перехода эквивалентно выполнению фрагмента сети, изображенного на рис. 4.6, справа. Здесь простые переходы *start* и *stop* характеризуют, соответственно, начало и завершение выполнения длительного перехода, позиция *active* характеризует активность длительного перехода, позиция *passive* — его пассивность. Дополнительная позиция, говорящая о состоянии пассивности перехода, необходима для защиты от повторного срабатывания уже активного перехода. Если длительный переход активен, фишки в ней нет, в связи с чем простой переход *start* запрещен. При завершении работы длительного перехода простой переход *stop* снова помещает фишку в позицию *passive*, разрешая тем самым переход *start*.

Операция, выполняемая в течение длительного перехода, может содержать полный жизненный цикл какой-либо вложенной подсети (рис. 4.7). В этом случае соответствую-

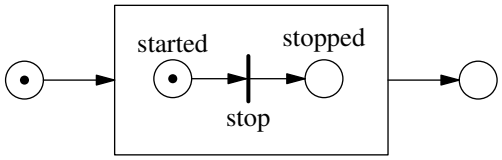



Рис. 4.7. Пример составного перехода

щий длительный переход называется составным. На основе подобного включения могут быть организованы иерархические сети произвольной вложенности. Если сеть не содержит дуг, соединяющих позиции и переходы разных уровней вложенности или же разных подсетей одного уровня, то такая сеть называется строго иерархической [18]. В дальнейшем мы будем говорить лишь о строго иерархических сетях.

Вложенная подсеть составного перехода начинает жизненный цикл в момент его активации, после извлечения фишек из входных позиций перехода. Функционирование составного перехода каждый раз при его срабатывании выполняется, исходя из начальной разметки вложенной подсети. Когда после очередного срабатывания какого-либо внутреннего перехода подсети в ней не остается разрешенных переходов, она прекращает работу, завершая выполнение соответствующего составного перехода. В этот момент составной переход помещает фишки в выходные позиции и переходит в пассивное состояние. При следующей активации составного перехода вложенная подсеть начинает новый жизненный цикл, будучи снова инициализированной в соответствии со своей начальной разметкой.

Вследствие отсутствия связей внутренних позиций и переходов вложенной подсети с внешними по отношению к ней, структура вложенной подсети скрыта от сети, охватывающей ее, и может быть реализована полностью независимо. С точки зрения охватывающей сети любой составной переход произвольной сложности, так же как и вообще любой длительный переход, может быть представлен в виде составного перехода, изображенного на рис. 4.7 и содержащего простую вложенную подсеть. Единственный простой переход в этой сети характеризует завершение работы длительного перехода. Это предоставляет определенное удобство использования иерархических сетей в параллельном программировании, поскольку длительные операции любого характера могут быть выполнены в виде составных переходов, которые могут выполняться параллельно.

### 4.1.3. Параллельные вычисления и синхронизация

Сетями Петри легко моделируется создание и выполнение параллельных ветвей различных вычислительных процессов. К примеру, на рис. 4.8 отражена одна из наиболее популярных на сегодняшний день конструкций ветвления $fork/join$. Переход $fork$ моделирует «разветвление» — создание из одной ветви выполнения двух параллельных ветвей. Это, как правило, реализуется путем создания одной дополнительной ветви вдобавок к существующей. Переход $join$, в свою очередь, осуществляет «слияние» двух ветвей по завершению их работы (уничтожение созданной параллельной ветви за ненадобностью). Два длительных перехода, лежащих между простыми переходами $fork$ и $join$, могут выполняться в произвольном последовательном порядке или же параллельно.

Широко используемые объекты синхронизации также легко моделируются сетями Петри. К примеру, на рис. 4.9 изображен фрагмент сети Петри, моделирующий барьерную синхронизацию трех параллельных процессов. В начале работы сети разрешенными являются три длительных перехода, характеризующих выполнение некоторых подзадач. Эти три подзадачи выполняются независимо, и лишь после их полного завершения становится

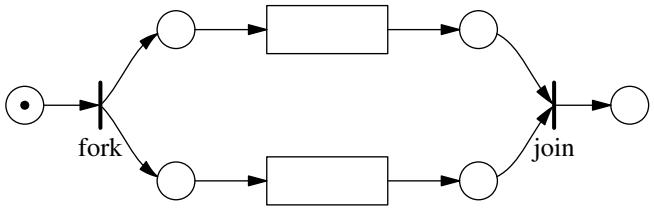



Рис. 4.8. Ветвление и слияние процессов

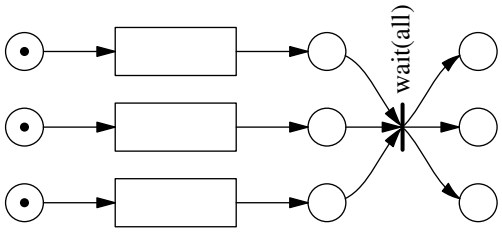



Рис. 4.9. Ожидание завершения всех процессов

разрешенным переход-барьер $wait(all)$. После его срабатывания все три процесса готовы к продолжению работы.

Похожим образом может быть смоделировано ожидание завершения любой из подзадач, первой завершившей свое выполнение. На рис. 4.10 показан пример такого фрагмента сети. Предполагается, что при завершении любой подзадачи результат ее работы требует некоторой обработки основным процессом (к примеру, добавление результата к общей сумме). При завершении выполнения любого длительного перехода становится разрешенным переход $wait(any)$. После его срабатывания осуществляется обработка результата, в то время как остальные подзадачи продолжают свое выполнение. Дополнительная входная позиция перехода $wait(any)$ защищает его от срабатывания при завершении остальных подзадач в случае, если в текущий момент ожидание не выполняется. В процессе дальнейшей работы сети фишка в эту позицию возвращается, и тем самым разрешается ожидание и обработка результатов выполнения остальных подзадач. После обработки всех трех результатов все процессы продолжают работу независимо. Последний переход выполняет роль барьера, поскольку из-за кратности входной дуги срабатывает он лишь тогда, когда завершаются все три операции.

Другой пример синхронизации параллельных процессов — критическая секция. Фрагмент сети на рис. 4.11 иллюстрирует взаимоисключающий доступ к некоторому общему ресурсу для двух процессов. Первый же процесс, захвативший доступ к критической секции, изымет фишку из соответствующей ей позиции $res$, тем самым запретив доступ к критической секции второму процессу до момента ее освобождения. В случае, когда в позиции $res$ изначально более одной фишки, такой фрагмент сети моделирует использование семафора — объекта синхронизации, предоставляющего одновременный доступ к ресурсу нескольким процессам в количестве не более заданного.

Важная особенность сетей Петри заключается в атомарности срабатывания перехода и, как следствие, возможности атомарного захвата одновременно нескольких ресурсов, что позволяет исключить возможность взаимоблокировки процессов (dead lock). Рассмотрим такую ситуацию на примере. Допустим, имеется два процесса, каждому из которых необходим одновременный доступ к двум общим ресурсам. Попробуем реализовать это классическим путем — с помощью критических секций. Допустим, порядок захвата критических

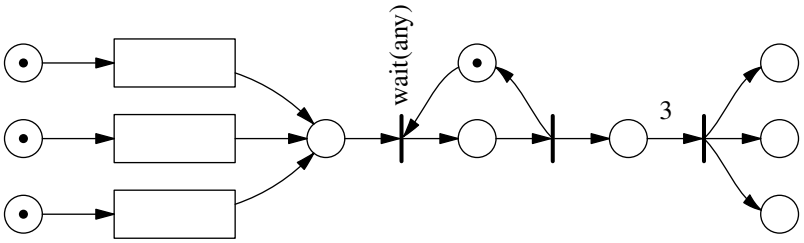



Рис. 4.10. Ожидание завершения любого процесса

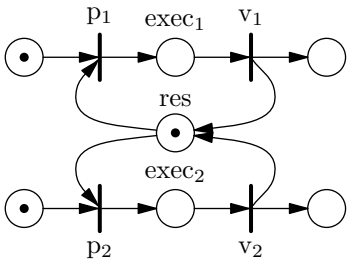



Рис. 4.11. Взаимоисключающий доступ к одному ресурсу

секций заранее неизвестен. Он может определяться некоторыми внешними обстоятельствами во время выполнения (к примеру, захват ресурсов может осуществляться в порядке их запроса пользователем). Вариант такой реализации, описанный в терминах сетей Петри, изображен на рис. 4.12.

Каждому из двух процессов требуются два ресурса, доступность которых характеризуют фишки в позициях $res_1$ и $res_2$. Оба процесса пытаются захватить ресурсы в произвольной последовательности. Если переходы $p_{11}$ и $p_{12}$ сработают раньше, чем переходы $p_{21}$ и $p_{22}$ (т.е. сначала первый процесс захватит оба ресурса, потом второй), либо наоборот, оба процесса благополучно закончат свое выполнение. Если же первыми сработают переходы $p_{11}$ и $p_{22}$ или $p_{12}$ и $p_{21}$ (оба процесса захватят по одному ресурсу), каждый процесс окажется навсегда заблокированным в ожидании другого (фишки в позиции завершения процессов не попадут).

Проблема может быть решена путем ухода от произвольной последовательности захвата ресурсов и введения жесткого порядка. К примеру, можно обязать оба процесса пытаться сначала захватить ресурс, связанный с позицией $res_1$. Фрагмент сети, моделирующий работу такой системы, изображен на рис. 4.13. Здесь неоднозначность возникает только в отношении того, какой именно процесс первым захватит ресурс $res_1$. Ресурс $res_2$ может быть захвачен лишь тем процессом, который уже захватил $res_1$, в связи с чем кольцевая зависимость процессов друг от друга невозможна. На этом основан известный принцип предотвращения взаимоблокировок путем задания всем ресурсам уникальных идентификаторов и последующего захвата ресурсов в порядке возрастания или убывания этих идентификаторов. В такой ситуации граф ожидания процессов всегда будет ациклическим, и опасность взаимоблокировки не возникает.

Для осуществления захвата двух ресурсов мы пытались строить сеть на основе рассмотренного выше популярного примитива синхронизации (критической секции). Однако попробуем использовать конструкции, моделирующие не использование объектов синхронизации, а непосредственно требуемый ход событий и учет условий их возникновения, к примеру, доступность ресурса. В частности, отразим непосредственно в сети тот факт, что

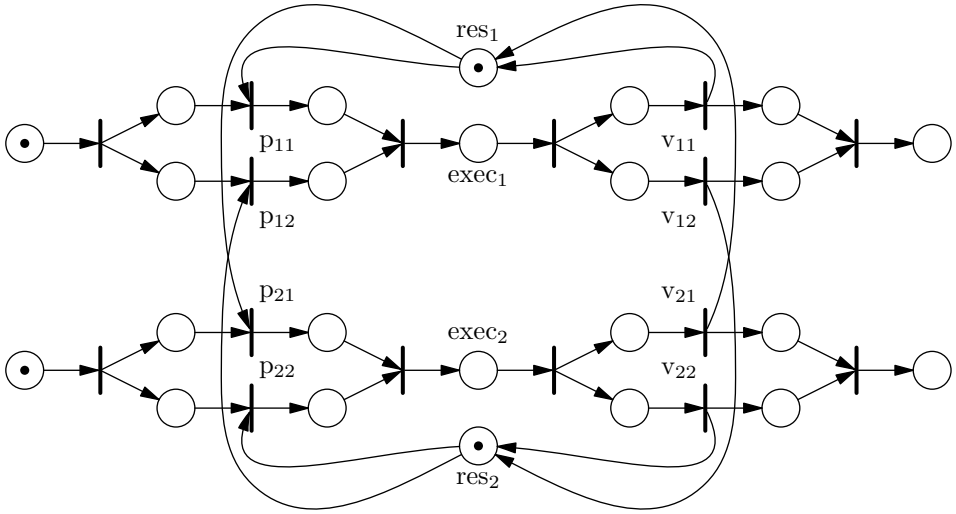



Рис. 4.12. Поочередный захват двух ресурсов в произвольном порядке

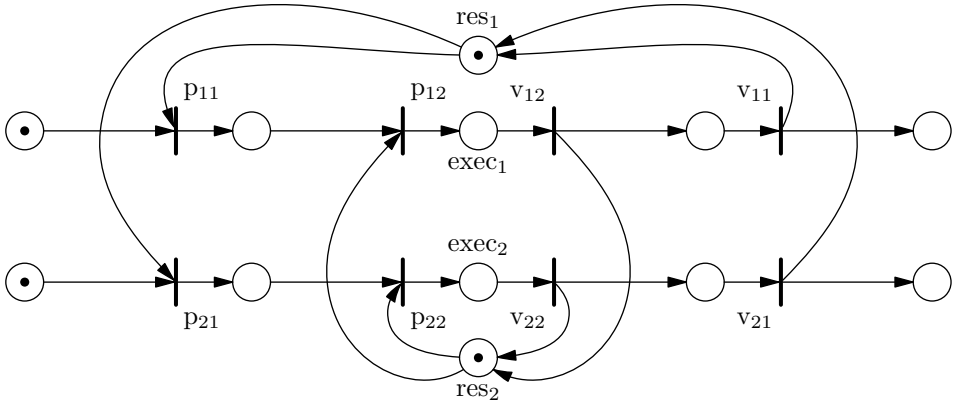



Рис. 4.13. Упорядоченный поочередный захват двух ресурсов

для начала выполнения операции процессу необходимо наличие сразу двух свободных ресурсов. Полученная в результате сеть (рис. 4.14) успешно решает поставленную задачу без введения порядка захвата и без опасности возникновения взаимоблокировки: захват обоих ресурсов осуществляется одновременно в момент срабатывания переходов $pp_1$ или $pp_2$.

В последнем примере видны преимущества сетей Петри перед популярными сегодня средствами синхронизации. С помощью них оказывается проще организовать сложное и безопасное взаимодействие различных участков системы, особенно в ситуациях, когда в разрабатываемой системе происходит широкое использование критических разделяемых ресурсов.

### 4.1.4. Задача об обедающих философах

Одной из классических задач, иллюстрирующих проблемы синхронизации процессов, является задача об обедающих философах. Ее постановка довольно проста и вкратце заключается в следующем. За круглым столом сидят пятеро философов, перед ними стоит блюдо спагетти. На столе лежат пять вилок, по одной между каждыми двумя соседними философами. По условию каждому философу для еды необходимо две вилки, лежащие

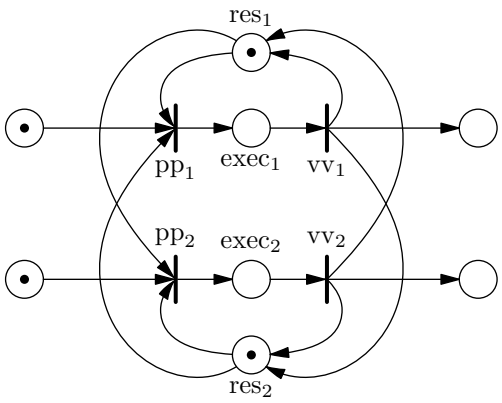



Рис. 4.14. Одновременный захват двух ресурсов

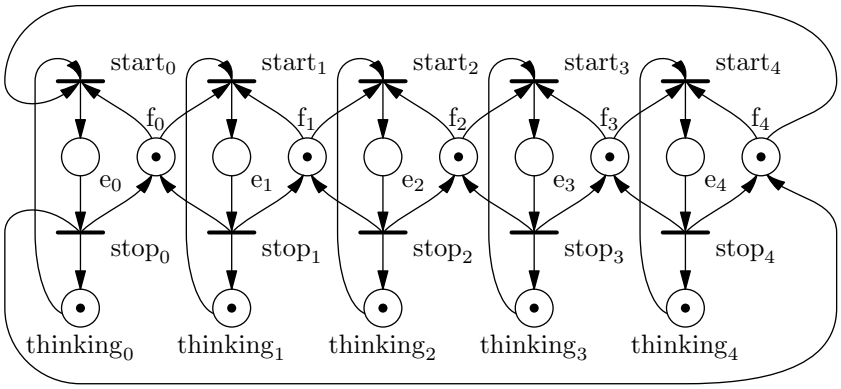



Рис. 4.15. Сеть, решающая задачу об обедающих философах

непосредственно слева и справа от него. Каждый философ пребывает за столом в одном из двух состояний: размышляет или ест. В последнем случае оба его ближайших соседа вынуждены размышлять, поскольку для еды им не хватает вилок.

Основная проблема, иллюстрируемая этой задачей, — проблема возможности взаимоблокировки, которая была описана выше. В случае реализации с последовательным захватом вилок возможна ситуация, когда одновременно все философы возьмут, к примеру, левую от себя вилку. Тогда ни один из них не сможет взять правую вилку, и каждый окажется заблокированным в вечном ожидании ее освобождения.

Как уже говорилось выше, проблема взаимоблокировки легко может быть решена сетями Петри, поскольку они предоставляют возможность атомарного захвата одновременно нескольких ресурсов.

На рис. 4.15 изображена сеть Петри, решающая задачу об обедающих философах [29]. Позиции $eating_i(e_i)$ и $thinking_i$ характеризуют здесь пребывание $i$-го философа в состоянии еды или размышлений. Позициями $fork_i(f_i)$ обозначается доступность вилок. Переходы $start_i$ и $stop_i$ характеризуют переход $i$-го философа к еде и к размышлениям соответственно. По сути, такая сеть представляет собой дополненный итерацией и расширенный до пяти процессов вариант сети, изображенной на рис. 4.14.

В некоторых ситуациях вводится третье возможное состояние философа: философ голоден. В этом случае к каждому соответствующему участку сети добавляется еще одна промежуточная позиция $hungry_i$, однако суть от этого не меняется. Позже мы рассмотрим

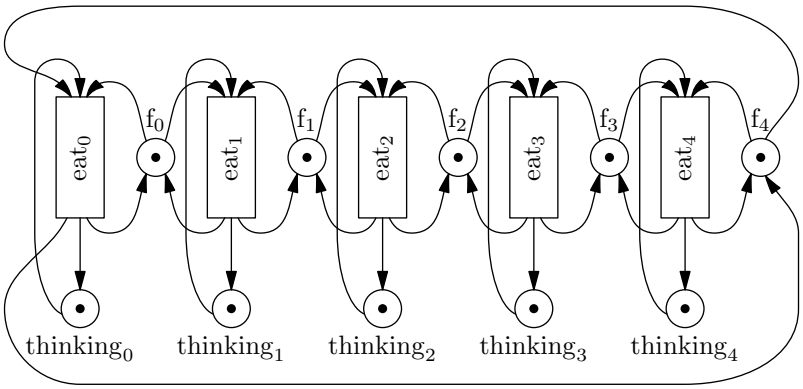



Рис. 4.16. Сеть с длительными переходами

вариант решения, в котором оказывается удобно ввести такое промежуточное состояние.

Однократный процесс приема пищи каждым философом является длительной операцией и может быть представлен в виде длительного составного перехода, изображенного на рис. 4.7. В этом случае срабатывание перехода *stop* характеризует завершение приема пищи философом. На рис. 4.16 изображена сеть с использованием составных переходов $eat_i$, содержимое которых здесь опущено для компактности.

Помимо проблемы взаимоблокировки, задача об обедающих философах демонстрирует возможность голодания (заговора соседей), т.е. возможность такой ситуации, в которой один или более философов никогда не получат доступ к еде. Обычно проблема голодания удачно решается регламентными средствами, поэтому применение сетей Петри здесь не столь иллюстративно, как при решении проблемы взаимоблокировки. Однако сетями Петри, в свою очередь, могут быть реализованы соответствующие требования регламента.

К примеру, можно потребовать, чтобы в соответствии с регламентом философы принимали пищу поэтапно, причем в рамках этапа каждый философ получал доступ к еде лишь единожды. Такое решение является довольно простым, хотя и далеко не самым лучшим. Подобное требование может быть реализовано сетью Петри следующим образом. В сеть, изображенную на рис. 4.16, внесем барьерную синхронизацию после того, как каждый философ осуществит прием пищи по одному разу. Для этого добавим к каждому $i$-му философу по две позиции: позицию $todo_i$, число фишек в которой отражает количество предстоящих подходов философа к еде до следующего барьера, и позицию $done_i$, отражающую количество уже выполненных подходов. Полученная сеть изображена на рис. 4.17. В случае если синхронизация после каждого подхода к еде является слишком частой мерой, количество фишек в каждой позиции $todo_i$ может быть увеличено. Более того, начальные количества фишек в этих позициях могут быть заданы различными, в зависимости от потребности в интенсивном питании каждого из философов. В соответствии с начальным количеством фишек в позициях $todo_i$ должна быть изменена и кратность дуг, ведущих в переход-барьер и из него.

С учетом такой модификации сети $i$-й философ будет получать доступ к еде в обычном порядке, пока не закончатся фишки в соответствующей позиции $todo_i$, после чего остановится на размышлениях. Таким образом, пока каждый из пяти философов не получит доступ к еде заданное количество раз, переход к следующему этапу осуществлен не будет. Когда все фишки из позиций $todo_i$ постепенно переместятся в соответствующие позиции $done_i$, переход-барьер снова переместит их в позиции $todo_i$, и начнется следующий цикл приема пищи.

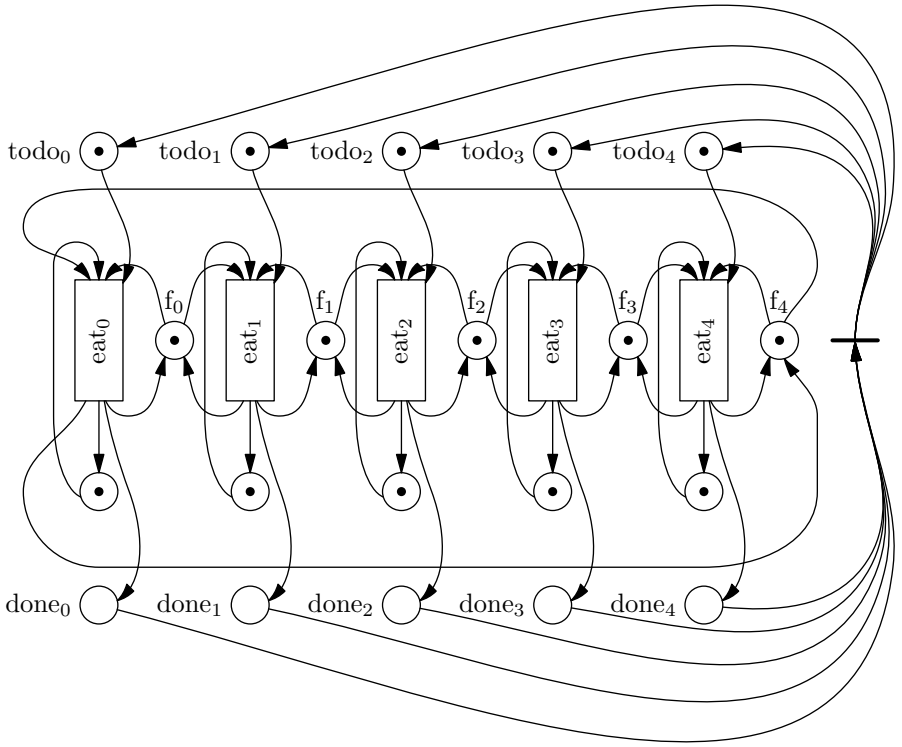



Рис. 4.17. Модифицированная сеть с ограничением на подходы к еде

Подобное решение проблемы голодания представляется недостаточно гибким, поскольку интенсивность питания каждого философа задается статически на уровне структуры сети, в то время как на самом деле его потребности могут меняться динамически. Более гибким считается, как назвали его авторы, «гигиеническое» решение (Chandy-Misra solution) [57]. В соответствии с ним любая вилка в каждый момент находится у одного из двух рядом сидящих философов (статус принадлежности) и может быть при этом чистой или грязной (статус чистоты). В процессе выполнения действуют следующие правила:

- если философ голоден, и у него в наличии обе вилки (чистые или не запрошенные соседями), он приступает к еде;
- после еды обе вилки становятся грязными и остаются таковыми до передачи соседу;
- если философ голоден, но для еды ему не хватает вилки, он запрашивает ее у соседа;
- обработка запросов вилки, пришедших от соседей во время еды философа, откладывается до момента ее завершения (процесс приема пищи не прерывается);
- при получении запроса на вилку философ чистит ее и отдает, если она грязная, в противном случае оставляет вилку себе (откладывает обработку запроса до тех пор, пока вилка не станет грязной).

Суть такого решения заключается в том, что приоритет обладания вилкой имеет философ, который ее еще не использовал с момента получения (вилка чистая). После приема пищи вилка становится грязной, и приоритет обладания вилкой у текущего философа снижается, вынуждая его отдать вилку соседу (если он голоден). Ключевым моментом,

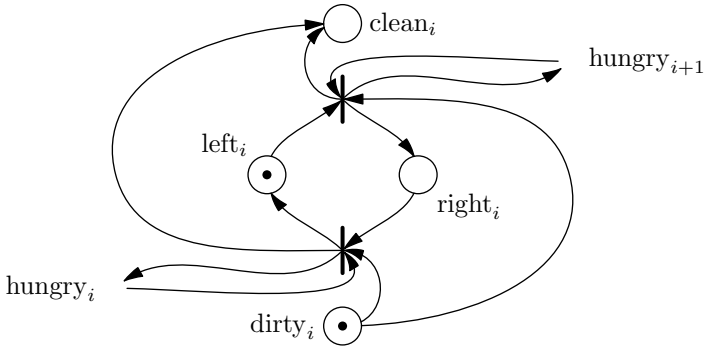



Рис. 4.18. Фрагмент сети, отражающий состояние вилки

отличающим это решение от предыдущего варианта с поэтапным доступом к еде, является то, что в данном случае философ, имея обе грязные вилки (и низкий приоритет), может есть неограниченное количество раз, если их не запрашивали соседи.

Система инициализируется грязными вилками с такой принадлежностью, чтобы философы не находились в одинаковом положении (когда у каждого из них по одной вилке). К примеру, все вилки, кроме одной, отдаются философу слева от нее, последняя — философу справа. В противном случае появляется возможность взаимоблокировки, поскольку, когда у всех философов есть по одной грязной вилке, они могут одновременно запросить и получить недостающую вторую (чистую), после чего запросить только что отданную первую и остаться в вечном ее ожидании.

Реализуем описанную схему взаимодействия с помощью сети с приоритетами. Поддержка приоритетов срабатывания переходов удачно согласуется с моделью сетей Петри, поскольку реализуется в рамках окружения, в котором функционирует сеть, и не требует внесения новых элементов в ее конструкцию и изменений в правилах ее функционирования. Для отражения потребности философа в вилках введем для него промежуточное состояние «голоден». Полная сеть будет довольно громоздкой, поэтому мы приведем изображения лишь ее фрагментов для $i$-й вилки и $i$-го философа. На каждом фрагменте номерами $i-1$ и $i+1$ будут обозначены соответственно позиции ближайшего слева и ближайшего справа фрагмента, при этом следует помнить, что, поскольку стол круглый, для первого фрагмента номером $i-1$ будет обозначен последний фрагмент, и наоборот, для последнего фрагмента $i+1$ — номер первого фрагмента. Зависимости фрагментов между собой мы отразим условно, не изображая соответствующих позиций, поскольку из-за обилия дуг это может существенно снизить наглядность.

На рис. 4.18 изображен фрагмент, отвечающий за состояние $i$-й вилки. Вилка может находиться у философа слева от нее или справа, за что отвечает фишка в позиции соответственно $left_i$ или $right_i$. В то же время она может быть чистой или грязной (фишка в позиции $clean_i$ или $dirty_i$). Два перехода отражают моменты передачи вилки от философа к философу. Философ отдает вилку только тогда, когда она грязная, и принимает от соседа только чистую, что отражается соответствующими дугами. Передача вилки соседу осуществляется лишь в случае, если он голоден (зависимости от не приведенных на этом фрагменте позиции $hungry_i$ левого философа и позиции $hungry_{i+1}$ правого философа). Поскольку при срабатывании перехода передачи вилки голодный философ остается голодным, фишка в соответствующую позицию возвращается дугой в обратном направлении.

Фрагмент сети на рис. 4.19 отражает состояние $i$-го философа. Для выполнения приема пищи философу необходимо, чтобы вилка слева от него находилась в правом положении

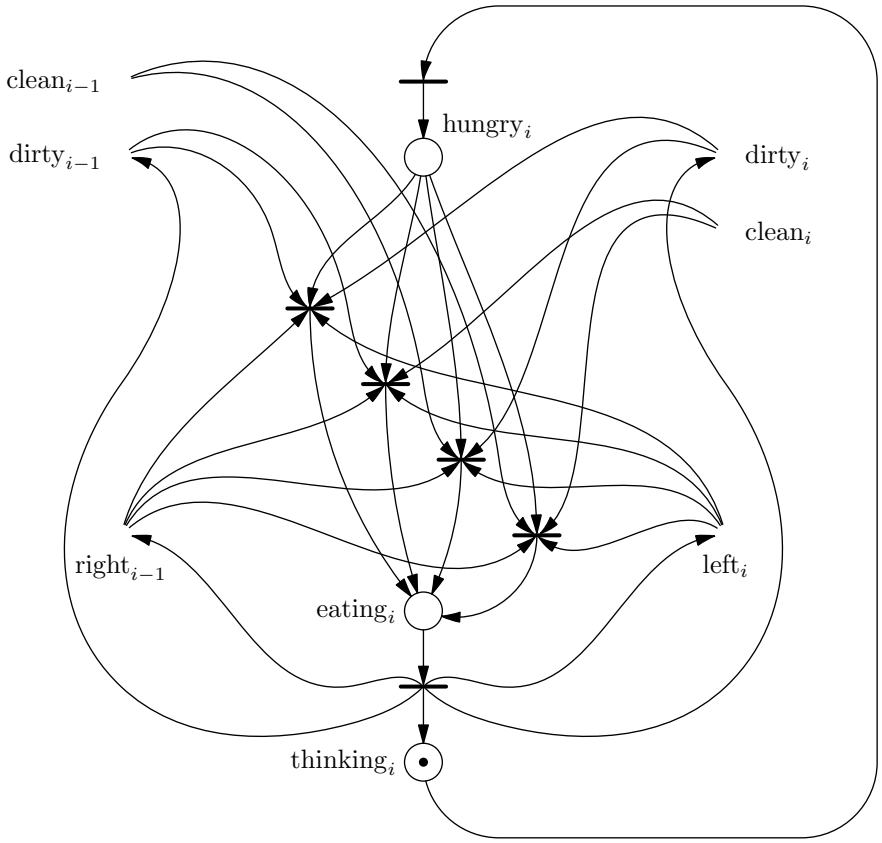



Рис. 4.19. Фрагмент сети, отражающий состояние философа

(т.е. была у него), а вилка справа от него, соответственно, в левом. При этом неважно, чистые вилки или грязные. После еды вилка в любом случае становится грязной, и для обеспечения корректной разметки в этой ситуации мы должны в момент начала еды изымать фишку также и из позиции статуса ее чистоты. Это вынуждает нас в момент начала еды философом рассматривать четыре возможных случая: обе вилки грязные, обе чистые и два случая вилок с разным статусом. В связи с этим фрагмент на рис. 4.19 содержит четыре перехода начала приема пищи. Каждый из них изымает все фишки из обоих близлежащих фрагментов состояния вилок, для чего он соединяется входными дугами с соответствующими четырьмя позициями. Переход завершения приема пищи, в свою очередь, также соединяется выходными дугами с четырьмя позициями двух фрагментов вилок (обе вилки помечаются грязными).

Полная сеть, отражающая работу всей системы, строится путем совмещения пяти фрагментов, изображенных на рис. 4.19, с пятью фрагментами, изображенными на рис. 4.18. В каком-либо одном случае разметка фрагмента вилки (рис. 4.18) меняется с принадлежности философу слева от нее на противоположную, чтобы обеспечить невозможность взаимоблокировки. Для гарантированного решения проблемы голодания требуется внесение в сеть приоритетов. Переходы передачи вилки от философа к философу на рис. 4.18 имеют приоритет над остальными переходами сети, поскольку при наличии двух грязных вилок и голодающих соседей философ должен отдать вилку, а не приступать к еде. Таким образом, окружение, в котором функционирует сеть, должно обеспечивать срабатывание одного из переходов передачи вилки, если таковые есть среди разрешенных, и лишь в противном случае позволять срабатывание какого-либо другого перехода.

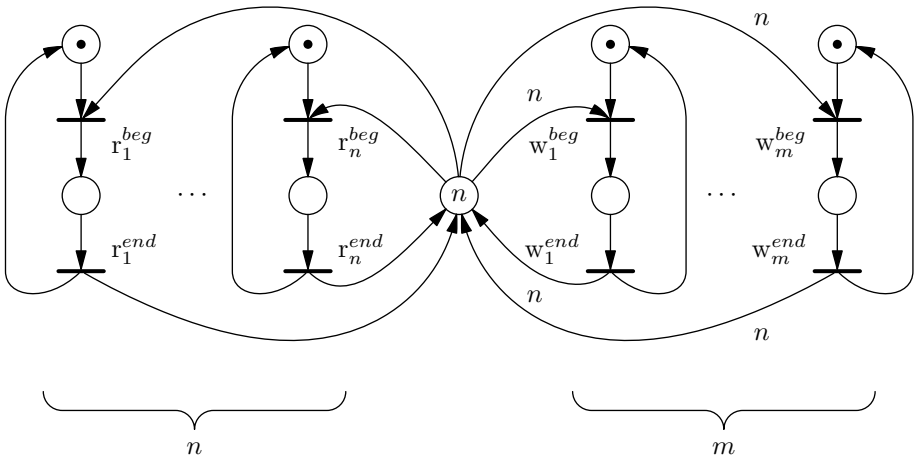



Рис. 4.20. Задача чтения-записи с отражением состояния всех процессов

### 4.1.5. Задача чтения-записи

Еще одна проблема синхронизации параллельных процессов иллюстрируется задачей чтения-записи. Предположим, есть система, содержащая некоторый ресурс общего доступа. Ресурс допускает выполнение операций чтения и записи (получения некоторой содержащейся в нем информации и ее модификации). При этом операция чтения допускает одновременное выполнение других операций чтения, а операция записи исключает выполнение любых других операций (для обеспечения целостности информации).

Помимо ресурса, система содержит $n$ параллельных процессов, периодически осуществляющих чтение данных из ресурса, и $m$ процессов осуществляющих запись. Если отображать состояние системы подробно, получим сеть, изображенную на рис. 4.20. В ней отражено не только состояние ресурса, но и состояние всех параллельных процессов. В левой части сети собраны участки, отображающие состояния $n$ читающих процессов, в правой части — $m$ пишущих процессов. Позиция в центральной части сети характеризует доступность ресурса. Каждый читающий процесс в момент начала операции чтения изымает из нее одну фишку. Поскольку фишек в ней хватает на все читающие процессы, они не взаимоисключают друг друга. Каждый пишущий процесс соединен с позицией доступности ресурса дугой кратности $n$, в связи с чем для начала любой операции записи необходимо наличие в ней всех $n$ фишек. Таким образом, если хоть один процесс осуществляет чтение, операция записи начата быть не может (фишек недостаточно). Если же выполняется операция записи, фишек в позиции доступности ресурса нет, в связи с чем не может быть начата ни одна операция чтения.

Обычно реальное количество читающих процессов в системе и количество фишек в позиции доступности ресурса указывают в общем случае разными [38], поскольку доступность ресурса (и соответствующая кратность дуг к пишущим процессам) характеризует лишь верхнюю границу возможного количества одновременно читающих процессов, а не реальное их количество. Мы же опустим эту подробность, чтобы не усложнять восприятие введением дополнительных параметров.

В некоторых ситуациях нас интересует состояние лишь самого ресурса, а не обращающихся к нему параллельных процессов. Тогда более естественным будет альтернативное представление, отражающее лишь общее состояние системы без деталей выполнения каждого отдельно взятого процесса (рис. 4.21).

В сети на рис. 4.20 количество параллельных процессов задавалось ее структурой, и по

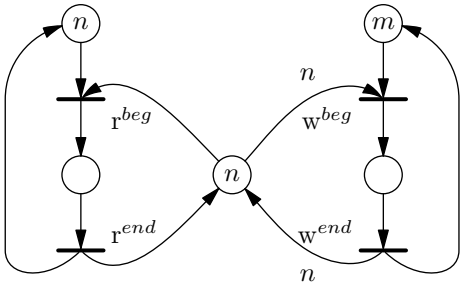



Рис. 4.21. Задача чтения-записи с точки зрения состояния ресурса

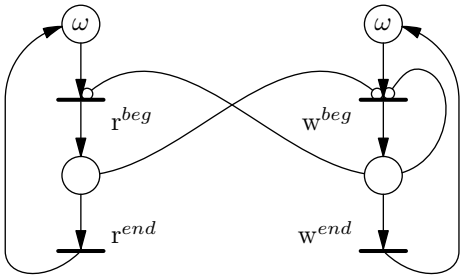



Рис. 4.22. Ингибиторная сеть для неограниченного числа процессов

этой причине было жестко фиксировано. В последнем же случае возникает естественное стремление уйти от ограничений на число процессов. Однако мы видим, что максимально возможное количество читающих процессов снова отражается в структуре сети (количество фишек в позиции доступности ресурса и кратность дуг от нее к пишущим процессам и обратно), в связи с чем такая сеть снова решает лишь задачу с ограниченным числом читающих процессов. Задача чтения-записи с неограниченным количеством читающих процессов классическими сетями Петри не решается. На практике это обычно не является существенным препятствием, поскольку в реальности всегда можно указать недостижимую верхнюю границу. Тем не менее, это один из простых и ярких примеров теоретической задачи, не решаемой классическими сетями Петри.

Многие существующие расширения классических сетей Петри охватывают более широкие классы задач. К примеру, задача чтения-записи с неограниченным количеством читающих процессов может быть решена с помощью ингибиторных сетей (рис. 4.22). В таких сетях допускается наличие ингибиторных дуг, т.е. таких входных дуг, при наличии которых переход считается разрешенным лишь в случае отсутствия фишек в соответствующих входных позициях [18, 14].

В примере на рис. 4.22 специальным символом $\omega$ обозначено бесконечное количество фишек в позиции. Такие позиции обычно можно без потери функциональности исключить из сети вместе с входящими и исходящими (не ингибиторными) дугами. В сети содержатся три ингибиторные дуги. Одна из них запрещает переход начала чтения, если в данный момент активна операция записи, две других запрещают начало записи, если активны запись или чтение. Сеть с аналогичной функциональностью может быть построена и с использованием лишь одной ингибиторной дуги, запрещающей запись в момент выполнения чтения, хотя такая сеть будет менее наглядна.

## 4.2. Программная реализация

Одним из подходов, используемых при внедрении сетей Петри в программирование, является отождествление переходов с операторами программы, а позиций — с их готовностью к выполнению. Здесь возникает соблазн пренебречь длительностью простых операций и представить программу в виде одноуровневой сети, формально содержащей только простые переходы. Однако в этом случае необходимо учитывать тот факт, что вследствие того, что срабатывания простых переходов последовательны, т.е. не могут перекрывать друг друга по времени, организованная таким образом программа не может быть параллельной. Поэтому при разработке параллельных программ мы будем рассматривать иерархические сети с составными переходами.

### 4.2.1. Функционирование строго иерархических сетей

При реализации механизма работы сети Петри мы будем исходить из того факта, что все срабатывания и соответствующие изменения разметки последовательны. Как следствие, выполнение сетей, в том числе иерархических, будет нами реализовано в виде последовательной программы. На основе этого механизма, однако, могут быть построены параллельные программы, процесс выполнения которых отражается выполнением соответствующих сетей.

В приложении Г приведен используемый в качестве примера исходный код, реализующий работу иерархических сетей. Среди описанных в нем классов объявляется класс позиции `place_type`, а также абстрактный класс перехода `transition_abstract_type`. На основе последнего строятся классы простого (`transition_simple_type`) и составного (`transition_compound_type`) переходов. Абстрактный класс перехода описывает следующий интерфейс:

```
// абстрактный тип перехода
class transition_abstract_type
{
public:
 // список разрешенных переходов
 typedef std::vector<transition_abstract_type *> enabledlist_type;
 // список помеченных позиций
 typedef std::map<place_type *, int> markedlist_type;

 // активация перехода
 virtual
 void activate(void) = 0;
 // получение информации, активен ли переход
 virtual
 bool is_active(void) const = 0;
 // получение списка внутренних разрешенных переходов
 // (только если текущий переход активен)
 virtual
 enabledlist_type get_enabled(void) const = 0;
 // получение списка внутренних помеченных позиций
 // (только если текущий переход активен)
 virtual
 markedlist_type get_marked(void) const = 0;
 // срабатывание одного из разрешенных внутренних переходов
 // (только если текущий переход активен)
 virtual
```

```
 void fire(int number) = 0;

 // обработчики событий
 virtual
 void on_activate(void) {}
 virtual
 void on_passivate(void) {}
};
```

Функция `activate` переводит переход в активное состояние. С помощью функции `is_active` может быть осуществлена проверка, находится ли переход все еще в активном состоянии. Если переход активен, т.е. имеет внутренние разрешенные переходы, список этих переходов возвращается функцией `get_enabled`. При этом возвращается полный список всех переходов, включая внутренние переходы активных составных переходов всех уровней вложенности. Также, если составной переход активен, с помощью функции `get_marked` может быть получено множество его внутренних помеченных позиций всех уровней вложенности. Наконец, функция `fire` осуществляет срабатывание конкретного перехода из списка, полученного с помощью функции `get_enabled`. На вход ей передается индекс перехода в этом списке. Функции `get_enabled`, `get_marked` и `fire` должны вызываться лишь в случае, когда функция `is_active` возвращает значение истины.

Помимо функций управления переходом, интерфейс `transition_abstract_type` описывает два обработчика, вызываемых при смене состояния перехода с пассивного на активное и обратно. Эти обработчики при желании могут быть переопределены в наследующих классах.

Класс простого перехода `transition_simple_type` предоставляет реализацию перечисленных функций с учетом специфики простого перехода. Простой переход формально активен лишь мгновение, т.е. при проверке состояния всегда пассивен. Помимо этого, простой переход не содержит вложенной сети, поэтому вызовы функций получения списков внутренних разрешенных переходов и помеченных позиций, а также функции срабатывания внутреннего перехода, бессмысленны.

Класс составного перехода `transition_compound_type` реализует работу одного уровня сети. Внутри него объявляется класс содержимого составного перехода `content_type`, который введен для удобства заполнения перехода внутренними элементами. Структуры, в которых удобно хранить информацию о внутренних элементах при заполнении, отличаются от тех, которые удобно использовать при выполнении составного перехода, вследствие чего в какой-то момент необходимо преобразование одних структур в другие. Именно эту задачу выполняет класс `content_type`, чтение и преобразование структур которого производится в момент создания объекта составного перехода.

Объект класса `content_type` хранит указатели на позиции и переходы сети вместе с их номерами в соответствии с порядком их добавления. Помимо этого, в нем хранится информация о дугах, а также о начальной разметке. Класс предоставляет клиентскому коду функции добавления позиции, перехода, входной и выходной дуг, а также помещения фишки в позицию. Для использования классом `transition_compound_type` предоставляются функции получения упорядоченных списков указателей на позиции и переходы, матриц входных и выходных дуг и вектора начальной разметки. Все эти данные передаются в соответствующие структуры объекта составного перехода в момент его создания.

В процессе выполнения составной переход (объект класса `transition_compound_type`) в каждый момент содержит текущий полный список внутренних разрешенных переходов `m_enabled` и полный список внутренних помеченных позиций `m_marked`. Для поддержания актуальности обоих этих списков используется функция `refresh`, о которой подробнее по-

говорим позже. Помимо этого, в объекте содержится вектор текущей разметки `m_marking`, а также две структуры данных, используемых для поиска размещения срабатывающего перехода.

Функция `activate` выставляет в составном переходе начальную разметку и заполняет списки внутренних разрешенных переходов и помеченных позиций. Если текущий список разрешенных переходов не пуст, функция `is_active` возвращает значение истины. Построенные списки разрешенных переходов и помеченных позиций могут быть получены с помощью функций `get_enabled` и `get_marked`.

Функция `fire` принимает в качестве параметра индекс срабатывающего перехода из списка `m_enabled`. В начале функции осуществляется определение размещения соответствующего перехода, а именно вычисление номера перехода текущего уровня (далее — локального перехода), к которому относится переданный индекс, и соответствующего индекса в полном списке его внутренних разрешенных переходов. Последний параметр имеет смысл, если соответствующий локальный переход окажется активным (а значит, составным). Если же он пассивен, значит, он просто разрешен и представлен в списке `m_enabled` лишь одним элементом.

После определения размещения перехода осуществляются действия по срабатыванию. Если переход активен, то вычисленная позиция в его внутреннем списке передается его функции срабатывания. Если же пассивен, выполняется изъятие фишек из его входных позиций и активация перехода. В этот момент пассивный переход может стать активным или же остаться пассивным, если это простой переход. Активный переход после осуществления внутреннего срабатывания также может остаться активным или завершить работу и перейти в пассивное состояние. Поэтому следующим этапом выполняется проверка активности локального перехода. Если переход пассивен, осуществляются действия по деактивации, и фишки помещаются в выходные позиции. В конце функции срабатывания перехода осуществляется обновление текущих списков разрешенных переходов и помеченных позиций.

Хранение полных списков внутренних разрешенных переходов и помеченных позиций нужно для того, чтобы пересчитывать их содержимое лишь в случаях, когда меняется внутренняя разметка соответствующего составного перехода. Это позволяет не производить при каждом срабатывании полный пересчет для всей многоуровневой сети, поскольку те составные переходы, которых текущее срабатывание не коснулось, просто возвращают списки, подготовленные ранее. Формирование хранимых списков `m_enabled` и `m_marked` осуществляет функция `refresh`, которая вызывается из функции активации составного перехода и после каждого срабатывания внутреннего перехода.

Полный список разрешенных переходов формируется функцией `refresh` в виде последовательности из локальных разрешенных переходов и внутренних списков каждого локального активного перехода. Если переход активен, он не может сработать, даже если разрешен. В то же время, если он активен, значит, он составной и содержит непустой список внутренних разрешенных переходов. Поэтому в список вносится либо сам рассматриваемый переход (если он разрешен и пассивен), либо его внутренние разрешенные переходы (если он активен).

В процессе построения полного списка разрешенных переходов осуществляется также сохранение в векторе `m_location` номеров локальных переходов, которым соответствуют конкретные элементы полного списка. В векторе `m_offset` сохраняются смещения областей каждого локального перехода от начала полного списка. Обе эти структуры нужны для локализации срабатывающего перехода в процессе выполнения функции `fire` по его индексу в полном списке `m_enabled`.

Помимо этого, функция `refresh` подготавливает список помеченных позиций. В него попадают все позиции текущего уровня с ненулевой разметкой, а также внутренние списки

помеченных позиций локальных активных переходов.

Факт разрешенности какого-либо пассивного перехода определяется на основе матрицы входных дуг и текущей разметки. В нашем случае производится полная проверка разрешенности для всего набора локальных пассивных переходов, что оказывается довольно тяжеловесной операцией в больших сетях. Следует заметить, что количество вычислений можно сократить, если после каждого срабатывания рассматривать лишь те переходы, для которых изменилась разметка входных позиций. Мы не приводим здесь такое усовершенствование, чтобы не усложнять и без того громоздкий код. Помимо этого, эффективность может быть повышена путем изменения структур данных, хранящих информацию о дугах. В нашем случае используются матрицы входных и выходных дуг. В большинстве случаев эти матрицы оказываются сильно разреженными, вследствие чего путем изменения способа их хранения могут быть сокращены затраты как памяти, так и времени выполнения.

При рассмотренном подходе приходится мириться с тем, что смена разметки сети при срабатывании перехода не является мгновенной операцией, и само по себе программное выполнение большой сети может занимать значительное время. Однако в случае строго иерархических сетей отдельные составные переходы с крупными вложенными сетями, как уже говорилось выше, могут быть представлены в виде составного перехода, изображенного на рис. 4.7 и содержащего простую сеть, срабатывание единственного перехода которой говорит о завершении работы исходной вложенной сети. Это позволяет возложить задачу выполнения чересчур сложных составных переходов в виде автономных сетей на другие параллельные ресурсы. Вопросы конкретной реализации параллельного выполнения будут рассмотрены ниже.

Наконец, класс `petrinet_type` реализует работу всей иерархической сети. Поскольку вся сеть является частным случаем составного перехода, класс наследуется от `transition_compound_type`. Класс `petrinet_type` определяет функцию выполнения жизненного цикла сети `live`, в которой осуществляется активация соответствующего составного перехода и последовательность срабатываний его внутренних переходов до тех пор, пока он не перестанет быть активным. Выбор срабатывающего внутреннего перехода каждый раз осуществляется извне с помощью переданного объекта окружения, реализующего интерфейс `environment_abstract_type`. Этот интерфейс содержит описание одной функции, которая принимает полный список разрешенных переходов сети, ожидает срабатывания любого из них и возвращает индекс сработавшего перехода из переданного списка. Также в эту функцию передается информация о помеченных позициях сети, что может быть удобно для анализа или отображения текущего состояния вне сети.

Классы окружения, реализующие интерфейс `environment_abstract_type`, могут быть довольно разными, в зависимости от решаемой задачи. Одним из самых простых вариантов, зачастую используемых при моделировании, является ожидание срабатывания на основе случайного выбора. Каждому переходу может быть сопоставлена своя интенсивность срабатываний, на основе которой вычисляется вероятность срабатывания среди конкретного набора разрешенных переходов. В качестве примера мы приведем вариант реализации для случая с одинаковой интенсивностью срабатывания у всех переходов:

```
// класс окружения с произвольным выбором перехода
class randomenv_type: public petrinet_type::environment_abstract_type
{
public:
 // генерация псевдослучайного числа в интервале [0, bound)
 static
 int random(int bound)
 {
  assert(bound > 0);
```

```
  return int(rand() / (RAND_MAX + 1.0) * bound);
 }
 // срабатывает произвольный переход
 int wait(
  const petrinet_type::enabledlist_type &enabled,
  const petrinet_type::markedlist_type &marked)
 {
  return random(enabled.size());
 }
};
```

С использованием описанных классов создание и выполнение сети Петри, изображенной на рис. 4.1, реализуется следующим кодом:

```
randomenv_type env;

// объекты–позиции
place_type p1, p2, p3, p4, p5, p6;
// объекты–переходы
transition_1_type t1;
transition_2_type t2;
transition_3_type t3;

// заполнение содержимого сети Петри
petrinet_type::content_type content;
// внесение позиций
content.add_place(p1);
content.add_place(p2);
content.add_place(p3);
content.add_place(p4);
content.add_place(p5);
content.add_place(p6);
// внесение переходов
content.add_transition(t1);
content.add_transition(t2);
content.add_transition(t3);
// внесение входных дуг
content.add_arc(p1, t1);
content.add_arc(p2, t2);
content.add_arc(p3, t2);
content.add_arc(p3, t3);
content.add_arc(p4, t3);
// внесение выходных дуг
content.add_arc(t1, p3);
content.add_arc(t1, p4);
content.add_arc(t2, p5);
content.add_arc(t3, p5);
content.add_arc(t3, p6);
// помещение фишек в позиции
content.add_token(p1, 2);
content.add_token(p2);
// создание сети Петри
petrinet_type petrinet(content);

// выполнение сети Петри
petrinet.live(env);
```

Здесь были использованы типы переходов, унаследованные от класса простого перехода, переопределяющие обработчик активации и используемые для отслеживания последовательности срабатываний:

```
class transition_1_type: public transition_simple_type
{
 void on_activate(void)
 {
  synprintf(stdout, "transition 1 fires\n");
 }
};
// ...
```

Описанные классы позволяют построение и выполнение строго иерархических сетей. К примеру, сеть, изображенная на рис. 4.7, строится и выполняется следующим образом:

```
class transition_stop_type: public transition_simple_type
{
 void on_activate(void)
 {
  synprintf(stdout, "transition stop fires\n");
 }
};

// ...

randomenv_type env;

// создание и заполнение составного перехода
place_type started, stopped;
transition_stop_type stop;
transition_compound_type::content_type cnt;
cnt.add_place(started);
cnt.add_place(stopped);
cnt.add_transition(stop);
cnt.add_arc(started, stop);
cnt.add_arc(stop, stopped);
cnt.add_token(started);
transition_compound_type subnet(cnt);

// создание и заполнение сети с составным переходом
petrinet_type::content_type content;
place_type begin, end;
content.add_place(begin);
content.add_place(end);
content.add_transition(subnet);
content.add_arc(begin, subnet);
content.add_arc(subnet, end);
content.add_token(begin);
petrinet_type petrinet(content);

// выполнение сети
petrinet.live(env);
```

Здесь вначале осуществляется создание и наполнение составного перехода, после чего он используется при создании и наполнении объемлющей сети. При выполнении иерархиче-

ской сети срабатывания разрешенных переходов происходят на всех уровнях вложенности.

### 4.2.2. Выполнение параллельных процессов

Для реализации параллельных программ мы будем отождествлять длительные операции с составными переходами, аналогичными изображенному на рис. 4.7. Предложенный ранее вариант окружения, в котором ожидание срабатывания основывается на случайном выборе перехода, в данном случае не подходит. Причиной является то, что каждый составной переход, соответствующий параллельно выполняемой длительной операции, содержит один простой переход, срабатывающий не произвольно, а вследствие завершения ее выполнения.

Интерфейсы OpenMP и MPI не обеспечивают достаточной гибкости, необходимой для удобства реализации параллельного выполнения программ на основе сетей Петри, поэтому обратимся к более низкоуровневым средствам. Для примера мы рассмотрим многопоточное распараллеливание, хотя с использованием сетевых коммуникаций может быть выполнено и распараллеливание в системах с распределенной памятью.

#### Многопоточная реализация на основе Windows API

Произведем распараллеливание выполнения длительных переходов между различными потоками с помощью интерфейса Windows API. Для этого реализуем новый класс окружения, который должен обеспечивать создание новых потоков для выполнения длительных операций и отслеживать их завершение в рамках функции ожидания срабатывания переходов. Ниже приведен код такого класса окружения:

```
// класс окружения с поддержкой многопоточного выполнения
class threadenv_type: public randomenv_type
{
public:
 // абстрактный тип работы, выполняемой во время длительного перехода
 class longjob_abstract_type
 {
 public:
  // функция выполнения работы
  virtual
  void run(void) = 0;
 };

private:
 // переход завершения длительной операции
 class transition_stop_type: public transition_simple_type
 {
 private:
  // номер соответствующего длительного перехода
  int m_id;
 public:
  transition_stop_type(int id): m_id(id) {}
  int id(void) const { return m_id; }
 };

 // структура параметров потока
 struct threadparam_type
 {
  // указатель на соответствующую работу
```

```
  longjob_abstract_type *pjob;
 };

 // структура данных, связываемая с каждым длительным переходом
 struct jobdata_type
 {
  // содержимое соответствующего длительного перехода
  place_type started, stopped;
  transition_stop_type stop;
  // выполняющий длительную работу поток
  HANDLE thread;
  // параметры потока
  threadparam_type param;
  // конструктор
  jobdata_type(int id, const threadparam_type &p): stop(id), param(p) {}
 };

public:
 // длительный переход, характеризующий выполнение работы
 class transition_long_type: public transition_compound_type
 {
 private:
  // окружение, с которым связан переход
  threadenv_type *m_penv;
  // номер перехода в списке длительных переходов окружения
  int m_id;

  // вызовы инициализации и финализации выполнения работы окружением
  void on_activate(void)
  {
   m_penv->initialize_longjob(m_id);
  }
  void on_passivate(void)
  {
   m_penv->finalize_longjob(m_id);
  }

 public:
  // конструктор длительного перехода
  // принимает ссылки на выполняемую работу и связанное окружение
  // первоначально создает пустой переход (без содержимого)
  transition_long_type(longjob_abstract_type &longjob,threadenv_type &env):
   transition_compound_type(content_type()), m_penv(&env)
  {
   // размещение данных длительного перехода
   std::pair<int, jobdata_type *> p = env.allocate_longjob(longjob);
   m_id = p.first;
   // формирование содержимого составного перехода
   jobdata_type &jobdata = *p.second;
   transition_compound_type::content_type content;
   content.add_place(jobdata.started);
   content.add_place(jobdata.stopped);
   content.add_transition(jobdata.stop);
   content.add_arc(jobdata.started, jobdata.stop);
   content.add_arc(jobdata.stop, jobdata.stopped);
```

```
    content.add_token(jobdata.started);
    // замена текущего содержимого составного перехода
    transition_compound_type::operator =(
     transition_compound_type(content));
   }
  };

private:
  // структуры данных длительных переходов
  // в векторе хранить нельзя, поскольку тогда при добавлении
  // нового элемента становятся невалидными указатели на старые
  std::deque<jobdata_type> m_alljobdata;

  // функция потока - выполняет вызов функции выполнения работы
  static
 DWORD WINAPI thr_proc(LPVOID param)
  {
   threadparam_type &p = *static_cast<threadparam_type *>(param);
   // выполнение работы
   p.pjob->run();
   return 0;
  }

  // запуск выполнения работы
  void initialize_longjob(int id)
  {
   // создание потока
   DWORD dwId;
   m_alljobdata[id].thread = ::CreateThread(
    NULL, 0,
    thr_proc, &m_alljobdata[id].param,
    0, &dwId);
   assert(m_alljobdata[id].thread != NULL);
  }
  // заключительные действия после выполнения работы
  void finalize_longjob(int id)
  {
   // освобождение ресурсов
   // к этому моменту поток уже закончил работу
   ::CloseHandle(m_alljobdata[id].thread);
  }

  // размещение структуры данных длительного перехода
  // возвращает присвоенный номер и указатель на структуру
  std::pair<int, jobdata_type *> allocate_longjob(
   longjob_abstract_type &longjob)
  {
   // номер создаваемого длительного перехода
   int id = m_alljobdata.size();

   // заполним структуру параметров потока
   threadparam_type param = {};
   param.pjob = &longjob;
   // разместим данные длительного перехода
   m_alljobdata.push_back(jobdata_type(id, param));
```

```
  // вернем номер перехода и указатель на его данные
  return std::make_pair(id, &m_alljobdata.back());
 }

public:
 // ожидание срабатывания перехода
 int wait(
  const petrinet_type::enabledlist_type &enabled,
  const petrinet_type::markedlist_type &marked)
 {
  // списки индексов переданных переходов
  // busy - переходы завершения работ, которые еще не завершены
  // finished - переходы завершения работ, которые уже завершены
  // free - переходы, не являющиеся переходами завершения работ
  std::vector<int> busy, finished, free;
  // хэндлы потоков, связанных с незавершенными работами
  std::vector<HANDLE> hdls;
  // проходим по всему списку переходов
  petrinet_type::enabledlist_type::const_iterator it;
  for (it = enabled.begin(); it != enabled.end(); ++it)
  {
   transition_stop_type *stop = dynamic_cast<transition_stop_type *>(*it);
   // если это не переход завершения работы, кладем индекс в свободные
   if (!stop)
    free.push_back(std::distance(enabled.begin(), it));
   else
   {
    // иначе получаем хэндл связанного потока
    HANDLE hdl = m_alljobdata[stop->id()].thread;
    // и проверяем, завершился ли он
    if (::WaitForSingleObject(hdl, 0) == WAIT_OBJECT_0)
     finished.push_back(std::distance(enabled.begin(), it));
    else
    {
     // если не завершился, добавляем информацию для ожидания
     busy.push_back(std::distance(enabled.begin(), it));
     hdls.push_back(hdl);
    };
   };
  };
  // формируем код возврата на основе приоритетов
  int rc;
  // если есть завершившиеся работы, возвращаем
  // индекс одного из соответствующих переходов
  if (!finished.empty())
   rc = finished[random(finished.size())];
  // если есть свободные переходы, вернем индекс одного из них
  else if (!free.empty())
   rc = free[random(free.size())];
  else
  {
   // в остальных случаях дождемся завершения одного из потоков
   DWORD dw = ::WaitForMultipleObjects(
    hdls.size(), &hdls.front(), FALSE, INFINITE);
   assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdls.size());
```

```
    // и вернем индекс соответствующего перехода
    rc = busy[dw - WAIT_OBJECT_0];
   };
   return rc;
  }
};
```

В начале приведенного класса объявляются используемые типы. Среди них интерфейс `longjob_abstract_type` длительной работы, выполняемой в рамках одного срабатывания длительного перехода, несколько служебных структур и класс длительного перехода `transition_long_type`. Последний унаследован от класса составного перехода и определяет обработчики событий активации и деактивации, один из которых инициирует начало выполнения длительной операции в параллельном потоке, другой выполняет завершающие действия. Эти операции непосредственно реализует объект окружения, класс же длительного перехода лишь передает ему необходимые для их выполнения данные.

С каждым длительным переходом сети ассоциирована структура данных `jobdata_type`. Она содержит объекты внутренних элементов составного перехода (рис. 4.7), а также данные, необходимые для взаимодействия с выполняющим длительную операцию потоком. Создание такой структуры и добавление ее к хранимому окружением контейнеру `m_alljobdata` производится функцией окружения `allocate_longjob`. Для идентификации длительного перехода в рамках окружения используется его номер в порядке добавления. Внутренний переход `stop`, характеризующий завершение выполнения длительной операции, представлен классом `transition_stop_type`, в задачи которого входит хранение номера соответствующего длительного перехода. Этот номер используется для идентификации завершившего выполнение длительного перехода в функции ожидания срабатывания.

Функция окружения `initialize_longjob` создает новый поток на основе функции `thr_proc`, передавая ему в качестве параметра структуру типа `threadparam_type` с указателем на требующую выполнения работу. Функция `finalize_longjob`, соответственно, освобождает ресурсы после завершения потока.

Перечисленные три функции используются классом длительного перехода. В момент создания клиентским кодом объекта типа `transition_long_type` конструктор вызывает функцию `allocate_longjob`, после чего осуществляет заполнение составного перехода элементами из созданной структуры `jobdata_type` в соответствии с рис. 4.7. В процессе выполнения сети при каждой активации составного перехода вызывается соответствующий обработчик, инициирующий выполнение длительной операции в параллельном потоке.

Функция ожидания срабатывания перехода `wait`, прежде всего, анализирует весь список переданных переходов и делит их на две группы: переходы, не являющиеся переходами завершения длительных операций, и, соответственно, являющиеся ими. Для каждого перехода завершения определяется факт завершения выполнения соответствующего потока. В зависимости от результата эти переходы также делятся на две группы. Таким образом, все переходы делятся на три группы, после чего осуществляется выбор срабатывающего перехода.

На первом этапе проверяется множество переходов завершения, для которых соответствующие потоки уже завершены. Если оно не пусто, возвращается произвольный индекс из этого множества, поскольку такие переходы должны обрабатываться в первую очередь. В противном случае проверяется множество переходов, не являющихся переходами завершения длительных операций, поскольку они в нашем случае не требуют ожидания. Если оно не пусто, из него возвращается произвольный индекс. По сути, таким образом мы реализуем приоритеты срабатывания: переходы завершения длительных операций, которые уже, действительно, завершены, имеют приоритет над другими переходами. Наконец, если

все разрешенные переходы являются переходами завершения, потоки которых до сих пор выполняются, осуществляется ожидание завершения любого из них, после чего возвращается индекс соответствующего перехода.

При такой организации приоритетов в функции `wait` простые операции (к примеру, синхронизация доступа к критическим ресурсам, начало выполнения длительных операций и т.п.) будут обрабатываться довольно быстро (без ожидания). Ожидание же будет осуществляться лишь в случаях, когда нет возможности срабатывания каких-либо других переходов, кроме переходов завершения выполняющихся длительных операций, что позволяет строить достаточно эффективные параллельные программы.

### Реализация на основе интерфейса pthreads

Реализуем теперь код окружения с аналогичной функциональностью с помощью альтернативного программного интерфейса pthreads. За основу возьмем код приведенного выше класса `threadenv_type`. Создание и уничтожение потоков на основе интерфейса pthreads производится схожим образом. Основная возникающая сложность заключается в том, что в этом интерфейсе нет простой возможности ожидания завершения любого из нескольких потоков, вследствие чего такая функциональность будет реализована нами «вручную». Для этого в классе окружения добавим два объекта синхронизации:

```
class threadenv_type: public randomenv_type
{
 // ...
private:
 // объекты синхронизации для ожидания завершения потоков
 pthread_mutex_t m_mutex;
 pthread_cond_t m_cond;
 // номера последних завершившихся потоков
 std::vector<int> m_lastid;

 // проверка кода возврата функций pthread_xxx
 static
 void chkzero(int retcode) { assert(retcode == 0); }
 // ...
};
```

В этом фрагменте приведено также объявление функции проверки кода возврата для всех используемых функций интерфейса pthreads, а также контейнера, содержащего номера последних завершившихся длительных переходов. Этот контейнер требуется нам для обеспечения функциональности, аналогичной возврату номера завершившегося потока при ожидании завершения любого из них.

Использование объектов синхронизации требует их инициализации и уничтожения, для чего классу окружения требуется конструктор и деструктор:

```
class threadenv_type: public randomenv_type
{
 // ...
public:
 // вместе с объектом окружения создаются и уничтожаются
 // объекты синхронизации mutex и cond
 threadenv_type(void)
 {
  chkzero(::pthread_mutex_init(&m_mutex, NULL));
  chkzero(::pthread_cond_init(&m_cond, NULL));
```

```
 }
 ~threadenv_type(void)
 {
  chkzero(::pthread_cond_destroy(&m_cond));
  chkzero(::pthread_mutex_destroy(&m_mutex));
 }
 // ...
};
```

Функции потока на этот раз недостаточно передачи лишь указателя на выполняемую работу, поскольку она теперь должна взаимодействовать с объектами синхронизации, в связи с чем меняется содержимое структуры данных `threadparam_type`:

```
class threadenv_type: public randomenv_type
{
 // ...
 // структура параметров потока
 struct threadparam_type
 {
  // указатель на соответствующую работу
  longjob_abstract_type *pjob;
  // окружение
  threadenv_type *penv;
  // номер в списке длительных переходов окружения
  int id;
  // признак завершения выполнения
  bool finished;
 };

 struct jobdata_type
 {
  // ...
  // выполняющий длительную работу поток
  pthread_t thread;
  // ...
 };
 // ...
};
```

Соответственно, в функции `allocate_longjob` пополняется инициализация этой структуры:

```
class threadenv_type: public randomenv_type
{
 // ...
 std::pair<int, jobdata_type *> allocate_longjob(
  longjob_abstract_type &longjob)
 {
  // ...
  // заполним структуру параметров потока
  threadparam_type param = {};
  param.pjob = &longjob;
  param.penv = this;
  param.id = id;
  // ...
 }
```

```
 // ...
};
```

В соответствии с требованиями используемого программного интерфейса изменяются функции запуска потока и последующего освобождения ресурсов:

```
class threadenv_type: public randomenv_type
{
 // ...
 // запуск выполнения работы
 void initialize_longjob(int id)
 {
  // установка признака - поток не завершен
  m_alljobdata[id].param.finished = false;
  // создание потока
  chkzero(::pthread_create(
   &m_alljobdata[id].thread, NULL,
   thr_proc, &m_alljobdata[id].param));
 }
 // заключительные действия после выполнения работы
 void finalize_longjob(int id)
 {
  // освобождение ресурсов
  // к этому моменту поток уже закончил работу
  chkzero(::pthread_join(m_alljobdata[id].thread, NULL));
 }
 // ...
};
```

Изменению подлежит и функция потока, которая теперь, помимо выполнения работы, должна сообщить функции ожидания о факте завершения, сохранив при этом номер завершившегося длительного перехода:

```
class threadenv_type: public randomenv_type
{
 // ...
 // функция потока - выполняет вызов функции выполнения работы
 static
 void *thr_proc(void *param)
 {
  threadparam_type &p = *static_cast<threadparam_type *>(param);
  // выполнение работы
  p.pjob->run();
  // установка блокировки
  chkzero(::pthread_mutex_lock(&p.penv->m_mutex));
  // модификация признаков завершения потока
  p.finished = true;
  p.penv->m_lastid.push_back(p.id);
  // отправка сигнала о завершении потока
  chkzero(::pthread_cond_signal(&p.penv->m_cond));
  // снятие блокировки
  chkzero(::pthread_mutex_unlock(&p.penv->m_mutex));
  return NULL;
 }
 // ...
};
```

Наконец, меняется функция ожидания срабатывания перехода:

```
class threadenv_type: public randomenv_type
{
 // ...
 // ожидание срабатывания перехода
 int wait(
  const petrinet_type::enabledlist_type &enabled,
  const petrinet_type::markedlist_type &marked)
 {
  chkzero(::pthread_mutex_lock(&m_mutex));
  // списки индексов переданных переходов
  // finished - переходы завершения работ, которые уже завершены
  // free - переходы, не являющиеся переходами завершения работ
  std::vector<int> finished, free;
  // отображение номеров длительных переходов на индексы
  // переходов завершения работ, которые еще не завершены
  std::map<int, int> busybyid;
  // проходим по всему списку переходов
  petrinet_type::enabledlist_type::const_iterator it;
  for (it = enabled.begin(); it != enabled.end(); ++it)
  {
   transition_stop_type *stop = dynamic_cast<transition_stop_type *>(*it);
   // если это не переход завершения работы, кладем индекс в свободные
   if (!stop)
    free.push_back(std::distance(enabled.begin(), it));
   else
   {
    // иначе проверяем, завершился ли поток
    if (m_alljobdata[stop->id()].param.finished)
     finished.push_back(std::distance(enabled.begin(), it));
    else
     // если не завершился, добавляем информацию для ожидания
     busybyid[stop->id()] = std::distance(enabled.begin(), it);
   };
  };
  // формируем код возврата на основе приоритетов
  int rc;
  // если есть завершившиеся работы, возвращаем
  // индекс одного из соответствующих переходов
  if (!finished.empty())
   rc = finished[random(finished.size())];
  // если есть свободные переходы, вернем индекс одного из них
  else if (!free.empty())
   rc = free[random(free.size())];
  else
  {
   // в остальных случаях дождемся завершения одного из потоков
   m_lastid.clear();
   while (m_lastid.empty())
    chkzero(::pthread_cond_wait(&m_cond, &m_mutex));
   // и вернем индекс соответствующего перехода
   rc = busybyid[m_lastid[random(m_lastid.size())]];
  };
  chkzero(::pthread_mutex_unlock(&m_mutex));
  return rc;
```

```
 }
 // ...
};
```

В начале функции выполняется блокировка объекта `m_mutex`, в связи с чем становится допустимой проверка информации о статусе завершения длительных операций. При выполнении ожидания внутри функции `pthread_cond_wait` объект `m_mutex` временно освобождается, в результате чего какой-либо завершающийся поток сможет заблокировать его, передать номер соответствующего длительного перехода и сигнализировать объект `m_cond`. Перед возвратом из функции `pthread_cond_wait` объект `m_mutex` снова блокируется, после чего в конце функции `wait` блокировка снимается.

### Пример клиентского кода

В качестве примера использования описанного окружения реализуем на его основе параллельное выполнение двух длительных операций (рис. 4.8):

```
// длительная операция
class longjob_type: public threadenv_type::longjob_abstract_type
{
private:
 int m_num;

 void run(void)
 {
  synprintf(stdout, "job %d begin\n", m_num);
  // ... длительные вычисления
  synprintf(stdout, "job %d end\n", m_num);
 }

public:
 longjob_type(int num): m_num(num) {}
};

// ...

threadenv_type env;

// позиции и простые переходы
place_type started, stopped;
place_type ready1, ready2;
place_type done1, done2;
transition_simple_type fork, join;
// выполняемые длительные работы
longjob_type job1(1), job2(2);
// длительные переходы
threadenv_type::transition_long_type exec1(job1, env);
threadenv_type::transition_long_type exec2(job2, env);

// наполнение сети
petrinet_type::content_type content;
content.add_place(started);
content.add_place(stopped);
content.add_place(ready1);
content.add_place(ready2);
```

```
content.add_place(done1);
content.add_place(done2);
content.add_transition(fork);
content.add_transition(join);
content.add_transition(exec1);
content.add_transition(exec2);

// внесение дуг
content.add_arc(started, fork);
content.add_arc(join, stopped);
// первая ветвь
content.add_arc(fork, ready1);
content.add_arc(ready1, exec1);
content.add_arc(exec1, done1);
content.add_arc(done1, join);
// вторая ветвь
content.add_arc(fork, ready2);
content.add_arc(ready2, exec2);
content.add_arc(exec2, done2);
content.add_arc(done2, join);

// фишка в начальной позиции
content.add_token(started);

// создание и запуск сети
petrinet_type petrinet(content);
petrinet.live(env);
```

Выполнение обеих работ реализуется в рамках класса `longjob_type`. Объект окружения создается раньше, чем объекты длительных переходов, поскольку ссылка на него передается их конструкторам. Это необходимо по причине того, что внутренние элементы длительного перехода создаются и хранятся объектом окружения. Созданные длительные переходы добавляются в сеть так же, как и простые.

## 4.3. Некоторые примеры использования

Наконец, рассмотрим некоторые примеры использования сетей Петри в программировании. Первый пример иллюстрирует удобство их применения для моделирования и мониторинга работы систем с большим количеством условий. Остальные примеры реализуют на их основе параллельное выполнение некоторых процессов и их синхронизацию между собой.

### 4.3.1. Реализация игры в жанре «квест»

В качестве первого примера использования сетей Петри мы реализуем простой вариант компьютерной игры-головоломки. Этот пример, в целом, не имеет отношения к распараллеливанию вычислений. Тем не менее, он дает понять, каков вообще смысл использования сетей Петри в программировании. Довольно широко распространено мнение о том, что они являются исключительно инструментом моделирования, а не построения программ. Тем не менее, задачей многих программ по сути как раз и является моделирование каких-либо процессов. К примеру, компьютерные игры, как правило, являются сильно упрощенными моделями реальной жизни. Программы управления оборудованием зачастую описывают

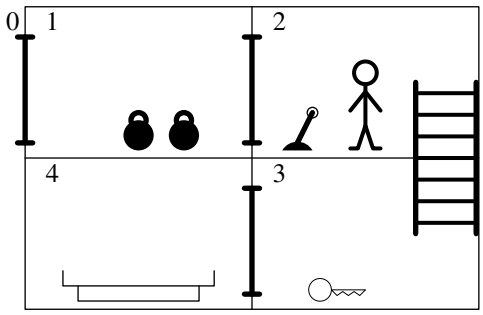



Рис. 4.23. Схема игры

некую виртуальную модель реального оборудования, в которую поступает информация с реальных датчиков и от которой исходят управляющие сигналы.

Пример с компьютерной игрой жанра «квест» иллюстративен тем, что довольно полно отражает характер задач, удачно описываемых и реализуемых в терминах сетей Петри. Такие игры обычно содержат множество элементов, пребывающих в тех или иных состояниях (могут быть описаны позициями с фишками или без). Герой, от лица которого происходит игра, вместе со своим положением в ней также может быть отождествлен с элементом в некотором состоянии. На элементы может быть осуществлено некоторое воздействие, в результате которого элемент может изменить состояние (срабатывание переходов, влияние на разметку входных и выходных позиций). Возможность осуществления воздействия на одни элементы может зависеть от состояния других (условия разрешенности перехода). Набор действий, доступных в каждый момент игроку, описывается общим состоянием игры на этот момент (список разрешенных переходов и разметка сети). Решение о том, какое действие выполнить, принимает игрок (срабатывающий переход и момент срабатывания определяются вне сети).

### Построение сети

Рассмотрим следующий простой пример игры (рис. 4.23). Герой пребывает в виртуальном мире, состоящем из четырех комнат. В момент начала игры он находится во второй комнате. Задача героя — покинуть мир через дверь, расположенную слева в первой комнате. Эта дверь заперта и открывается автоматически в случае, если на весах, расположенных в четвертой комнате, лежат две гири. Обе гири изначально лежат в первой комнате, при этом герой может носить с собой не более одной из них. Дверь между первой и второй комнатами также заперта и может быть открыта с помощью ключа, лежащего в третьей комнате. Подъемная дверь между третьей и четвертой комнатами открывается и закрывается с помощью рычага, расположенного во второй комнате.

Для простоты и наглядности размещение героя в разных частях комнаты и направление его взгляда не учитываются, учитывается лишь факт его пребывания в той или иной комнате. Таким образом, на протяжении всей игры герой может (при соответствующем стечении обстоятельств) выполнять следующие действия:

- перейти в комнату слева ($left$);
- перейти в комнату справа ($right$);
- подняться в комнату сверху ($up$);
- спуститься в комнату снизу ($down$);

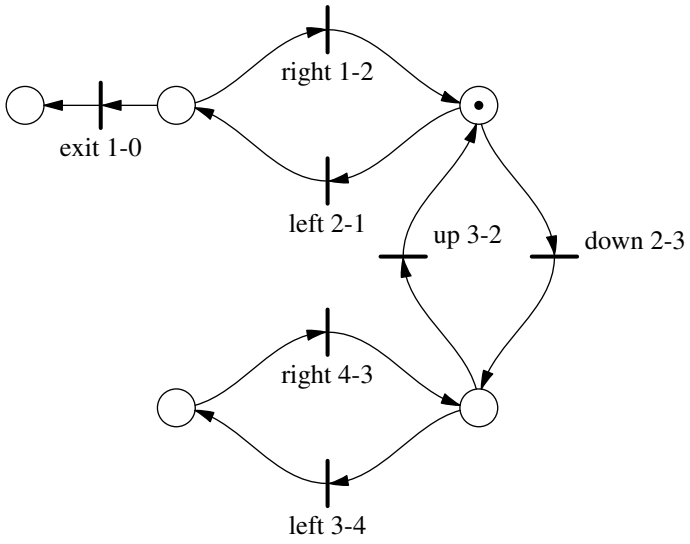



Рис. 4.24. Фрагмент сети, описывающий перемещение героя

- взять гирю с пола (*get*);
- положить гирю на весы (*put*);
- переключить рычаг (*switch*);
- найти и подобрать ключ (*find*);
- отпереть дверь (*open*);
- выйти наружу (*exit*).

Рассмотрим поочередно фрагменты будущей сети, отвечающие за функционирование различных элементов. Прежде всего, рассмотрим фрагмент, отвечающий за расположение героя (рис. 4.24). Герой может перемещаться из комнаты в комнату путем движений влево, вправо, вверх и вниз. Факт присутствия героя в той или иной комнате отражается фишкой в соответствующей позиции. Первоначально фишка размещена во второй позиции, отражая присутствие героя во второй комнате. По факту прохождения игры фишка помещается в нулевую позицию, при этом жизненный цикл сети завершается, поскольку не остается разрешенных переходов. В этом фрагменте пока не отражены зависимости возможности передвижения героя от состояния других элементов (в данном случае — закрытые двери). Эти зависимости мы внесем позже, пока же просто будем о них помнить.

Далее рассмотрим фрагменты сети для всех элементов управления. Самым простым является рычаг, открывающий и закрывающий дверь между третьей и четвертой комнатами (рис. 4.25). Он может быть в двух положениях, за что отвечают две соответствующие позиции. В каждый момент времени фишка может быть лишь в одной из них. Поскольку игрой предусмотрена возможность многократного переключения рычага, присутствуют переходы для перемещения фишки в обе стороны. Следует иметь в виду, что оба перехода *switch* зависят еще и от положения героя: чтобы у игрока появилась возможность переключения рычага, герой должен находиться во второй комнате.

Следующий фрагмент описывает процесс открывания двери между первой и второй комнатами (рис. 4.26). Для ее отпирания необходим ключ, который лежит в третьей комнате. Соответственно, фишка в первой позиции отражает тот факт, что ключ пока еще на

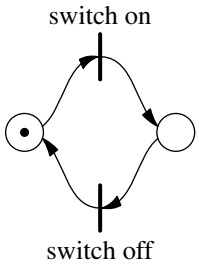



Рис. 4.25. Фрагмент сети, описывающий операции с рычагом

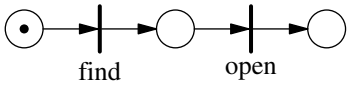



Рис. 4.26. Фрагмент сети, описывающий операции с дверью и ключом

полу. Во вторую позицию фишка перемещается, когда герой находит ключ (срабатывает переход $find$), т.е. она характеризует наличие ключа у героя. Чтобы поднять ключ, герою необходимо находиться в третьей комнате. Наконец, если герой находится во второй комнате, и у него есть ключ, вследствие срабатывания перехода *open* фишка из второй позиции может переместиться в последнюю: герой отпирает дверь.

Последний фрагмент описывает взаимодействие героя с гирями (рис. 4.27). Первая позиция отражает наличие гирь на полу в первой комнате. Изначально их там две, о чем говорит соответствующее количество фишек. Последняя позиция говорит о наличии гирь на весах в четвертой комнате. По ходу игры путем срабатывания переходов *get* и *put* обе фишки должны переместиться из первой позиции в последнюю. Снова следует учитывать не отраженные на фрагменте зависимости: переход *get* может сработать, лишь когда герой находится в первой комнате, переход *put* — лишь в четвертой комнате. Две промежуточные позиции говорят об отсутствии и наличии гири у героя. В позиции, говорящей об отсутствии гири, первоначально присутствует фишка. Эта позиция необходима, поскольку герой не должен иметь возможность нести сразу две гири (которая бы появилась при отсутствии этой позиции). Когда герой берет гирю, фишка из этой позиции изымается, в связи с чем герой не может снова взять гирю до тех пор, пока не положит первую.

Наконец, составляем из всех фрагментов полную сеть и вносим все указанные выше зависимости (рис. 4.28).

Переключение рычага в обе стороны может быть осуществлено лишь во второй комнате, в связи с чем оба перехода *switch* связаны входными дугами с соответствующей позицией. При переключении рычага герой остается во второй комнате, т.е. после срабатывания любого перехода *switch* фишка в этой позиции должна также оставаться. По этой причине оба перехода связываются с позицией второй комнаты еще и выходными дугами. Переход $find$ может сработать лишь в третьей комнате, переход *open* — лишь во второй, в связи с чем они также связаны с соответствующими позициями дугами в обоих направлениях. То же касается и переходов *get* и *put*, поскольку они могут сработать лишь когда герой находится в первой и четвертой комнатах соответственно.

Возможность выхода наружу и перемещения героя между комнатами также зависит от состояния остальных фрагментов. Перемещение из третьей комнату в четвертую зависит от положения рычага, при этом в момент перемещения положение рычага не меняется, поэтому соответствующий переход $left$ также связывается с нужной позицией дугами в

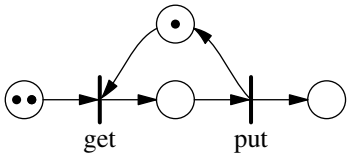



Рис. 4.27. Фрагмент сети, описывающий операции с гирями

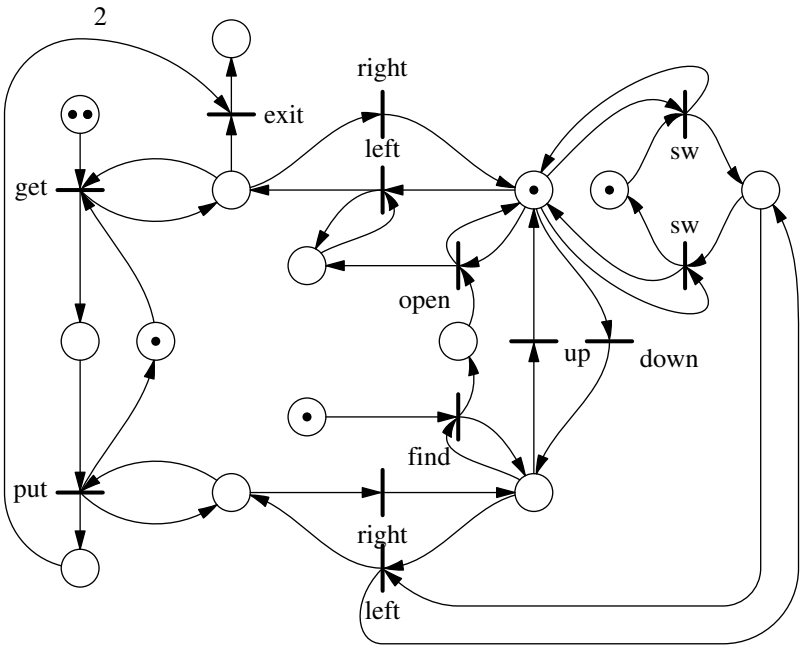



Рис. 4.28. Сеть Петри, полностью описывающая игру

обе стороны. То же касается и зависимости возможности перемещения из второй комнаты в первую от наличия фишки в позиции, характеризующей незапертость двери. Наконец, возможность выхода наружу зависит от наличия двух гирь на весах. Эта зависимость отражается дугой двойной кратности от соответствующей позиции. При достижении героем выхода сеть прекращает выполнение, поэтому возвращать фишки в эту позицию не требуется, и в обратную сторону дуг нет.

В построенной сети отражена зависимость возможности прохода из третьей комнаты в четвертую от состояния рычага, однако не внесена такая же зависимость для обратного прохода из четвертой в третью. В нашем случае ситуация запертой двери при обратном проходе недостижима (поскольку некому повернуть рычаг), поэтому мы можем себе позволить не нагружать сеть дополнительными связями. Однако в общем случае такие зависимости лучше сразу указывать, поскольку иначе легко допустить нарушение логики работы программы при внесении модификаций в сеть и соответствующем изменении достижимости разметок. К примеру, если позже будет введена возможность присутствия второго героя, который также может дернуть рычаг, или же рычаг будет сам поворачиваться обратно по прошествии некоторого времени, при отсутствии указанной зависимости первый герой сможет пройти через запертую дверь. То же относится и к переходу из первой комнаты во вторую.

При построении отдельных фрагментов сети мы держали неучтенные зависимости в голове, что может оказаться довольно утомительным при построении больших сетей. В

нашем случае это было сделано намеренно, чтобы нагляднее выделить каждый фрагмент сети, не обременяя его наличием позиций из других фрагментов. Однако более правильным подходом является явное указание в каждом фрагменте необходимых зависимостей вместе с соответствующими позициями и разметкой. После этого такие фрагменты могут быть объединены вместе с помощью операции наложения [18].

Построенная сеть довольно проста, вследствие чего ее визуальное представление оказалось вполне наглядным. Однако далеко не все задачи описываются столь удобными для отображения сетями. Это не должно пугать, поскольку графическое представление полной сети, как правило, вообще не требуется, и может быть интересно лишь в академических целях. Обычно при построении можно ограничиться графическим представлением лишь фрагментов сети и зависимостей от позиций других фрагментов. Для выполнения реализации этого оказывается вполне достаточно.

### Реализация

Для программной реализации игры нам необходимо новое окружение, в рамках которого будет «жить» соответствующая сеть Петри. При выполнении своих задач объект окружения будет взаимодействовать с объектами помеченных позиций и разрешенных переходов и получать от них некоторую информацию. По этим причинам нам потребуется определить для позиций и переходов новые классы, унаследованные от классов `place_type` и `transition_simple_type`. В такой ситуации допустимо определить для каждой позиции и каждого перехода свой класс, однако в нашем случае это даст больше сложностей, чем преимуществ, поэтому мы определим один класс `gstate_type` для всех позиций сети (различных состояний элементов игры) и один класс `gaction_type` для всех переходов (выполняемых в процессе игры действий):

```
// позиция сети Петри (состояние какого-либо элемента в игре)
class gstate_type: public place_type
{
private:
 int type;
public:
 enum {
  OUTSPACE,     // герой снаружи
  ROOM1,        // герой в комнате 1
  ROOM2,        // герой в комнате 2
  ROOM3,        // герой в комнате 3
  ROOM4,        // герой в комнате 4
  SWOFF,        // рычаг в положении "выкл"
  SWON,         // рычаг в положении "вкл"
  KLYING,       // ключ на полу
  KTAKEN,       // ключ у героя
  KUSED,        // герой использовал ключ
  DLYING,       // гиря на полу
  DTAKEN,       // гиря взята героем
  DHAVENO,      // у героя нет гири (может взять)
  DPLACED,      // гиря помещена на весы
  NUMBER        // полное количество позиций
 };
 explicit
 gstate_type(int t): type(t)
 {
  assert(type >=0 && type < NUMBER);
```

```
 }
 // ...
};

// переход сети Петри (какое-либо действие в игре)
class gaction_type: public transition_simple_type
{
private:
 int type;
public:
 enum {
  EXIT,          // покинуть помещение
  LEFT21,        // пройти влево из 2 комнаты в 1
  RIGHT12,       // пройти вправо из 1 комнаты во 2
  LEFT34,        // пройти влево из 3 комнаты в 4
  RIGHT43,       // пройти вправо из 4 комнаты в 3
  UP32,          // подняться из 3 комнаты во 2
  DOWN23,        // спуститься из 2 комнаты в 3
  TURNON,        // повернуть рычаг в положение "вкл"
  TURNOFF,       // повернуть рычаг в положение "выкл"
  GETKEY,        // взять ключ
  USEKEY,        // отпереть дверь
  GETDMB,        // взять гирю
  PUTDMB,        // положить гирю на весы
  NUMBER         // полное количество переходов
 };
 explicit
 gaction_type(int t): type(t)
 {
  assert(type >=0 && type < NUMBER);
 }
 // ...
};
```

Все позиции нумеруются с нуля, соответствующий номер передается конструктору `gstate_type` при создании объекта позиции. То же самое можно сказать и в отношении объектов `gaction_type`. На основе своего номера объекты позиций и переходов могут передавать объекту окружения ту или иную необходимую информацию. Объект окружения получает эту информацию путем использования указателей, переданных в списках помеченных позиций и разрешенных переходов:

```
// окружение сети Петри (взаимодействие игры с игроком)
class genv_type: public petrinet_type::environment_abstract_type
{
public:
 int wait(
  const petrinet_type::enabledlist_type &enabled,
  const petrinet_type::markedlist_type &marked)
 {
  // отображение игроку текущего состояния игры
  // (на основе содержимого marked)
  // ...

  // предложение игроку возможных вариантов действий
  // (на основе содержимого enabled)
```

```
  // ...

  // получение от игрока номера выбранного действия
  // int choice = ...

  return choice;
 }
};
```

Основными задачами объекта окружения являются:

- отображение игроку текущего состояния игры (положение героя в той или иной комнате, положение рычага, состояние дверей, наличие гири или ключа «в кармане» и т.п.) на основе текущей разметки сети;
- предоставление игроку вариантов допустимых в текущий момент действий (перемещение, переключение рычага, взятие предмета, использование предмета и т.п.) на основе списка разрешенных переходов;
- получение от игрока информации о выбранном действии, т.е. о том, какой из разрешенных переходов должен сработать.

Мы не приводим здесь детали отображения текущего состояния игры, поскольку они сильно зависят от платформы и предпочтений разработчика. Допустима графическая реализация, основанная на выборе текущего изображения из статического набора заранее отрисованных изображений всех возможных состояний игры на основе текущей разметки. Также возможен более сложный, но и более экономичный вариант динамического построения изображения на основе учета наличия фишек в отдельных позициях. Наконец, возможен и самый простой вариант, не использующий изображений вообще: игра в текстовом режиме, в котором производится лишь отображение набора строк, описывающих состояние различных элементов игры в соответствии с помеченными позициями. Аналогично, от платформы и замыслов разработчика полностью зависит и способ получения от игрока информации о выбранном им действии. Это может быть сложный анализ действий мышью или же просто получение номера выбранного действия с клавиатуры. Все эти детали реализации не имеют отношения к рассматриваемой теме, поэтому мы их в коде не приводим.

Наконец, рассмотрим код построения и запуска сети Петри в соответствии с рис. 4.28:

```
// заполнение содержимого сети Петри
petrinet_type::content_type content;

// внесение объектов-позиций
deque<gstate_type> p;
for (int i = 0; i < gstate_type::NUMBER; ++i)
{
 p.push_back(gstate_type(i));
 content.add_place(p.back());
};
// внесение объектов-переходов
deque<gaction_type> t;
for (int i = 0; i < gaction_type::NUMBER; ++i)
{
 t.push_back(gaction_type(i));
 content.add_transition(t.back());
```

```
};

// дуги фрагмента перемещения героя
content.add_arc(p[gstate_type::ROOM1], t[gaction_type::RIGHT12]);
content.add_arc(t[gaction_type::RIGHT12], p[gstate_type::ROOM2]);
content.add_arc(p[gstate_type::ROOM2], t[gaction_type::LEFT21]);
content.add_arc(t[gaction_type::LEFT21], p[gstate_type::ROOM1]);
content.add_arc(p[gstate_type::ROOM2], t[gaction_type::DOWN23]);
content.add_arc(t[gaction_type::DOWN23], p[gstate_type::ROOM3]);
content.add_arc(p[gstate_type::ROOM3], t[gaction_type::UP32]);
content.add_arc(t[gaction_type::UP32], p[gstate_type::ROOM2]);
content.add_arc(p[gstate_type::ROOM3], t[gaction_type::LEFT34]);
content.add_arc(t[gaction_type::LEFT34], p[gstate_type::ROOM4]);
content.add_arc(p[gstate_type::ROOM4], t[gaction_type::RIGHT43]);
content.add_arc(t[gaction_type::RIGHT43], p[gstate_type::ROOM3]);
content.add_arc(p[gstate_type::ROOM1], t[gaction_type::EXIT]);
content.add_arc(t[gaction_type::EXIT], p[gstate_type::OUTSPACE]);
// возможность перемещения в зависимости от состояния дверей
content.add_arc(p[gstate_type::KUSED], t[gaction_type::LEFT21]);
content.add_arc(t[gaction_type::LEFT21], p[gstate_type::KUSED]);
content.add_arc(p[gstate_type::SWON], t[gaction_type::LEFT34]);
content.add_arc(t[gaction_type::LEFT34], p[gstate_type::SWON]);
content.add_arc(p[gstate_type::DPLACED], t[gaction_type::EXIT], 2);

// дуги фрагмента переключения рычага
content.add_arc(p[gstate_type::SWOFF], t[gaction_type::TURNON]);
content.add_arc(t[gaction_type::TURNON], p[gstate_type::SWON]);
content.add_arc(p[gstate_type::SWON], t[gaction_type::TURNOFF]);
content.add_arc(t[gaction_type::TURNOFF], p[gstate_type::SWOFF]);
// зависимости возможности переключения от расположения героя
content.add_arc(p[gstate_type::ROOM2], t[gaction_type::TURNON]);
content.add_arc(t[gaction_type::TURNON], p[gstate_type::ROOM2]);
content.add_arc(p[gstate_type::ROOM2], t[gaction_type::TURNOFF]);
content.add_arc(t[gaction_type::TURNOFF], p[gstate_type::ROOM2]);

// дуги фрагмента операций с ключом
content.add_arc(p[gstate_type::KLYING], t[gaction_type::GETKEY]);
content.add_arc(t[gaction_type::GETKEY], p[gstate_type::KTAKEN]);
content.add_arc(p[gstate_type::KTAKEN], t[gaction_type::USEKEY]);
content.add_arc(t[gaction_type::USEKEY], p[gstate_type::KUSED]);
// зависимости от расположения героя
content.add_arc(p[gstate_type::ROOM3], t[gaction_type::GETKEY]);
content.add_arc(t[gaction_type::GETKEY], p[gstate_type::ROOM3]);
content.add_arc(p[gstate_type::ROOM2], t[gaction_type::USEKEY]);
content.add_arc(t[gaction_type::USEKEY], p[gstate_type::ROOM2]);

// дуги фрагмента операций с гирями
content.add_arc(p[gstate_type::DLYING], t[gaction_type::GETDMB]);
content.add_arc(p[gstate_type::DHAVENO], t[gaction_type::GETDMB]);
content.add_arc(t[gaction_type::GETDMB], p[gstate_type::DTAKEN]);
content.add_arc(p[gstate_type::DTAKEN], t[gaction_type::PUTDMB]);
content.add_arc(t[gaction_type::PUTDMB], p[gstate_type::DHAVENO]);
content.add_arc(t[gaction_type::PUTDMB], p[gstate_type::DPLACED]);
// зависимости от расположения
content.add_arc(p[gstate_type::ROOM1], t[gaction_type::GETDMB]);
```

```
content.add_arc(t[gaction_type::GETDMB], p[gstate_type::ROOM1]);
content.add_arc(p[gstate_type::ROOM4], t[gaction_type::PUTDMB]);
content.add_arc(t[gaction_type::PUTDMB], p[gstate_type::ROOM4]);

// начальная разметка
content.add_token(p[gstate_type::DLYING], 2);
content.add_token(p[gstate_type::DHAVENO]);
content.add_token(p[gstate_type::KLYING]);
content.add_token(p[gstate_type::SWOFF]);
content.add_token(p[gstate_type::ROOM2]);

// создание и выполнение сети Петри
genv_type env;
petrinet_type petrinet(content);
petrinet.live(env);
```

Вначале создаются и вносятся в содержимое сети все объекты позиций и переходов. В этот момент запоминаются лишь указатели на соответствующие элементы, в связи с чем внесенные объекты не должны перемещаться в памяти по мере создания и добавления новых. По этим причинам для хранения объектов позиций и переходов использованы контейнеры `std::deque`.

Следующим этапом осуществляется внесение полного набора дуг сети. Дуги добавляются последовательно по фрагментам (так же, как и строилась сеть). Первыми вносятся все дуги фрагмента перемещения героя (рис. 4.24) вместе с зависимостями от состояния дверей. Далее следует добавление дуг фрагментов взаимодействия с рычагом (рис. 4.25), ключом (рис. 4.26) и гирями (рис. 4.27). Во всех случаях сразу вносятся существующие внешние зависимости переходов, т.е. зависимости возможности выполнения соответствующих действий от присутствия героя в той или иной комнате.

В конце выполняется начальная разметка сети и запускается ее жизненный цикл. Работа программы завершается, когда игрок побеждает, т.е. когда герой выходит наружу, поскольку лишь в этом случае в сети не остается разрешенных переходов.

Приведенный код построения сети может пугать своей линейностью и объемом. Не стоит забывать, однако, что он приведен в таком виде лишь в иллюстративных целях, вообще же структура сети Петри, как правило, не представляется в программе статически, а загружается динамически на основе некоторых описывающих ее данных. К примеру, для подобных целей может быть использован язык разметки сетей Петри (PNML, Petri Net Markup Language).

### 4.3.2. Обработка потоков данных

В главе 2 мы рассматривали способ упорядочения и параллельного выполнения некоторого набора операций (комплекса работ) с учетом их зависимостей между собой по входным и выходным данным. Похожий механизм используется и в модели потоков данных (dataflow). Важное отличие заключается в том, что ярусно-параллельная форма программы (и, соответственно, последовательность выполнения операций) строилась нами предварительно на основе набора зависимостей операций между собой. В обработке же потоков данных управление запуском операций производится на основе готовности данных в соответствующих входных очередях, т.е. в процессе выполнения. Такая постановка позволяет производить выполнение комплексной задачи одновременно для большого количества исходных данных, внося параллелизм не только между работами яруса в рамках одной задачи, но и между разными ярусами различных задач (конвейерная обработка).

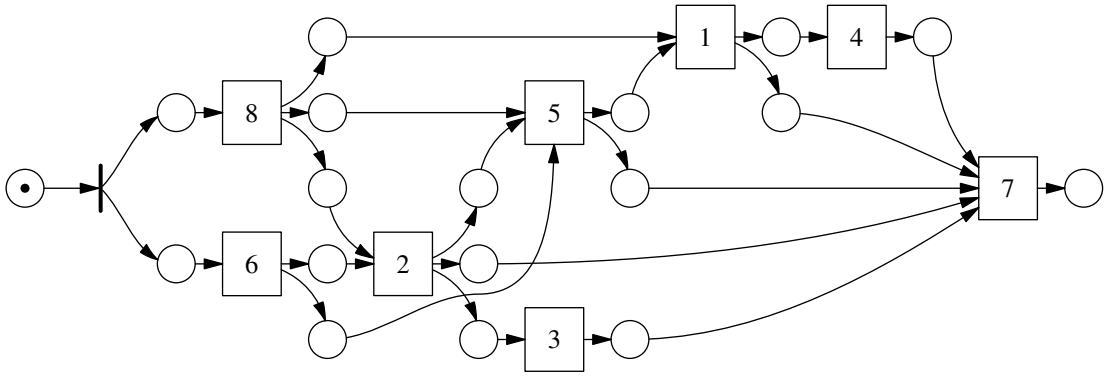



Рис. 4.29. Обработка потоков данных

Процесс обработки потоков данных может быть смоделирован с помощью потоковых сетей Петри [18]. В таких сетях переходам сопоставляются выполняемые операции, позициям — очереди входных данных для каждой операции. Наличие фишек в позициях говорит о готовности соответствующего количества элементов в очереди. К примеру, на рис. 4.29 изображена потоковая сеть, отражающая процесс выполнения комплекса работ, изображенного на рис. 2.4.

Очереди данных организованы по принципу FIFO (first in — first out). Благодаря этому, на протяжении всей работы сети порядок обработки данных не нарушается, и промежуточные данные разных комплексных операций не путаются между собой. Таким образом, если во входную очередь потоковой сети поместить несколько элементов исходных данных (несколько фишек в первой позиции), на выходе получим корректно упорядоченную очередь результатов. При большом количестве элементов во входной очереди все переходы сети потенциально могут работать одновременно. В некоторых случаях такая степень параллелизма может оказаться избыточной, в связи с чем в сеть могут быть внесены элементы ограничения параллелизма, к примеру, синхронизация доступа к каким-либо ресурсам.

Рассмотрим следующий простой пример. Допустим, есть некоторая система, состоящая из большого количества независимых подсистем с различными идентификаторами. От системы в любой момент может быть получена информация о текущем состоянии любой подсистемы, и на любую подсистему может быть осуществлено некоторое управляющее воздействие. В обоих случаях используется некоторый канал связи, по которому в один момент должно отправляться не более одного запроса. По каждой подсистеме программа хранит набор правил, на основе которых, исходя из текущего ее состояния, может быть вычислено управляющее воздействие на подсистему. Есть множество из $n$ идентификаторов подсистем, по каждому из которых требуется подготовить набор правил, получить текущее состояние подсистемы, вычислить требуемое управляющее воздействие и отправить эту информацию в систему.

Сеть, реализующая такую процедуру, изображена на рис. 4.30. При обработке каждого из $n$ идентификаторов осуществляется помещение его во входные очереди трех операций: *prepare* (подготовка набора правил), *get* (получение состояния подсистемы) и *post* (передача в систему управляющего воздействия). Результатом операции *prepare* является набор правил, который необходим на входе операции обработки состояния и вычисления управляющего воздействия (*process*). Также операция *process* требует на входе информацию о состоянии подсистемы, которую ей предоставляет операция *get*. Полученная в процессе выполнения *process* информация об управляющем воздействии передается на вход операции *post*, которая отсылает ее вместе с идентификатором подсистемы по каналу. Канал явля-

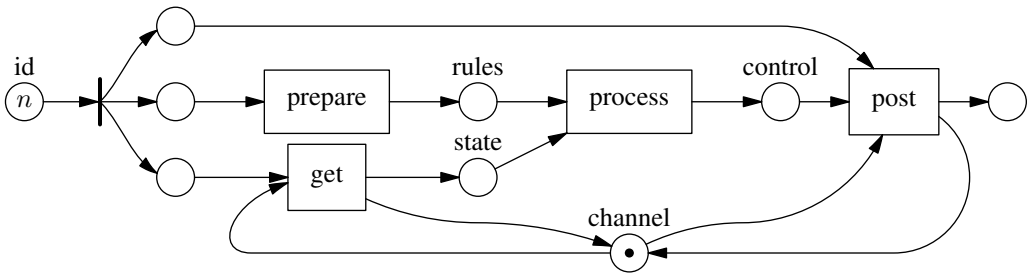



Рис. 4.30. Управление независимыми участками системы

ется критическим ресурсом, не допускающим параллельной обработки запросов, в связи с чем операции *get* и *post* связаны с позицией, обеспечивающей синхронизацию доступа к каналу.

Для случая $n = 1$ (один идентификатор во входной очереди) лишь первые две операции могут выполняться параллельно. Однако если $n$ велико, практически все время будут одновременно работать три операции из четырех. Путем более детального разбиения операций *prepare* и *process* может быть достигнута гораздо более высокая степень параллелизма.

Следующий код иллюстрирует построение и выполнение такой сети:

```
// хранилище очередей
struct queues_type
{
 // ...
};

// длительные операции
class jobprepare_type: public threadenv_type::longjob_abstract_type
{
private:
 queues_type &m_queues;
 void run(void)
 {
  synprintf(stdout, "prepare begin\n");
  // ... длительные вычисления
  synprintf(stdout, "prepare end\n");
 }
public:
 jobprepare_type(queues_type &queues): m_queues(queues) {}
};
class jobget_type: public threadenv_type::longjob_abstract_type
{
 // ...
};
class jobprocess_type: public threadenv_type::longjob_abstract_type
{
 // ...
};
class jobpost_type: public threadenv_type::longjob_abstract_type
{
 // ...
};
```

```
// ...

const int n = 10;

// очереди данных
queues_type queues;
// ... наполнение входной очереди идентификаторов

// выполняемые длительные работы
jobprepare_type jprepare(queues);
jobget_type jget(queues);
jobprocess_type jprocess(queues);
jobpost_type jpost(queues);

threadenv_type env;
// позиции
place_type id, id1, id2, id3;
place_type rules, state, control, result, channel;
// переходы
transition_simple_type split;
threadenv_type::transition_long_type prepare(jprepare, env);
threadenv_type::transition_long_type get(jget, env);
threadenv_type::transition_long_type process(jprocess, env);
threadenv_type::transition_long_type post(jpost, env);

// наполнение сети
petrinet_type::content_type content;
// позиции
content.add_place(id);
content.add_place(id1);
content.add_place(id2);
content.add_place(id3);
content.add_place(rules);
content.add_place(state);
content.add_place(control);
content.add_place(result);
content.add_place(channel);
// переходы
content.add_transition(split);
content.add_transition(prepare);
content.add_transition(get);
content.add_transition(process);
content.add_transition(post);
// дуги
content.add_arc(id, split);
content.add_arc(split, id1);
content.add_arc(split, id2);
content.add_arc(split, id3);
content.add_arc(id1, get);
content.add_arc(get, state);
content.add_arc(state, process);
content.add_arc(id2, prepare);
content.add_arc(prepare, rules);
content.add_arc(rules, process);
content.add_arc(process, control);
```

```
content.add_arc(id3, post);
content.add_arc(control, post);
content.add_arc(post, result);
content.add_arc(channel, get);
content.add_arc(get, channel);
content.add_arc(channel, post);
content.add_arc(post, channel);
// разметка
content.add_token(id, n);
content.add_token(channel);

// создание и запуск сети
petrinet_type petrinet(content);
petrinet.live(env);
```

В приведенном примере каждая длительная операция представляется отдельным классом. Предполагается, что помещение данных в очереди и извлечение из них производится атомарно в рамках реализации соответствующих функций `run`.

### 4.3.3. Реализация задачи об обедающих философах

Наконец, рассмотрим варианты реализации доступа к разделяемым ресурсам в соответствии с приведенным ранее описанием задачи об обедающих философах.

#### Одноуровневая сеть Петри

Прежде всего, рассмотрим вариант одноуровневой сети (рис. 4.15). Такая сеть потребует два типа переходов, отражающих начало еды и начало размышлений соответственно. Для каждого типа перехода мы реализуем отдельный класс:

```
// переход "приступает к еде"
class transition_eating_start_type: public transition_simple_type
{
private:
 int m_num;
 void on_activate(void)
 {
  synprintf(stdout, "philosopher %d begins eating\n", m_num);
 }
public:
 transition_eating_start_type(int num): m_num(num) {}
};

// переход "приступает к размышлениям"
class transition_eating_stop_type: public transition_simple_type
{
private:
 int m_num;
 void on_activate(void)
 {
  synprintf(stdout, "philosopher %d begins thinking\n", m_num);
 }
public:
 transition_eating_stop_type(int num): m_num(num) {}
};
```

Конструкторы обоих классов принимают на вход параметр — порядковый номер философа, используемый для отображения последовательности срабатываний. Ниже приводится код создания и выполнения соответствующей сети Петри:

```
randomenv_type env;

const int N = 5;
// заполняем содержимое сети
deque<place_type> eating(N), thinking(N), fork(N);
deque<transition_eating_start_type> start;
deque<transition_eating_stop_type> stop;
petrinet_type::content_type content;
for (int i = 0; i < N; ++i)
{
 // добавляем в сеть позиции
 content.add_place(eating[i]);
 content.add_place(thinking[i]);
 content.add_place(fork[i]);
 // создаем и добавляем в сеть переходы
 start.push_back(transition_eating_start_type(i));
 content.add_transition(start.back());
 stop.push_back(transition_eating_stop_type(i));
 content.add_transition(stop.back());
};
for (int i = 0; i < N; ++i)
{
 // входные и выходные дуги перехода start
 content.add_arc(thinking[i], start[i]);
 content.add_arc(fork[(i + N - 1) % N], start[i]);
 content.add_arc(fork[i], start[i]);
 content.add_arc(start[i], eating[i]);
 // входные и выходные дуги перехода stop
 content.add_arc(eating[i], stop[i]);
 content.add_arc(stop[i], fork[(i + N - 1) % N]);
 content.add_arc(stop[i], thinking[i]);
 content.add_arc(stop[i], fork[i]);
};
for (int i = 0; i < N; ++i)
{
 // кладем фишки
 content.add_token(thinking[i]);
 content.add_token(fork[i]);
};
// создаем и запускаем сеть
petrinet_type petrinet(content);
petrinet.live(env);
```

Следует учитывать, что такая сеть будет «жива» всегда, поэтому, фактически, в последней строке выполняется вечный цикл. Для обеспечения возможности выхода из него можно, к примеру, модифицировать сеть так, чтобы ограничить общее количество циклов еды.

### Иерархическая сеть

Теперь рассмотрим построение иерархической сети, изображенной на рис. 4.16. В этом случае будет использован другой тип перехода, который является составным. Мы определим его следующим образом:

```
// составной переход "философ ест"
class transition_eat_type: public transition_compound_type
{
private:
 int m_num;

 void on_activate(void)
 {
  synprintf(stdout, "philosopher %d begins eating\n", m_num);
 }
 void on_passivate(void)
 {
  synprintf(stdout, "philosopher %d begins thinking\n", m_num);
 }

public:
 transition_eat_type(const content_type &content, int num):
  transition_compound_type(content),
  m_num(num)
 {}
};
```

Обработчики этого перехода снова используются для отслеживания последовательности срабатываний, однако на этот раз срабатывают начало и завершение выполнения составного перехода. Каждый составной переход содержит вложенную сеть, совпадающую с изображенной на рис. 4.7 (на рис. 4.16 содержимое составных переходов опущено для компактности). Ниже следует код, реализующий создание и выполнение такой сети:

```
randomenv_type env;

const int N = 5;
// позиции сети верхнего уровня
deque<place_type> thinking(N), fork(N);
// внутренние позиции и переходы составных переходов
deque<place_type> started(N), stopped(N);
deque<transition_simple_type> stop(N);
// сами составные переходы
deque<transition_eat_type> eat;

// заполняем содержимое сети
petrinet_type::content_type content;
for (int i = 0; i < N; ++i)
{
 // добавляем в сеть позиции
 content.add_place(thinking[i]);
 content.add_place(fork[i]);

 // заполняем составной переход
 transition_compound_type::content_type cnt;
 cnt.add_place(started[i]);
```

```
 cnt.add_place(stopped[i]);
 cnt.add_transition(stop[i]);
 cnt.add_arc(started[i], stop[i]);
 cnt.add_arc(stop[i], stopped[i]);
 cnt.add_token(started[i]);
 eat.push_back(transition_eat_type(cnt, i));
 // и добавляем его в сеть
 content.add_transition(eat.back());
};
for (int i = 0; i < N; ++i)
{
 // входные дуги перехода eat
 content.add_arc(thinking[i], eat[i]);
 content.add_arc(fork[(i + N - 1) % N], eat[i]);
 content.add_arc(fork[i], eat[i]);
 // выходные дуги перехода eat
 content.add_arc(eat[i], thinking[i]);
 content.add_arc(eat[i], fork[(i + N - 1) % N]);
 content.add_arc(eat[i], fork[i]);
};
for (int i = 0; i < N; ++i)
{
 // кладем фишки
 content.add_token(thinking[i]);
 content.add_token(fork[i]);
};
// создаем и запускаем сеть
petrinet_type petrinet(content);
petrinet.live(env);
```

Вначале создаются и заполняются составные переходы, после чего они включаются в содержимое объемлющей сети. Во время выполнения сети происходят срабатывания длительных переходов *eat*, во время которых срабатывают их внутренние переходы *stop*.

### Параллельное выполнение длительных операций

До текущего момента «прием пищи» философами выполнялся параллельно лишь логически. Реализуем теперь физическое распараллеливание выполнения длительных переходов (рис. 4.16). Будем считать, что каждому философу во время однократного «приема пищи» требуется произвести некоторую длительную операцию (вычисления произвольной длительности). Для реализации этой операции создадим соответствующий класс длительной работы:

```
// длительная операция "философ ест"
class longjob_eating_type: public threadenv_type::longjob_abstract_type
{
private:
 int m_num;
 void run(void)
 {
  synprintf(stdout, "philosopher %d begins eating\n", m_num);
  // ... длительные вычисления
  synprintf(stdout, "philosopher %d begins thinking\n", m_num);
 }
public:
```

```
 longjob_eating_type(int num): m_num(num) {}
};
```

Код, реализующий создание и выполнение сети с параллельным выполнением таких работ, отличается от приведенного выше кода на основе составных переходов лишь в части, касающейся создания этих переходов:

```
threadenv_type env;

const int N = 5;
// позиции сети верхнего уровня
deque<place_type> thinking(N), fork(N);
// длительные работы
deque<longjob_eating_type> eatingjob;
// длительные переходы для выполнения работ
deque<threadenv_type::transition_long_type> eat;
// заполняем содержимое сети
petrinet_type::content_type content;
for (int i = 0; i < N; ++i)
{
 // добавляем в сеть позиции
 content.add_place(thinking[i]);
 content.add_place(fork[i]);
 // создаем и добавляем в сеть длительный переход
 eatingjob.push_back(longjob_eating_type(i));
 eat.push_back(
  threadenv_type::transition_long_type(eatingjob.back(), env));
 content.add_transition(eat.back());
};
for (int i = 0; i < N; ++i)
{
 // входные дуги перехода eat
 content.add_arc(thinking[i], eat[i]);
 content.add_arc(fork[(i + N - 1) % N], eat[i]);
 content.add_arc(fork[i], eat[i]);
 // выходные дуги перехода eat
 content.add_arc(eat[i], thinking[i]);
 content.add_arc(eat[i], fork[(i + N - 1) % N]);
 content.add_arc(eat[i], fork[i]);
};
for (int i = 0; i < N; ++i)
{
 // кладем фишки
 content.add_token(thinking[i]);
 content.add_token(fork[i]);
};
// создаем и запускаем сеть
petrinet_type petrinet(content);
petrinet.live(env);
```

В рассмотренном фрагменте сперва осуществляется создание длительных работ, потом создаются соответствующие длительные переходы, которые вносятся в содержимое сети. В отличие от рассмотренного ранее кода с использованием составных переходов, в данном случае наполнение длительных переходов внутренними элементами осуществляется конструктором `transition_long_type` неявно и потому клиентским кодом не реализуется.

# Глава 5.

# Модель актеров

В этой главе мы рассмотрим модель актеров, которая во многом напоминает описанные нами ранее сети конечных автоматов.

Одним из главных достоинств и в тоже время недостатков сетей конечных автоматов является наличие глобальной синхронизации всех автоматов сети между выполнением отдельных тактов. С одной стороны, наличие синхронизации делает вычисления полностью детерминированными, что сильно облегчает реализацию многих алгоритмов. С другой стороны, когда речь заходит о действительно распределенных системах, глобальную синхронизацию оказывается довольно сложно реализовать. Более того, даже если реализация удается, в большинстве случаев она оказывается неэффективной, поскольку в распределенных системах эффективность глобальной синхронизации упирается в скорость каналов взаимодействия между различными узлами, которая может быть разной и непостоянной. Даже если предположить, что время вычислений одного такта на всех узлах примерно одинаково, нельзя рассчитывать на то, что сигнал глобальной синхронизации придет на все узлы в одно и то же время, при этом следующий сигнал будет отправлен не ранее, чем придет последний ответ о завершении выполнения такта. Тем самым, быстродействие такой системы определяется самым медленным каналом связи.

Модель актеров лишена такого недостатка, поскольку исключает синхронизацию отдельных вычислительных элементов (актеров) между собой. Другое преимущество модели актеров заключается в возможности динамического роста системы в рамках распределенной среды.

## 5.1. Описание модели актеров

### 5.1.1. Первоначальное описание модели

Модель актеров описывает процесс вычислений как результат взаимодействия множества активных объектов — актеров. Термин «актер» используется здесь не в том смысле, который вкладывается в него в русском языке, а в смысле сущности, которая выполняет действия (actions). По этой причине зачастую в русскоязычных материалах вместо него используют термин «актор».

Актеры не имеют общих данных и взаимодействуют между собой посредством асинхронной отправки друг другу сообщений. Для этого с каждым актером связывается его имя (почтовый адрес). Актер характеризуется также своим поведением (behaviour), которое определяет типы сообщений, на которые он будет реагировать, а также его реакцию на эти сообщения. В каждый момент времени актер может реагировать на пришедшее сообщение либо находиться в ожидании прихода нового. В условиях отсутствия входящих

сообщений актер просто существует и не выполняет никаких действий. В ответ на пришедшее сообщение актер реагирует выполнением некоторых действий в соответствии со своим поведением, в связи с чем актер является сущностью скорее даже реактивной, нежели активной [49].

Отправка сообщений рассматривается в модели как универсальный примитив управления. Итерация, рекурсия, синхронизация, захват ресурсов и прочие подобные операции строятся на базе этого примитива [65, 66]. Как следствие, для актера в общем случае нет заданной изначально программистом непрерывной последовательности выполнения команд, не зависящей от последовательности приходящих сообщений. Именно таким образом ломается стереотип параллельно-последовательного программирования, и программа обретает в высшей степени параллельную структуру [48].

**Основные операции**

При выполнения действия (реакции на пришедшее сообщение) актеру доступны три операции, с помощью которых он может:

- отправить конечное количество сообщений себе или другим актерам (операция `send`);
- создать конечное количество новых актеров с некоторым поведением (операция `create`);
- задать свое поведение в ответ на приход следующего сообщения (операция `become`).

Порой, к выше перечисленным добавляют еще один пункт: в ответ на сообщение актер также может принять какие-либо локальные решения. Мы не перечислили его здесь, поскольку локальные решения явно не воздействуют на систему актеров и потому не имеют прямого отношения к описанию модели. Однако следует помнить, что на основе этих решений формируются те или иные последовательности выполнения операций, влияющих на систему актеров. Также именно здесь подразумевается возможность выполнения как таковых вычислений непосредственно текущим актером.

Все перечисленные операции могут выполняться актером параллельно, за исключением тех случаев, когда порядок выполнения операций определяется семантическими зависимостями [49]. Примером такой зависимости может быть создание актера, которому в параметрах инициализации передается адрес другого только что созданного актера.

Выполнение действия в рамках актера атомарно, т.е. пока актер не завершит выполнение одного действия в ответ на некоторое сообщение, он не получит новое сообщение и не начнет выполнение соответствующего ему другого действия [70].

При выполнении действия актер может послать сообщение не только другим актерам, но и себе. Учитывая атомарность действий, это было бы невозможно в условиях синхронного обмена сообщениями (так называемый метод «рандеву»), поскольку актер не может одновременно выполнять синхронные операции приема и отправки. Это является одним из аргументов в пользу асинхронной отправки сообщений [48].

При выполнении операции `create` происходит лишь как таковое создание нового актера и его почтового ящика, адрес которого возвращается. Важно помнить, что в этот момент актером выполняется лишь его инициализация, т.е. прием параметров, переданных при вызове операции создания. Никаких внешних действий актер при этом не выполняет, поскольку такие действия могут выполняться лишь в качестве реакции на пришедшее сообщение.

Операция `become` предполагает создание нового актера с новым поведением, который будет привязан к почтовому адресу текущего актера. При этом текущий актер становится

анонимным, т.е. актером без адреса и, следовательно, лишается дальнейшей возможности получения сообщений. По завершении обработки текущего сообщения он переходит в режим вечного ожидания новых сообщений, которых не получит, и формально перестает существовать для системы.

Операция `become` во время каждого действия, т.е. по приходу любого сообщения, должна выполняться не более одного раза. В противном случае такое поведение актера считается ошибочным [49]. Если она не выполнена явно, по умолчанию используется поведение, идентичное поведению текущего актера [48]. Таким образом, после каждого действия актер «перерождается» заново, т.е. каждый актер существует лишь в течение одного действия [48].

Как указывается в [48], операция `become` может порождать нового актера в момент ее выполнения, а не по завершении текущего действия. При этом новый актер может сразу приступать к приему и обработке следующего сообщения. Такой подход, хоть и усложнит реализацию, существенно повысит степень параллелизма актеров. По смыслу же такой подход полностью эквивалентен рассмотрению с точки зрения атомарности действия (т.е. замены актера лишь после завершения текущего действия), поскольку актеры не имеют разделяемых данных, а потому не завершивший текущее действие актер не может конфликтовать с только что созданным новым.

Работа системы актеров считается завершенной, когда она достигает состояния останова (halt), т.е. когда всеми актерами завершается обработка всех отправленных им сообщений, и они переходят в режим вечного ожидания новых.

По перечисленным доступным актеру операциям можно сделать вывод, что каждый актер во многом схож с автоматом. Исходя из своего текущего состояния (поведения) в ответ на входные данные (полученное сообщение) актер может порождать выходные данные (отправлять сообщения другим актерам) и менять внутреннее состояние (определять свое дальнейшее поведение). Однако система актеров заметно сильнее отличается от сети конечных автоматов, чем единичный актер от автомата. Отличия касаются, прежде всего, наличия в модели актеров асинхронности обмена сообщениями и недетерминированности момента их доставки. Также моделью актеров предусмотрено наличие операции создания нового актера, тогда как сети конечных автоматов статичны.

### Доставка сообщений

Отправка сообщений происходит асинхронно, т.е. актер, отправивший сообщение, не дожидается, пока оно будет доставлено актеру-получателю. В связи с этим возникает понятие события прихода сообщения, но отсутствует понятие события отправки сообщения [58]. Модель однако гарантирует, что сообщение обязательно будет доставлено актеру в течение некоторого произвольного конечного интервала времени.

Модель не вносит ограничений на порядок доставки сообщений, т.е. нет никаких гарантий, что некоторый набор сообщений будет доставлен актеру в том же порядке, в котором он был отправлен ему другим актером. Актеры не должны ориентироваться на конкретный порядок доставки. Это является следствием неопределенности времени доставки сообщения, которая естественным образом возникает при работе в распределенной среде, когда конфигурация путей доставки может динамически меняться. Если доставка некоторого сообщения заняла больше времени, чем доставка другого сообщения, отправленного позже, порядок их прихода, разумеется, будет не определен. Если бы потребовалось при этом сохранить порядок доставки, более поздние сообщения, пришедшие раньше, вынуждены были бы ожидать в очереди, пока не будут доставлены предшествующие им. Таким образом, жесткая привязка к порядку доставки могла бы существенно снизить производительность.

Сообщения должны доставляться актеру и обрабатываться им последовательно, даже если множество актеров отправляют ему сообщения одновременно. Для этого вводится понятие арбитра [64]. К каждому актеру (а вернее, его почтовому адресу) привязывается свой арбитр. В его задачи входит непосредственная передача сообщений актеру по одному за раз. Также именно на него возлагается задача доставки актеру за конечное время некоторого сообщения от другого актера на фоне непрерывного потока сообщений от третьего [64].

### Выборка сообщений

С каждым актером связан его так называемый почтовый ящик, т.е. множество сообщений, на текущий момент ему уже отправленных, но еще им не полученных и не обработанных. Получение актером сообщений является синхронным в том смысле, что если на текущий момент в его почтовом ящике нет подходящего сообщения, он его ожидает. Под подходящим сообщением подразумевается сообщение, успешно прошедшее операцию сопоставления с образцом (pattern matching).

Использование механизма сопоставления с образцом было предложено уже в самых ранних описаниях модели актеров [48, 65]. Образец в общем случае может включать описание типа сообщения или типов элементов, из которых оно состоит, также может содержать конкретные значения отдельных элементов или всего сообщения.

Если пришедшее сообщение соответствует некоторому заданному образцу, выполняется его обработка по правилам, поставленным в соответствие этому образцу. Если же сообщение не соответствует ни одному из заданных актером образцов, он не может быть применен для обработки этого сообщения [65]. В последнем случае сообщение актеру не доставляется, что аналогично игнорированию сообщения актером.

### Динамическая топология

Каждый актер может отправлять сообщения только тем актерам, адрес которых ему известен. Известны актеру могут быть только следующие адреса:

- содержащиеся в сообщении, в ответ на которое выполнятся текущее действие (переданные текущему актеру с помощью операции `send`);
- содержащиеся в параметрах инициализации при создании текущего актера (переданные в соответствующем вызове `create` или `become`);
- адреса актеров, которые были только что созданы текущим актером в рамках выполнения этого же действия (возвращенные операцией `create`);
- собственный адрес текущего актера (обычно обозначается `self`).

Поскольку формирование сообщений для отправки зависит от принятия тех или иных решений соответствующими актерами в момент выполнения, в общем случае заранее нельзя сказать, какие актеры будут общаться между собой. Более того, зачастую нельзя даже заранее сказать, сколько всего будет создано актеров с тем или иным поведением в процессе работы системы. Здесь следует учитывать недетерминированность порядка доставки сообщений, от которого может зависеть направление развития системы актеров.

Таким образом, организация связей между актерами на основе содержимого сообщений в совокупности с наличием операции создания новых актеров обуславливает возможность создания систем с динамической топологией. Это отличает модель актеров от, к примеру,

сетей конечных автоматов, в которых наличие конструктивных элементов и связей между ними предполагается изначальным, т.е. которые обладают статической топологией.

**Отсутствие глобального состояния**

Система актеров не содержит никаких разделяемых между актерами данных. Любые данные в системе либо являются внутренними данными некоторого актера, либо содержатся в сообщении, следующем до некоторого почтового ящика, и потому скоро также станут внутренними данными соответствующего актера. Каждый актер получает любые данные лишь двумя путями:

- через параметры инициализации при создании (переданные при вызове операции `create` или `become`);
- через параметры принятых сообщений (переданные при выполнении операции `send`).

Каждый актер автономно владеет своими внутренними данными, т.е. никакие данные системы не являются в ней глобальными. В совокупности с недетерминированностью доставки сообщений этот факт обуславливает невозможность наличия в общем случае состояния системы, которое имело бы глобальный смысл [67]. Здесь имеется в виду отсутствие понятия абсолютного состояния системы, описанного совокупностью состояний всех ее актеров без участия «наблюдателя», т.е. которое могло бы быть интерпретировано как общее состояние системы по аналогии с состоянием сети конечных автоматов. Любая оценка состояния системы может быть выполнена лишь с точки зрения «наблюдателя», т.е. некоторого актера, которого интересует состояние всей системы или какой-либо ее части, и который может собрать соответствующую информацию путем обмена сообщениями. Любая техника рассмотрения программы в терминах глобального вычислительного состояния позиционируется авторами как пагубная [65].

Как следствие, в системе актеров отсутствует понятие глобального времени. Каждый актер может вести свой локальный счетчик времени, но в общем случае он не соотносится с такими же счетчиками других актеров. Глобальное время не может иметь смысл снова по причине недетерминированности порядка и времени прихода сообщений. Каждый актер в качестве наблюдателя может сделать вывод о происшествии того или иного события только по факту прихода информации об этом, т.е. по приходу сообщения. Но, поскольку время доставки не определено, выводов по этим сообщениям об абсолютном (глобальном) времени происшествия событий актер делать не может.

Разумеется, для решения проблемы отсутствия глобального состояния может быть выделен отдельный актер, владеющий общей для всех информацией и предоставляющий к ней доступ. Однако этот подход не является аналогом общего состояния, поскольку снова упирается в недетерминированность времени доступа к общей информации. К примеру, можно попытаться организовать таким образом систему глобального времени, которая каждую секунду рассылала бы текущее значение времени всем актерам. Однако из-за неопределенности времени доставки каждый актер в результате получал бы сообщение с уже некорректным в общем случае значением, что лишает смысла такую реализацию.

### 5.1.2. Язык SAL для описания поведения актеров

В дальнейшем нам потребуется язык для краткого описания поведения актеров. Мы используем для этого некий диалект языка SAL (Simple Actor Language), предложенного в педагогических целях в [48].

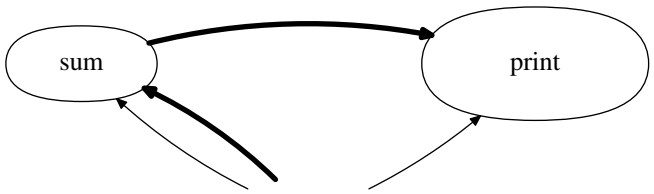



Рис. 5.1. Простая система актеров

Примером использования такого языка может служить следующая простейшая программа суммирования двух чисел:

```
def summator () [a, b, cust]
    send [a + b] to cust
end def

let s = create summator (),
    p = create print ()
in { send [2, 2, p] to s }
```

В этой программе содержится описание поведения одного актера, заключенное в конструкцию `def ... end def`. В начале такой конструкции задается имя поведения актера, список параметров инициализации в круглых скобках и список параметров принимаемого сообщения в квадратных скобках. Далее приводится описание реакции, выполняемой актером в ответ на пришедшее сообщение. В данном случае список параметров инициализации актера пустой, список параметров сообщения содержит два числа и адрес актера, которому надлежит отправить результат их суммирования.

Фрагмент кода после описания поведения выполняет начальную инициализацию системы актеров — операции создания актеров и отправки им сообщений. Код на этом уровне может содержать лишь вызовы операций `create` или `send`. Операция `become` этом уровне вызвана быть не может, поскольку здесь нет текущего актера, выполняющего действие [49]. По той же причине на этом уровне в параметрах создаваемого актера или в содержимом отправляемого сообщения не может быть передан адрес текущего актера, возвращаемый оператором `self`.

Оператор `let` создает привязки идентификаторов, которые будут использованы в соответствующем блоке `in { ... }`. В привычных нам терминах это означает объявление переменных и присваивание им значений. Важное отличие заключается в том, что такое присваивание происходит однократно, т.е. значение, связанное с идентификатором, не может быть изменено. В данном случае происходит привязка к идентификаторам `s` и `p` адресов создаваемых актеров, после чего эти идентификаторы используются при отправке сообщения. Поведение актера `print` здесь не описано, в его задачи входит вывод на экран содержимого всех пришедших сообщений. Выполнение программы завершается, когда все созданные в процессе ее работы актеры обработают все свои сообщения.

Назначение приведенной программы заключается в выводе на экран суммы двух чисел. Создаются два актера с поведениями `summator` и `print`, после чего первому из них отправляется сообщение, в котором передаются аргументы суммирования и адрес второго. В качестве реакции на пришедшее сообщение сумматор выполняет суммирование и отправляет результат актеру, адрес которого был передан в сообщении, т.е. актеру `print`. На рис. 5.1 отображена схема взаимодействия в такой системе актеров. Здесь и в дальнейшем жирными стрелками обозначается пересылка сообщений, тонкими — создание актеров.

В качестве более сложного примера использования языка SAL рассмотрим программу

параллельного суммирования сдваиванием:

```
def customer (value, cust) [newval]
    if value = NIL
    then
        become customer (newval, cust)
    else
        send [value + newval] to cust
    fi
end def

def summator () [b, e, cust]
    if e - b = 1
    then
        send [b] to cust
    else
        let c = create customer (NIL, cust),
            s1 = create summator (),
            s2 = create summator (),
            h = floor((e - b) / 2)
        in {
            send [b, b + h, c] to s1
            send [b + h, e, c] to s2
        }
    fi
end def

let s = create summator (),
    p = create print ()
in { send [1, 5, p] to s }
```

Эта программа выполняет суммирование всех целых чисел от единицы до четырех включительно. Диапазон суммируемых чисел задается в последней строке (правая граница в интервала не включается). В программе описано поведение двух типов актеров. Актер с поведением `summator` принимает сообщение с границами диапазона чисел и адресом актера, которому следует отправить результат. В задачи этого актера входит проверка ширины переданного интервала и, если интервал содержит лишь одно число, отправка этого числа актеру-заказчику. Если же ширина интервала больше единицы, создаются актер с поведением `customer`, который теперь будет ответственен за передачу результата актеру-заказчику, и два новых актера с поведением `summator`. После вычисления приблизительной середины интервала выполняется отправка двух сообщений каждому из созданных актеров `summator`. В отправляемых сообщениях указываются границы половинных интервалов и адрес актера `customer`, которому оба сумматора должны направить результат.

В задачи актера с поведением `customer` входит ожидание двух сообщений с некоторыми значениями, суммирование этих значений и отправка результата далее по цепочке актеру-заказчику, адрес которого текущий актер получил в параметрах инициализации. Получив первое значение, актер запоминает его, выполняя операцию `become` с новыми инициализационными параметрами. После получения второго сообщения актер формирует и отправляет их сумму по назначению.

В начале работы программы создаются два актера, после чего производится отправка интервала суммирования актеру `summator` с передачей адреса актера `print` в качестве заказчика (рис. 5.2). В процессе работы первоначально созданный актер `summator` в соответствии с только что описанным поведением создает еще трех актеров. Два из них (также актеры с поведением `summator`) снова создают по три актера каждый. На этом уровне

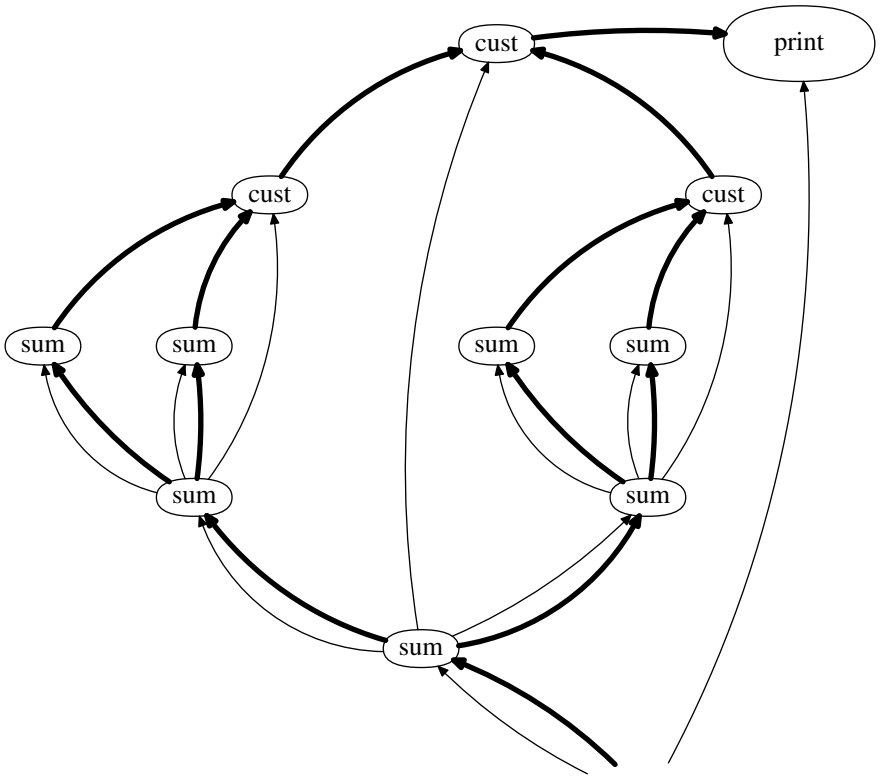



Рис. 5.2. Система актеров, реализующая суммирование сдваиванием

дальнейший рост данной системы прекращается, актеры `summator` отправляют значения соответствующим актерам `customer`, которые по цепочке суммируют значения и отправляют конечный результат актеру `print`.

В приведенной программе каждый актер `summator`, обработав одно сообщение, далее остается в системе, не обрабатывая более никаких сообщений. Это не создает никаких принципиальных противоречий с моделью, однако на практике может заставить систему актеров слишком быстро разрастаться. В такой ситуации было бы правильнее использовать именно этого актера вместо одного из актеров `summator` следующего уровня. В этом случае следует вместо адреса второго актера, опустив его создание, передавать адрес текущего актера с помощью оператора `self`:

```
def summator () [b, e, cust]
...
        let c = create customer (NIL, cust),
            s = create summator (),
            h = floor((e - b) / 2)
        in {
            send [b, b + h, c] to s
            send [b + h, e, c] to self
        }
...
end def
```

Разумеется, столь короткого описания языка недостаточно для внесения полной ясности о деталях его использования. Более детальное описание синтаксиса SAL может быть почерпнуто из первоисточников [48, 49].

В приведенных нами на текущий момент примерах программ синтаксис языка SAL соответствовал описанному в [48]. Для рассмотрения в дальнейшем более сложных случаев, когда актер может принимать сообщения нескольких типов, мы внесем в этот синтаксис изменение на уровне анализа содержимого пришедшего сообщения.

Предположим, актер, в отличие от предыдущих примеров, может принимать сообщения разных типов и, соответственно, по-разному на них реагировать. К примеру, актер может принимать сообщение `get`, в ответ на которое он возвращает значение своей внутренней переменной, а также сообщение `put`, задающее новое значение этой переменной. В исходном синтаксисе SAL с целью описания поведения такого актера использовалась следующая конструкция [48]:

```
def variable (value)
[
case operation of
    put: (newval)
    get: (cust)
end case
]
    if operation = put
    then
        become variable (newval)
    else
        send [value] to cust
    fi
end def
```

Здесь оператор ветвления `case ... end case` используется для привязки идентификаторов к содержимому пришедшего сообщения, после чего в теле реакции (которая едина для для всех типов принимаемых сообщений) производится еще одно ветвление на основе оператора `if`. В тех же целях нами в дальнейшем будет использован следующий синтаксис:

```
def variable (value)
[put, newval]
    become variable (newval)
[get, cust]
    send [value] to cust
end def
```

Ключевое отличие заключается в том, что в рамках описания поведения актера может быть приведен не один, а несколько вариантов списка параметров принимаемого сообщения, после каждого из которых следует описание соответствующей реакции. Это позволяет нам избежать повторного ветвления, сократить запись и повысить ее наглядность. Синтаксис отправки сообщений такому актеру в обоих случаях одинаков [48]:

```
send [put, 1] to var
send [get, self] to var
```

Здесь предполагается, что идентификатор `var` привязан к адресу актера с поведением `variable`.

В некоторых случаях при описании поведений мы будем опускать идентификаторы типов сообщений, если они могут быть однозначно определены на основе типов содержимого сообщения. К примеру, в приведенном примере содержимое одного сообщения — число, содержимое другого — почтовый адрес актера, который в общем случае числом не является. Это позволяет нам опустить идентификаторы `put` и `get`, поскольку при анализе сообщения мы можем отталкиваться от типа содержимого.

### 5.1.3. Некоторые существующие модификации модели

Как уже говорилось выше, в отличие от сетей конечных автоматов и сетей Петри, в которых существование конструктивных элементов предполагается изначальным, модель актеров предусматривает операцию создания актера. При этом в модели не предусмотрено операции уничтожения актера. Иначе говоря, в соответствии с моделью, однажды созданный актер существует на протяжении всей жизни системы, т.е. модель описывает монотонно возрастающую систему актеров. Это представляется довольно естественным подходом, когда речь идет о формировании баз знаний, на использование в которых зачастую и ориентировались авторы [65], поскольку знания должны накапливаться. Однако это создает существенные трудности при реализации параллельных вычислений, поскольку некоторые промежуточные результаты, будучи один раз использованными, в большинстве случаев в дальнейшем не понадобятся, вследствие чего представляется целесообразным освобождение связанных с ними ресурсов.

Завершение вычислений в исходной модели описывается как состояние останова, когда каждый актер системы обработал все свои входные сообщения и находится в ожидании новых, которым, разумеется, неоткуда при этом взяться, поскольку остальные актеры также неактивны и находятся в ожидании. Фактически, в терминах привычной нам модели программирования это означает, что завершение вычислений наступает тогда, когда достигается глобальная взаимоблокировка актеров (deadlock). Вопрос целесообразности такой интерпретации готовности системы актеров к завершению при проведении параллельных вычислений также остается открытым, поскольку требует решить проблему регулярного выполнения проверки пустоты большого количества почтовых ящиков в распределенной среде.

Наконец, известны некоторые сложности, возникающие как следствие недетерминированности порядка доставки сообщений [70]. Например, при реализации стека на основе актеров положительный ответ на запрос о пустоте стека в текущий момент может означать как то, что он пуст, так и то, что запросы на помещение значений в стек, отправленные ранее, на момент прихода запроса о пустоте все еще не достигли адресата. Авторы модели указывают, что сохранение порядка сообщений при необходимости может быть внесено путем введения номеров сообщений с последующим их анализом и буферизацией на приемной стороне [48]. Однако сам по себе такой подход, как минимум, создает некоторые неудобства, требуя дополнительных усилий от разработчика.

Другой вопрос здесь возникает касательно буферизации несвоевременно пришедших сообщений, и ответ на него вызывает некоторые противоречия. Если для буферизации используется хранилище (к примеру, список), также построенное на основе актеров, то корректность взаимодействия с ним снова упирается в порядок прихода сообщений. Снова нет никакой гарантии, что сообщения от актера-хранилища придут в той же последовательности, в которой они им отправлены. Этот вопрос может быть решен путем использования протокола взаимодействия с хранилищем по типу «запрос — ответ» с отправкой следующего запроса не ранее, чем получен ответ на предыдущий. Но при использовании в системе взаимодействия по такому протоколу вообще становятся не нужны номера сообщений и их буферизация во временных хранилищах, однако существенно повышаются временные затраты на взаимодействие актеров.

Вследствие этих сложностей для сохранения порядка следования сообщений предлагается использовать внутренние очереди [51]. Такая очередь может реализовываться, к примеру, списком значений, передаваемых в качестве параметра инициализации при создании актера. Соответственно, при выполнении буферизации нового сообщения актером должна выполняться операция `become` с параметром — текущим содержимым списка, дополнен-

ным новым элементом. Такой подход, разумеется, решает поставленный вопрос, однако все еще требует дополнительных усилий от программиста, а также в некоторых реализациях может повлечь существенные потери производительности из-за передачи большого количества параметров. Помимо этого, использование внутренней очереди может потребовать выполнения итерации или рекурсии в рамках одного действия, что само по себе отвергается моделью.

Перечисленные выше моменты заставляют, зачастую, пересматривать модель актеров в практических реализациях и вводить в нее дополнительные правила, расширения или ограничения с целью повышения ее эффективности либо удобства использования с учетом сегодняшних методик и средств программирования.

Сама по себе модель актеров для большинства разработчиков оказывается несколько непривычной, в связи с чем многие реализации предлагают некий компромисс — попытку интегрировать ее в более привычные модели программирования. В частности, в очень многих реализациях актер активен, а не реактивен, т.е. программа, представляющая актера, может не бездействовать до момента прихода сообщения, а работать по своему усмотрению. Иначе говоря, в обычном режиме программа не является актером, однако при необходимости может стать им путем явного вызова синхронной операции получения сообщения. В этот момент она переходит в тот режим, в котором и должен пребывать актер, т.е. ожидает прихода подходящего сообщения, после чего выполняет соответствующий обработчик. По завершении выполнения обработчика завершается и выполнение синхронной функции получения, после чего программа снова перестает быть актером в классическом понимании.

Выбор подходящего сообщения производится на основе фильтра, передаваемого функции получения. Фильтр представляет собой образец или набор образцов, с которыми производится сопоставление сообщений в почтовом ящике. Если сообщение не удовлетворяет ни одному образцу, оно может быть проигнорировано (как изначально предлагается моделью), а может быть сохранено в почтовом ящике. В последнем случае оно может быть получено и обработано позже, когда актером будет определен другой фильтр (что аналогично изменению поведения актера с помощью операции `become`). В связи с этим в таких системах понятие прихода сообщения актеру (arrival) разделяется на два: доставку сообщения в почтовый ящик актера (delivery) и получение сообщения актером (reception) из своего почтового ящика.

Во многих реализациях актер существует с момента создания не постоянно, а лишь до тех пор, пока сам не посчитает нужным покинуть систему. Поскольку актеру обычно как никому другому известно его предназначение, ему, как правило, не составляет труда определить момент, когда он уже выполнил свою задачу и больше не нужен. Соответственно, моментом завершения вычислений в таком случае можно считать завершение работы всех актеров системы. Отправка сообщения несуществующему актеру в таких системах обычно не вызывает ошибки [50].

Подобный подход используется, порой, и в классической модели актеров с помощью операции `become sink ()`, где поведение `sink` предписывает соответствующему актеру игнорировать все принимаемые сообщения. Учитывая это, такой актер может быть безболезненно удален из системы, поскольку никаких действий он уже выполнять не будет в любом случае. При этом, разумеется, следует уничтожать и сообщения, отправленные на его адрес.

Помимо завершения актера, зачастую модель также дополняется ограничением на порядок следования сообщений. Разумеется, в распределенных системах трудно рассчитывать на сохранение порядка следования сообщений между узлами, однако упорядоченность может быть обеспечена на уровне реализации арбитра прозрачно для актера с помощью, к примеру, введения номеров сообщений, как и предлагали авторы модели [48]. Конечно,

это обуславливает некоторое снижение производительности по сравнению с задуманным авторами подходом, однако во многом существенно облегчает использование актеров в программировании.

Практически все перечисленные модификации модели актеров реализованы в языке Erlang. Считается, что модель актеров стала популярна именно благодаря ему. Большинство прочих реализаций модели актеров на сегодняшний момент так или иначе повторяют подход, использованный в Erlang. К примеру, такой подход используют многие библиотеки для работы с актерами в интерпретируемых языках, таких как Ruby, а также функции работы с актерами из состава стандартной библиотеки Scala. Возможно, именно по причине наличия подобных отступлений от классической модели в книгах и документации по Erlang, как правило, модель актеров явно упоминается в лучшем случае лишь в библиографии [50]. Нельзя не признать, что эти отступления, действительно, весьма удачно согласуются с привычными подходами написания программ, что, вероятно, и обеспечило рост популярности Erlang. В наши же задачи входит рассмотрение классической модели, поэтому мы в дальнейшем постараемся, по возможности, количество подобных отступлений со своей стороны сократить.

## 5.2. Различные варианты реализации

Удобнее всего модель актеров реализуется в рамках высокоуровневых языков с наличием развитых механизмов сопоставления с образцом (pattern matching) и сериализации (преобразования данных для передачи в распределенной среде). Также немаловажной является возможность использования сборки мусора (garbage collection), которая помогла бы решить проблему уничтожения актеров, адреса которых в системе больше не используются. Отсутствие в C++ встроенной поддержки этих механизмов, тем не менее, не помешают нам предложить простую реализацию, приемлемую для использования в иллюстративных примерах.

### 5.2.1. Простая одноуровневая реализация

Существует немало реализаций модели актеров. Большинство из них содержат отклонения от классической модели, которые, по сути, являются оправданными и зачастую довольно удачными попытками спроецировать ее на популярные сегодня подходы в программировании. Однако в наши задачи входит знакомство читателя именно с исходной моделью актеров, по причине чего мы будем стараться реализовать ее, по возможности, наиболее отвлеченно, не опираясь на простоту и кажущиеся преимущества последовательного императивного подхода. Разумеется, и в нашем случае неизбежны отклонения от классической модели, однако мы по мере возможности будем стараться сводить их к минимуму.

Следует отметить, что мы реализуем предложенную авторами модель в демонстрационных целях, при этом реализация по возможности сделана простой и наглядной. Эта реализация, однако, далеко не во всех деталях оптимальна и при большом количестве актеров может оказаться неэффективной (во многом благодаря как раз тому, что следует описанной модели). Тем не менее, она удовлетворяет нашим целям проиллюстрировать модель актеров. Любую же оптимизацию, усовершенствование модели с целью большего соответствия существующим средствам программирования, повышение гибкости при написании клиентского кода и прочие усложнения предложенной реализации каждый разработчик, исходя из своих задач, всегда может внести по своему усмотрению. Одно из подобных расширений модели (выполнение полных жизненных циклов вложенных подсистем актеров) будет рассмотрено нами позже.

### Сериализация и сопоставление с образцом

Механизм сопоставления с образцом в предлагаемой реализации основан на типе передаваемого сообщения, что является довольно простым подходом. Поскольку реализация может быть многопроцессной, рассчитывать на выбор обработчика пришедшего сообщения средствами компилятора не приходится, в связи с чем вместе с сообщением пересылается идентификатор его типа, полученный с помощью оператора `typeid`. Для пересылки сообщений между процессами требуется сериализация данных. В нашем случае каждое сообщение рассматривается как участок памяти и преобразуется в массив байтов, что также является самым простым способом сериализации, имеющим известные недостатки и ограничения. В более сложных вариантах реализации можно использовать, к примеру, преобразование в текст с последующим использованием регулярных выражений.

Помимо необходимости пересылки данных между процессами, представление сообщения в виде массива байтов требуется нам для того, чтобы привести сообщения различных типов к одному для хранения в очередях. В рамках одного процесса можно вместо такого преобразования ввести абстрактный класс сообщения, однако в наши цели входит иллюстрация не языковых возможностей, а возможностей различных реализаций распараллеливания.

Вследствие использования столь упрощенного механизма сериализации у нас появляются ограничения на содержимое сообщений и используемые в них типы данных. А именно, сообщения могут быть представлены одним из базовых типов данных (`char`, `int`, `double` и т.п.) или же структурой, которая должна самодостаточно содержать представляемые ей данные и не должна явно или неявно ссылаться на другие области памяти. Т.е. она не должна содержать указателей и ссылок, а также внутренних структур или объектов, содержащих указатели или ссылки (в частности, объектов-контейнеров STL наподобие `std::vector` или `std::list`). Такие требования к типу сообщения соответствуют заявленному в стандарте языка определению типов POD (plain old data), дополненному необходимостью отсутствия указателей. В соответствии со стандартом, данные POD-структуры могут быть скопированы в массив байтов того же размера и позже восстановлены. Отсутствие указателей добавляет возможность безопасного восстановления структуры в памяти другого процесса.

Наконец, при многопроцессной реализации все процессы должны будут работать на машинах со схожей архитектурой и компилироваться с одинаковыми флагами, иначе могут возникнуть ошибки из-за различий в представлениях базовых типов данных или в выравниваниях полей структур. Кроме того, на всех машинах должны использоваться совместимые версии компилятора, генерирующие средствами `typeid` одинаковые идентификаторы для одинаковых типов данных. Все это, разумеется, в известной степени ограничивает возможности использования предлагаемой реализации. Однако эти ограничения являются следствием ее простоты и могут быть устранены путем введения более сложных механизмов сериализации и сопоставления с образцом.

### Схема взаимодействия классов

Как и ранее, будем реализовывать модель в виде набора контекстно-независимых классов, которые так или иначе будут использованы клиентским кодом. Оговорим сущности, которые нам потребуются в рамках этого набора.

Прежде всего, разумеется, нам нужен базовый класс актера, который смогут наследовать классы клиентского кода при реализации актеров с тем или иным поведением. С точки зрения клиентского кода, этот класс должен предоставлять возможность вызова четырех основных функций (`create`, `send`, `become` и `self`), а также функцию регистрации

обработчика для сообщений конкретного образца. Каждый обработчик сообщений является функцией-членом наследующего класса с сигнатурой фиксированного формата. Базовый класс актера содержит описание таблицы обработчиков и обеспечивает их выполнение в момент прихода соответствующих сообщений.

Исходя из описания модели, при выполнении операции `become` очередь входящих сообщений текущего актера привязывается к новому, только что созданному. Текущий же актер лишается своего адреса и сразу после завершения обработки текущего сообщения становится для системы бесполезным, что позволяет его безболезненно уничтожить. Таким образом, почтовый адрес привязан не к актеру, а к почтовому ящику, или, точнее, к арбитру, который руководит доставкой сообщений актеру, ассоциированному с этим почтовым адресом в текущий момент. В отличие от актера, которые в общем случае могут постоянно меняться, арбитр привязан к почтовому адресу постоянно. Именно на него возложим, помимо доставки сообщений, инициирование создания и уничтожения актеров.

Также нам потребуется объект фабрики актеров. Фабрика нужна по той причине, что создание актера, хоть и производится по инициативе клиентского кода, все же не должно им осуществляться непосредственно, поскольку иначе на него возлагается и решение множества сопутствующих вопросов, таких как размещение созданного объекта в другом потоке или процессе. Для создания и уничтожения актеров различных типов фабрика должна содержать соответствующий набор функций. Поскольку во время работы системы актеры создаются и уничтожаются постоянно (в рамках выполнения операций `create` и `become`), этим функциям нужно уделить особое внимание в отношении скорости выполнения. К примеру, для быстрого отображения идентификатора поведения создаваемого актера на соответствующую функцию создания удобно использовать контейнер `std::map` (или даже `std::tr1::unordered_map`). Чтобы это не потребовало дублирования соответствующего кода в каждой программе, реализация класса фабрики также должна быть перенесена на уровень базового набора классов. На совести клиентского кода в этом случае остается лишь объявление класса актера с конструктором, имеющим сигнатуру фиксированного формата, и регистрация этого класса в фабрике актеров.

Наконец, для организации взаимодействия актеров потребуется планировщик. В его задачи будет входить обработка запросов на создание новых актеров, а именно создание новых арбитров, и доставка сообщений существующим арбитрам. Также именно на него возложим в дальнейшем распределение арбитров по параллельным ресурсам.

Таким образом, предлагаемая реализация модели актеров содержит следующий набор классов (рис. 5.3). Планировщик, представленный классом `scheduler_type`, по запросам на создание актеров осуществляет создание арбитров, каждому из которых ставит в соответствие некоторый почтовый адрес (тип `address_type`). Каждый арбитр (класс `arbiter_type`) при создании, а также после каждого выполнения его текущим актером операции `become`, осуществляет обращение к фабрике актеров. В ответ на это фабрика актеров (класс `factory_type`) осуществляет создание и уничтожение актеров, которые являются объектами классов, унаследованных от базового класса `actor_type`. Актеры во время обработки приходящих сообщений путем вызова функций `create` и `send` передают планировщику запросы на создание актеров и доставку сообщений, а также с помощью функции `become` передают своему арбитру запрос на замену текущего актера. На рис. 5.3 жирными стрелками обозначена доставка сообщений, тонкими — операция создания, пунктирными — передача соответствующих запросов.

Каждый класс актера должен быть унаследован от класса `actor_type`, содержать конструктор и набор обработчиков сообщений. Поскольку конструктор вызывается фабрикой, формат вызова должен быть фиксированным. То же касается и обработчиков сообщений, поскольку они вызываются базовым классом. Каждый обработчик имеет имя `action` и

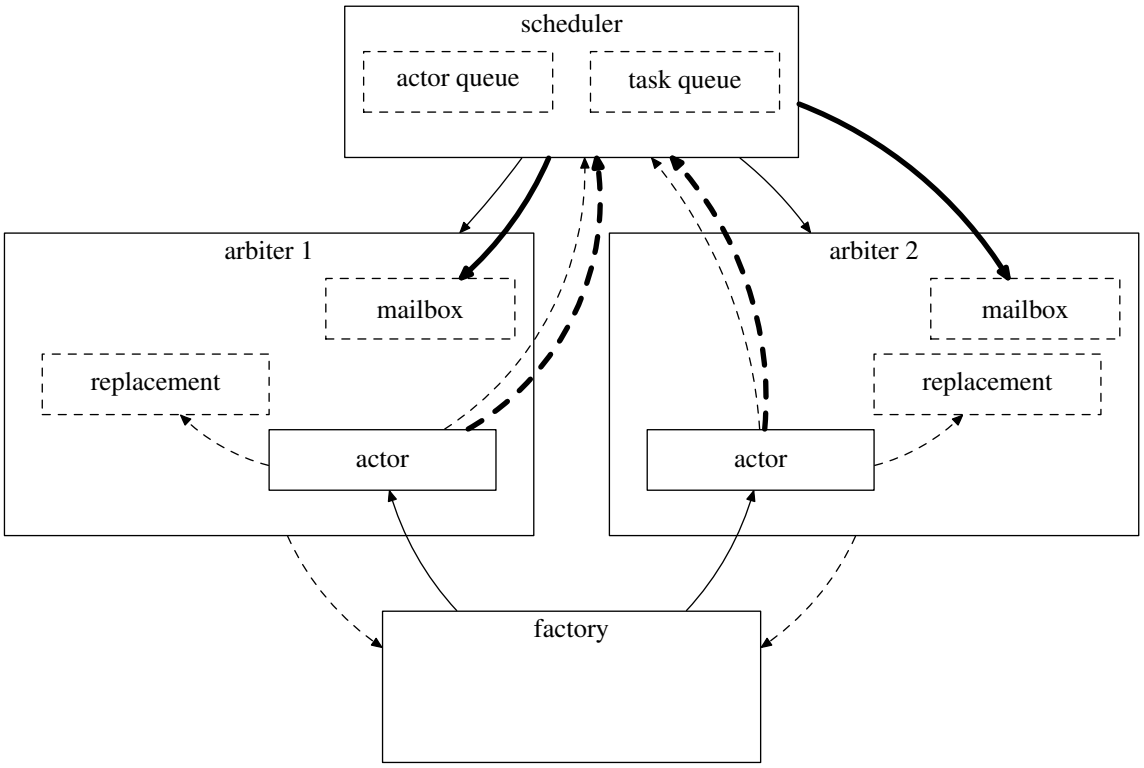



Рис. 5.3. Схема взаимодействия классов

один параметр — ссылку на немодифицируемый объект некоторого произвольного типа. Сопоставление с образцом при приходе сообщения производится именно на основе типа отправленного сообщения. Конструктор также обязан иметь в качестве параметра ссылку на немодифицируемый объект произвольного типа. В качестве примера приведем следующий код:

```
struct message1_type
{
 // ...
};
struct message2_type
{
 // ...
};
class someactor_type: public actor_type
{
public:
 struct init_type
 {
  // ...
 };
 someactor_type(const init_type &init)
 {
  add_action<someactor_type, message1_type>();
  add_action<someactor_type, message2_type>();
 }
 void action(const message1_type &msg)
```

```
 {
  // ...
 }
 void action(const message2_type &msg)
 {
  // ...
 }
};
```

Класс `someactor_type` описывает поведение актера, параметры инициализации которого представлены его внутренней структурой `init_type`. Актер предоставляет обработчики для сообщений двух образцов: `message1_type` и `message2_type`. Обработчики регистрируются в конструкторе, а не, к примеру, в статической функции класса, вызываемой единожды для всех актеров с данным поведением, поскольку набор актуальных обработчиков в общем случае может зависеть от параметров инициализации. Никакие другие функции (такие как создание, отправка и т.п.) в конструкторе не вызываются, так как они могут быть вызваны только в обработчике — в качестве реакции на сообщение.

Выполнение жизненного цикла системы с участием такого актера может быть представлено, к примеру, следующим фрагментом кода:

```
// регистрация актеров в фабрике
factory_type factory;
factory.add_definition<someactor_type, someactor_type::init_type>();

// планировщик
scheduler_type sched;

// инициализация системы актеров
address_type addr;
someactor_type::init_type init = { /* ... */ };
addr = sched.system().create<someactor_type>(init);
message1_type msg1 = { /* ... */ };
sched.system().send(addr, msg1);
message2_type msg2 = { /* ... */ };
sched.system().send(addr, msg2);

// жизненный цикл системы актеров
sched.evolve(factory);
```

Здесь выполняется регистрация одного определения поведения в фабрике актеров, далее в рамках начальной инициализации системы актеров создается один актер, после чего ему отправляются два сообщения. В этот момент системе лишь передаются запросы на создание и доставку, реально же создание не производится, в связи с чем фабрика нужна лишь при выполнении функции `evolve`, выполняющей полный жизненный цикл системы актеров.

### Описание реализации

В приложении Д приведена реализация описанного набора классов. Отметим, что на этот раз мы сосредоточились на защите клиентского кода от использования методов, не предназначенных для него, в связи с чем закрыты даже интерфейсы, предоставляемые перечисленными классами друг другу. В частности, класс арбитра вообще не содержит открытых членов, поскольку он не предназначен для использования в клиентском коде. Для

доступа служебных классов к предоставляемым ими друг другу интерфейсам использовано явное указание их в качестве «друзей» (`friend`).

Для клиентского кода класс `actor_type` предоставляет два специальных типа, а также набор функций, которые могут быть использованы наследующими его классами. Тип `empty_type` предназначен для краткой записи в клиентском коде операций `create` или `become` для случаев, когда описание поведения актера не содержит параметров инициализации. В таких случаях следует указать тип `empty_type` в качестве параметра конструктора соответствующего класса актера. Тип `unknown_type` служит в качестве образца сообщения для обработчика, который должен вызываться для сообщений, не удовлетворяющих ни одному из других образцов. Поскольку нам не известен исходный тип данных такого сообщения (иначе мы могли бы указать конкретный образец), в отношении сообщений типа `unknown_type` допустима только операция пересылки другому актеру.

Класс `actor_type` предоставляет клиентскому коду следующие функции:

- `add_action` — добавление в таблицу обработчика, соответствующего некоторому заданному типу сообщения;
- `create` — создание нового актера, возврат его почтового адреса;
- `become` — объявление последующего поведения текущего актера;
- `send` — отправка сообщения актеру с заданным почтовым адресом;
- `self` — получение почтового адреса текущего актера.

С помощью вызова функции `add_action` из конструктора класса, наследующего `actor_type`, актер задает набор обработчиков, соответствующих различным типам сообщений (различным образцам). Также актер может указать один обработчик на случай, если пришедшее сообщение не соответствует ни одному из предусмотренных образцов. Для этого в параметрах обработчика указывается специальный тип `unknown_type`. Функция `add_action` добавляет в таблицу `m_acttable` новый элемент, соответствующий заданному типу сообщения. При этом инстанцируется шаблон статической функции `activate`, задача которой заключается в вызове обработчика — члена класса конкретного актера. Для специального типа `unknown_type` функция `activate` отличается. Перегруженная функция `get_activator` используется для инстанцирования одного из двух шаблонов `activate`. Для выбора компилятором конкретного ее экземпляра используется передача фиктивных нулевых указателей из функции `add_action`.

Функция `create` выполняет формирование структуры поведения нового актера на основе заданного типа актера, типа и содержимого его параметра инициализации, после чего передает результат планировщику. Функция `become` также формирует структуру поведения нового актера, однако передает результат арбитру, поскольку именно им будет создан новый актер на замену текущему. В рамках функции `send` осуществляется формирование структуры пересылаемого сообщения на основе его типа и содержимого, результат передается планировщику для доставки адресату. В случае, если переданный параметр имеет тип `unknown_type`, из него извлекается пересылаемое сообщение, в котором заменяется адрес назначения.

Помимо функциональности актера, в классе `actor_type` объявляются многие внутренние типы и общие функции, используемые в том числе и остальными классами. В частности, сюда попадают объявления структуры поведения `behaviour_type` и структуры пересылаемого сообщения `task_type`. Также сюда попадает функция преобразования произвольного сообщения или параметров создания актера в массив байтов — `param2chunk`, а также функция, выполняющая обратное преобразование — `chunk2param`.

Наконец, здесь же объявлены функции генерации идентификаторов поведения и образца сообщения — `def_id` и `pat_id` соответственно. Обе они возвращают строку, соответствующую переданным типам. В случае генерации идентификатора поведения в начале функции производится проверка наличия объявленного конструктора с параметром переданного типа. Эта проверка производится во время компиляции, во время же выполнения объект не создается, для чего и использован тернарный оператор. Идентификатор типа параметра также включается в строку идентификатора поведения, чтобы не возникло ситуации, при которой актер создается не с тем типом параметра, который был указан фабрике актеров. Так может получиться, если класс актера содержит несколько конструкторов с параметрами разных типов.

Класс актера предоставляет классу арбитра следующие функции:

- `bind` — привязка объекта актера к конкретным объектам арбитра и планировщика;
- `match` — проверка наличия у актера обработчика для сообщения заданного образца;
- `apply` — выполнение актером действия над переданным сообщением.

При создании актера фабрика не передает ему через конструктор указатели на соответствующие объекты планировщика и арбитра. Если бы они передавались через конструктор актера, это бы потребовало явного указания такой передачи базовому конструктору `actor_type` в клиентском коде, что, безусловно, добавило бы головной боли при его написании. Поскольку мы стараемся упростить клиентский код, передача указателей на объекты планировщика и арбитра в обход наследующего конструктора возложена на функцию `bind`, которая должна быть вызвана сразу после создания актера. Как мы говорили выше, актер в момент создания не производит действий, т.е. не отправляет сообщений и не задает последующего поведения, поэтому в момент создания арбитр и планировщик ему не нужны. Это позволяет нам вынести передачу указателей на них в функцию, вызываемую после создания.

Фабрика актеров представлена классом `factory_type`. Она предоставляет клиентскому коду одну функцию: `add_definition`. В ее задачи входит инстанцирование и сохранение в соответствующих таблицах двух статических функций — для создания (`construct`) и уничтожения (`destruct`) актера заданного типа. Инстанцированные и сохраненные в таблицах `m_ctortable` и `m_dtortable` функции вызываются по инициативе арбитра. Для этого ему предоставляется следующий интерфейс:

- `create_actor` — создание актера с заданным поведением, возвращает указатель на созданного актера;
- `destroy_actor` — уничтожение актера по переданным указателю и описанию поведения.

Арбитр является сущностью, требующейся на уровне реализации, клиентскому же коду класс `arbiter_type` интерфейса не предоставляет. Каждый арбитр создается планировщиком в ответ на обработку запроса на создание нового актера, при этом созданному арбитру передается описание поведения актера, который должен быть создан, а также его почтовый адрес. В задачи арбитра входит хранение содержимого почтового ящика, передача сообщений из него текущему актеру и, наконец, создание и уничтожение актеров, ассоциированных с текущим почтовым адресом, в рамках их смены поведения. Создание и уничтожение производится с помощью фабрики актеров, ссылка на которую также передается арбитру в параметрах конструктора. Арбитр предоставляет классу `actor_type` следующие функции:

- `new_behaviour` — смена поведения, вызывается актером в рамках выполнения операции `become`;

- `address` — возврат почтового адреса текущего арбитра, вызывается в ответ на обращение к операции `self`.

Во время вызова функции `new_behaviour` запрос на смену поведения сохраняется во внутренней очереди арбитра. Смена же поведения осуществляется арбитром путем пересоздания актера после завершения выполнения действия текущим актером.

Помимо интерфейса для актера, арбитр также предоставляет функции планировщику:

- `deliver` — помещение очередного сообщения в почтовый ящик;

- `empty` — проверка почтового ящика на пустоту;

- `retrieve` — попытка получения очередного сообщения из ящика;

- `process` — передача сообщения актеру на обработку.

Функция `retrieve` осуществляет выборку сообщений из ящика в произвольном порядке, а не в порядке их прихода, для чего использован случайный выбор из всего содержимого ящика. Это сделано для большего соответствия модели, поскольку считается, что множество сообщений, поступивших в почтовый ящик, не упорядочено, и в каждый момент актер может получить любое из них. Некоторые пришедшие сообщения могут не соответствовать ни одному заявленному образцу текущего актера, тогда такое сообщение остается в почтовом ящике. Выбор другого сообщения в такой ситуации не производится и откладывается до следующей попытки. По этим причинам функция `retrieve` не обязательно извлекает сообщение из почтового ящика, даже если ящик не пуст.

Может возникнуть вопрос, правомерно ли внесение элемента случайности прямо в реализацию базового набора классов. В модели актеров, как и, к примеру, в сетях Петри, неопределенность момента возникновения событий опирается на факторы внешней среды. Однако в сетях Петри порядок возникновения событий не определен моделью, но может быть определен во внешней среде, т.е. выбор может быть не случайным, а просто лежать за рамками модели. В случае же модели актеров недетерминированность порядка доставки присуща природе следования сообщений в распределенной среде, и потому является частью описания модели именно в виде неподвластного нам элемента неопределенности. По этой причине случайность внесена нами непосредственно в реализацию модели.

В функции `process` осуществляется обработка текущим актером сообщения, извлеченного ранее из почтового ящика с помощью функции `retrieve`. В случае, если во время выполнения действия было задано замещающее поведение, осуществляется замена актера. При этом выполняется уничтожение текущего актера, ассоциированного с данным почтовым адресом, и создание вместо него нового.

Планировщик, представленный классом `scheduler_type`, осуществляет функционирование системы актеров, в рамках которого обеспечивается обмен сообщениями и создание новых актеров. Для выполнения этих задач в планировщике хранятся две очереди запросов: очередь запросов на создание новых актеров `m_actorqueue` и очередь запросов на доставку сообщений `m_taskqueue`. Помимо этого, в объекте планировщика хранится массив созданных им арбитров `m_arbiterlist`, индексация в котором производится по их почтовым адресам.

Планировщик предоставляет классу `actor_type` следующие функции:

- `new_actor` — помещение в очередь запроса на создание нового актера с возвратом его уникального почтового адреса, вызывается актером при выполнении операции `create`;

- `new_task` — помещение в очередь запроса на доставку сообщения, вызывается в результате операции `send`.

Поскольку код планировщика распараллелен с помощью директив OpenMP, доступ к очередям из разных актеров защищен критическими секциями. Очереди для разных запросов разные, поэтому для каждой из них задана критическая секция со своим именем.

Класс планировщика используется клиентским кодом и предоставляет ему следующие функции:

- `system` — возвращает ссылку на объект с интерфейсом актера, посредством использования которого клиентский код может осуществлять начальную инициализацию системы актеров;

- `evolve` — осуществляет выполнение полного жизненного цикла системы, начиная с создания первых актеров и заканчивая уничтожением всех актеров, созданных в процессе работы системы.

В классе планировщика объявлен объект `m_origin`, представляющий псевдо-актера — некий несуществующий представитель системы актеров, которую будет поддерживать текущий планировщик. Этот объект привязан к планировщику, но не привязан ни к одному арбитру, поскольку у него нет почтового адреса. Ссылка на этот объект возвращается функцией `system`. Его задача — предоставить клиентскому коду возможность вызова функций `create` и `send` на этапе начальной инициализации системы актеров, а не в качестве реакции на пришедшее сообщение.

Функция `evolve` выполняется до тех пор, пока в системе есть необработанные сообщения. Перед выполнением действий актерами планировщик выполняет обработку запросов на создание актеров и доставку сообщений. При обработке запроса на создание актера создается новый арбитр, указатель на который сохраняется в массиве `m_arbiterlist`. Из каждого запроса на доставку сообщения извлекается почтовый адрес назначения, после чего запрос помещается в почтовый ящик арбитра, доступ к которому получен путем индексации в массиве `m_arbiterlist` по этому адресу.

Выполнение обработки сообщений осуществляется в цикле по всем арбитрам, который распараллелен с помощью директивы OpenMP. В теле цикла последовательно выполняется проверка пустоты почтового ящика, попытка извлечения очередного сообщения и, при успешном результате, его обработка. Проверка наличия сообщений в ящике арбитра производится перед обработкой сообщения, а не после. В противном случае могла бы возникнуть такая ситуация, что после обработки сообщения почтовый ящик окажется пустым, и, если остальные ящики также пусты, система завершит работу. Однако во время последней обработки актер мог послать новые сообщения, которые в такой ситуации не будут обработаны. Выполнение же проверки пустоты ящика перед обработкой сообщения гарантирует, что если во всех ящиках не было сообщений, их нет и в системе, что означает, что она может завершать работу.

### Некоторые примеры актеров

Приведем некоторые примеры определений поведения актеров, функционирующих на основе описанных классов. Одним из базовых типов актеров, использованных, к приме-

ру, при реализации стека в [48, 70], является актер `forwarder`. В его задачи входит перенаправление всех принятых сообщений другому актеру, адрес которого получен им при инициализации:

```
def forwarder (cust) [msg]
    send [msg] to cust
end def
```

Описание поведения такого актера на основе приведенного набора классов может быть представлено следующим фрагментом:

```
// актер forwarder
class forwarder_type: public actor_type
{
private:
 address_type m_address;

public:
 struct init_type
 {
  address_type address;
 };

 forwarder_type(const init_type &init):
  m_address(init.address)
 {
  add_action<forwarder_type, unknown_type>();
 }
 void action(const unknown_type &msg)
 {
  send(m_address, msg);
 }
};
```

Актер регистрирует один обработчик для сообщений типа `unknown_type`, в результате чего ему на обработку будут передаваться все пришедшие сообщения. В обработчике вызывается функция отправки сообщения на другой адрес, полученный из параметров инициализации.

Также в [48] в некоторых ситуациях, к примеру, в качестве завершающего элемента списка при реализации стека, используется актер `sink`. Его задача — игнорировать все пришедшие сообщения:

```
def sink () [msg]
end def
```

Класс, представляющий поведение такого актера, также основан на использовании типа `unknown_type`:

```
// актер sink
class sink_type: public actor_type
{
public:
 sink_type(const empty_type &init)
 {
  add_action<sink_type, unknown_type>();
 }
 void action(const unknown_type &msg)
```

```
 {}
};
```

Наконец, в дальнейших примерах нами не раз будет использован актер `print`. Он может принимать сообщения нескольких типов, выводя содержимое каждого принятого сообщения:

```
// актер print
class print_type: public actor_type
{
public:
 struct strmsg_type
 {
  enum { MAX_SIZE = 4096 };
  char str[MAX_SIZE];
 };

 print_type(const empty_type &init)
 {
  add_action<print_type, int>();
  add_action<print_type, double>();
  add_action<print_type, strmsg_type>();
 }
 void action(const int &msg)
 {
  synprintf(stdout, "print int: %d\n", msg);
 }
 void action(const double &msg)
 {
  synprintf(stdout, "print dbl: %f\n", msg);
 }
 void action(const strmsg_type &msg)
 {
  synprintf(stdout, "print str: %s\n", msg.str);
 }
};
```

С использованием этого актера можем описать код, реализующий работу приведенной ранее простой системы актеров, схема которой изображена на рис. 5.1:

```
struct message_type
{
 int arg1, arg2;
 address_type cust;
};
class summator_type: public actor_type
{
public:
 summator_type(const empty_type &init)
 {
  add_action<summator_type, message_type>();
 }
 void action(const message_type &msg)
 {
  send(msg.cust, msg.arg1 + msg.arg2);
 }
```

```
};

// ...

// регистрация актеров в фабрике
factory_type factory;
factory.add_definition<summator_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();

// планировщик
scheduler_type sched;
// инициализация системы актеров
message_type msg = {
 2, 2,
 sched.system().create<print_type>()
};
sched.system().send(
 sched.system().create<summator_type>(),
 msg);
// жизненный цикл системы актеров
sched.evolve(factory);
```

В рамках начальной инициализации этой системы создаются два актера — `summator` и `print`. После этого формируется и отправляется актеру `summator` сообщение, в котором передаются данные для суммирования, а также адрес актера, которому надлежит отправить результат. В качестве него передается адрес только что созданного актера `print`.

### 5.2.2. Многопроцессный вариант

Выше была описана последовательная реализация модели актеров, распараллеленная с помощью директив OpenMP. Теперь приведем пример, как набор классов из приложения Д может быть модифицирован для распараллеливания работы между несколькими процессами с помощью интерфейса MPI.

Работа актеров должна быть распределена между процессами по возможности равномерно. В связи с этим по мере создания новых актеров будем циклически распределять соответствующих арбитров по процессам группы коммуникатора `MPI_COMM_WORLD`. Будучи созданным в некотором процессе арбитр далее своего размещения не меняет, т.е. на весь жизненный цикл системы остается работать в этом процессе.

При межпроцессном распараллеливании мы уже не можем использовать в качестве почтового адреса, как раньше, просто порядковый номер созданного актера во всей системе. Препятствием является тот факт, что вследствие отсутствия общих данных процессы не могут иметь один счетчик, доступный во всех процессах. В то же время, использовать в каждом процессе свой счетчик в качестве значений новых адресов в общем случае некорректно, поскольку тогда адреса потеряют уникальность. В связи с этим мы зададим в качестве адреса совокупность (`process_rank`, `actor_number`), где первый элемент — ранг процесса, которым была выполнена соответствующая операция `create`, второй — порядковый номер созданного актера в рамках этого процесса. Таким образом, для описания адресов актеров используем следующий тип:

```
// почтовый адрес актера
class address_type
{
 friend class dispatcher_type;
```

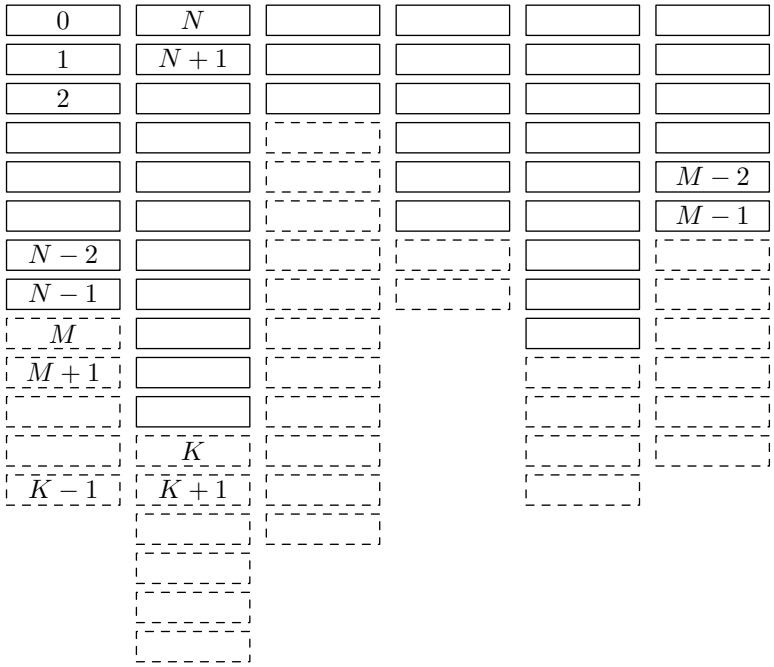



Рис. 5.4. Структура данных для отображения адреса актера на его размещение

```
 int rank, num;
public:
 friend
 bool operator ==(const address_type &l, const address_type &r)
 {
  return l.rank == r.rank && l.num == r.num;
 }
};
```

Следует отметить, что указание в адресе ранга создавшего процесса необходимо лишь для обеспечения уникальности адресов. Из того, что в адресе указан ранг конкретного процесса не следует, что соответствующий арбитр физически будет размещен в этом процессе. Если бы актеры, созданные некоторым процессом, размещались в том же процессе, могла бы возникнуть ситуация, что все актеры работают в одном процессе, пока остальные процессы простаивают. Поскольку такая ситуация неприемлема, нам потребуется некая структура данных, выполняющая отображение адресов актеров в реальное их размещение, т.е. в номер процесса, в котором размещен соответствующий арбитр, и порядковый номер арбитра в рамках содержащего его процесса.

В качестве такой структуры данных выбран массив размером в количество процессов, элементы которого — массивы переменной длины из целых чисел (рис. 5.4). В каждом из них содержатся абсолютные номера актеров, причем эти номера уникальны в рамках всей структуры. Два поля почтового адреса актера являются индексами соответствующих массивов в этой структуре данных, что обеспечивает время вычисления абсолютного номера любого актера по его адресу, не зависящее от размеров структуры.

Поскольку неизвестно, какие адреса будет использовать клиентский код в конкретно взятом процессе, копия этой структуры должна содержаться во всех. Для поддержания ее актуальности после каждого цикла работы актеров она должна пополняться новыми

абсолютными номерами для новых только что созданных адресов (сплошными линиями на рис. 5.4 обозначено содержимое структуры на некотором шаге, пунктиром — пополнение на следующем).

По назначенному абсолютному номеру каждый арбитр в соответствии с циклическим распределением получает ранг процесса, в котором он будет физически размещен, а также свой локальный номер в рамках этого процесса. На основе известной информации о том, к какому процессу относится тот или иной адрес, запросы на создание актера и доставку сообщения из соответствующих очередей `m_actorqueue` и `m_taskqueue` каждого процесса должны быть переправлены в соответствующий целевой процесс. Ответственность за взаимодействие между процессами возложим на объект-диспетчер, для чего добавим еще один класс. На диспетчере будут лежать три основные задачи: поддержка структуры отображения адресов в абсолютные номера, распределение запросов между процессами и проверка занятости системы актеров. Приведем код такого класса:

```
// диспетчер, клиентскому коду интерфейса не предоставляет
class dispatcher_type
{
private:
 typedef actor_type::chunk_type chunk_type;

 // тип пересылаемого запроса
 enum { REQ_CREATE, REQ_SEND };
 // структура запроса
 struct request_type
 {
  int type;
  address_type addr;
  std::string id;
  chunk_type body;
 };

 // тип упакованных данных
 typedef std::vector<char> pack_type;
 // тип очереди запросов
 typedef std::list<request_type> queue_type;
 // смещения в заголовке упакованного запроса
 enum {
  HDR_TYPE,
  HDR_ADDRRANK,
  HDR_ADDRNUM,
  HDR_IDLEN,
  HDR_BODYLEN,
  HEADERLENGTH
 };

 // количество процессов и ранг текущего
 int m_size, m_rank;
 // отображение адресов на абсолютные номера арбитров
 std::vector<std::vector<int> > m_addrmap;
 // счетчики созданных адресов в текущем процессе и во всех
 int m_count, m_full;
 // выходные очереди
 std::vector<queue_type> m_sque;
 // входная очередь
```

```
queue_type m_rque;

// упаковка очереди для отправки
static
pack_type packque(const queue_type &queue)
{
 pack_type pack;
 int pos = 0;
 queue_type::const_iterator it;
 for (it = queue.begin(); it != queue.end(); ++it)
 {
  const request_type &req = *it;
  // формируем заголовок пакета
  int hdr[HEADERLENGTH] = {0};
  hdr[HDR_TYPE] = req.type;
  hdr[HDR_ADDRRANK] = req.addr.rank;
  hdr[HDR_ADDRNUM] = req.addr.num;
  hdr[HDR_IDLEN] = req.id.size();
  hdr[HDR_BODYLEN] = req.body.size();
  // вычисляем полный размер пакета
  int full = 0, size;
  MPI_Pack_size(
   HEADERLENGTH, MPI_INT,
   MPI_COMM_WORLD, &size);
  full += size;
  MPI_Pack_size(
   req.id.size(), MPI_CHAR,
   MPI_COMM_WORLD, &size);
  full += size;
  MPI_Pack_size(
   req.body.size(), MPI_BYTE,
   MPI_COMM_WORLD, &size);
  full += size;
  // формируем пакет и заполняем его данными
  pack.resize(pack.size() + full);
  MPI_Pack(
   &hdr, HEADERLENGTH, MPI_INT,
   &pack.front(), pack.size(), &pos,
   MPI_COMM_WORLD);
  // вносим строку, если она не пуста
  if (hdr[HDR_IDLEN] > 0)
  {
   MPI_Pack(
    (void *) req.id.data(), req.id.size(), MPI_CHAR,
    &pack.front(), pack.size(), &pos,
    MPI_COMM_WORLD);
  };
  // вносим тело, также если не пусто
  if (hdr[HDR_BODYLEN] > 0)
  {
   MPI_Pack(
    (void *) &req.body.front(), req.body.size(), MPI_BYTE,
    &pack.front(), pack.size(), &pos,
    MPI_COMM_WORLD);
  };
```

```
  };
 return pack;
}

// распаковка принятых данных и формирование очереди
static
queue_type unpackque(const pack_type &pack)
{
 queue_type queue;
 int pos = 0;
 while (size_t(pos) < pack.size())
 {
  request_type req;
  // распаковываем заголовок
  int hdr[HEADERLENGTH] = {0};
  MPI_Unpack(
   (void *) &pack.front(), pack.size(), &pos,
   &hdr, HEADERLENGTH, MPI_INT,
   MPI_COMM_WORLD);
  // сохраняем данные заголовка
  req.type = hdr[HDR_TYPE];
  req.addr.rank = hdr[HDR_ADDRRANK];
  req.addr.num = hdr[HDR_ADDRNUM];
  // строку формируем через промежуточный буфер
  if (hdr[HDR_IDLEN] > 0)
  {
   std::vector<char> buf(hdr[HDR_IDLEN]);
   MPI_Unpack(
    (void *) &pack.front(), pack.size(), &pos,
    &buf.front(), buf.size(), MPI_CHAR,
    MPI_COMM_WORLD);
   req.id = std::string(buf.begin(), buf.end());
  };
  // тело в байтах получаем напрямую
  if (hdr[HDR_BODYLEN] > 0)
  {
   req.body.resize(hdr[HDR_BODYLEN]);
   MPI_Unpack(
    (void *) &pack.front(), pack.size(), &pos,
    &req.body.front(), req.body.size(), MPI_BYTE,
    MPI_COMM_WORLD);
  };
  queue.push_back(req);
 };
 return queue;
}

// === интерфейс для scheduler_type ===
friend class scheduler_type;
// конструктор
dispatcher_type(void):
 m_count(0), m_full(0)
{
 MPI_Comm_size(MPI_COMM_WORLD, &m_size);
 MPI_Comm_rank(MPI_COMM_WORLD, &m_rank);
```

```
 m_addrmap.resize(m_size);
 m_sque.resize(m_size);
}

// создание нового уникальноого адреса
address_type newaddr(void)
{
 address_type newaddr;
 newaddr.rank = m_rank;
 newaddr.num = m_count++;
 return newaddr;
}
// межпроцессная репликация адресов
void replicate(void)
{
 // собираем счетчики из всех процессов
 std::vector<int> allcnt(m_size);
 MPI_Allgather(
  &m_count, 1, MPI_INT,
  &allcnt.front(), 1, MPI_INT,
  MPI_COMM_WORLD);
 // дополняем карту новыми абсолютными номерами
 for (int rank = 0; rank < m_size; ++rank)
 {
  int oldcnt = m_addrmap[rank].size();
  m_addrmap[rank].resize(allcnt[rank]);
  for (int i = oldcnt; i < allcnt[rank]; ++i)
   m_addrmap[rank][i] = m_full++;
 };
}
// размещение (ранг процесса) арбитра с заданным адресом
int location(const address_type &address) const
{
 assert(address.rank >= 0 && address.rank < m_size);
 assert(
  address.num >= 0 &&
  size_t(address.num) < m_addrmap[address.rank].size());
 return m_addrmap[address.rank][address.num] % m_size;
}
// номер арбитра в содержащем его процессе
int index(const address_type &address) const
{
 assert(location(address) == m_rank);
 return m_addrmap[address.rank][address.num] / m_size;
}

// помещение запроса в одну из выходных очередей
void send(const request_type &req)
{
 m_sque[location(req.addr)].push_back(req);
}
// изъятие запроса из входной очереди
bool recv(request_type &req)
{
 bool rc = !m_rque.empty();
```

```
  if (rc)
  {
   req = m_rque.front();
   assert(location(req.addr) == m_rank);
   m_rque.pop_front();
  };
  return rc;
}
// межпроцессный обмен запросами
// опустошаются выходные очереди, заполняется входная
void dispatch(void)
{
 // отправляемые и принимаемые данные
 pack_type sdata, rdata;
 // размеры и смещения отправляемых порций данных
 std::vector<int> ssize(m_size), spos(m_size);
 // размеры и смещения принимаемых порций данных
 std::vector<int> rsize(m_size), rpos(m_size);

 // пробегаем по всем выходным очередям
 for (int rank = 0; rank < m_size; ++rank)
 {
  // пакуем и чистим очередь
  pack_type pack = packque(m_sque[rank]);
  m_sque[rank].clear();
  // добавляем к отправляемым данным
  sdata.insert(sdata.end(), pack.begin(), pack.end());
  // сохраняем размер
  ssize[rank] = pack.size();
  // и смещение от начала отправляемых данных
  spos[rank] = !rank ? 0 : spos[rank - 1] + ssize[rank - 1];
 };
 // производим обмен размерами отправляемых порций данных
 // результат - размеры принимаемых порций данных
 MPI_Alltoall(
  &ssize.front(), 1, MPI_INT,
  &rsize.front(), 1, MPI_INT,
  MPI_COMM_WORLD);
 // формируем массив и смещения для принятия данных
 for (int rank = 0; rank < m_size; ++rank)
 {
  rdata.resize(rdata.size() + rsize[rank]);
  rpos[rank] = !rank ? 0 : rpos[rank - 1] + rsize[rank - 1];
 };

 // добавляем по байту для безопасного обращения к front()
 sdata.push_back(0);
 rdata.push_back(0);
 // производим обмен данными между всеми процессами
 MPI_Alltoallv(
  &sdata.front(), &ssize.front(), &spos.front(), MPI_PACKED,
  &rdata.front(), &rsize.front(), &rpos.front(), MPI_PACKED,
  MPI_COMM_WORLD);
 // убираем лишний байт
 sdata.pop_back();
```

```
  rdata.pop_back();

  // распаковываем и сохраняем принятую очередь запросов
  queue_type rque = unpackque(rdata);
  m_rque.insert(m_rque.end(), rque.begin(), rque.end());
 }

 // проверка занятости системы
 bool busy(bool b) const
 {
  // объединение признаков занятости из всех процессов
  int localbusy = b ? 1 : 0, allbusy;
  MPI_Allreduce(
   &localbusy, &allbusy, 1, MPI_INT,
   MPI_LOR, MPI_COMM_WORLD);
  return allbusy != 0;
 }
};
```

Данные отображения адресов в абсолютные номера арбитров хранятся в структуре `m_addrmap`. Она соответствует той, что изображена на рис. 5.4. Также в объекте диспетчера хранятся счетчики, необходимые для пополнения этой структуры новыми данными — счетчик созданных адресов в текущем процессе и счетчик созданных адресов во всех процессах. Для работы с этой структурой планировщику предоставляются следующие функции:

- `newaddr` — создание нового адреса, при этом увеличивается внутренний счетчик созданных адресов;
- `replicate` — обновление структуры `m_addrmap`, в рамках которой происходит получение текущих локальных счетчиков из всех процессов и пополнение `m_addrmap` на их основе;
- `location` — получение ранга процесса, в котором уже размещен или будет размещен арбитр с заданным адресом;
- `index` — получение локального номера арбитра в рамках содержащего его процесса.

Функция `newaddr` вызывается планировщиком в рамках выполнения клиентским кодом операций `create`. При этом меняется только локальный счетчик созданных адресов, структура `m_addrmap` не меняется. Пересылка в соответствующий процесс запросов на создание и, возможно, доставку сообщений производится планировщиком позже, при этом, разумеется, происходят обращения к функциям `location` и `index`. Чтобы они отработали корректно, необходимо, чтобы к этому моменту структура `m_addrmap` уже содержала необходимые данные, т.е. перед распределением запросов необходим межпроцессный вызов функции `replicate`. В результате выполнения последней все процессы содержат одинаковые экземпляры структуры `m_addrmap`. В этой функции происходит последовательное присваивание абсолютных номеров новым адресам, информация о которых получена из всех процессов с помощью функции `MPI_Allgather`. Эта работа, в принципе, может быть распараллелена между процессами. Однако, во-первых, довольно тяжело ее распараллелить равномерно при неравномерном создании процессами новых актеров, во-вторых, подобная работа — хороший пример не обремененной вычислениями задачи, которую в большинстве случаев быстрее выполнить локально, чем распределенно, учитывая время на коммуникации.

Для распределения запросов между процессами диспетчер предоставляет планировщику следующие функции:

- `send` — помещение в выходную очередь запроса на создание актера или доставку сообщения;
- `dispatch` — сбор данных из всех выходных очередей, распределение на соответствующие процессы, формирование входной очереди на основе принятых от других процессов данных;
- `recv` — получение очередного запроса из входной очереди.

Диспетчер содержит внутри себя набор выходных очередей запросов, по одной для каждого процесса в группе. При выполнении функции `send` диспетчер на основе изъятого из запроса адреса получает номер процесса, в который предстоит отправить запрос, после чего помещает запрос в соответствующую очередь. Функция `dispatch` выполняет одновременное распределение данных из всех процессов во все процессы с помощью функции `MPI_Alltoallv`. Перед отправкой все выходные очереди запросов упаковываются и формируется исходящий массив байтов `sdata`. В это же время формируются массивы размеров порций данных, отправляемых в каждый процесс, и их смещений относительно начала `sdata`. Поскольку количество данных, принимаемых от каждого процесса, заранее неизвестно, предварительно выполняется обмен размерами отправляемых данных с помощью функции `MPI_Alltoall`. На основе полученных размеров формируется приемный буфер `rdata` нужного размера и массив смещений от его начала. После выполнения рассылки данных между всеми процессами полученные данные из `rdata` распаковываются, в результате чего формируется входная очередь запросов. Чтение запросов из нее планировщик осуществляет с помощью функции `recv`.

Для единовременной отправки и приема содержимого очередей используются операции упаковки и распаковки данных (`MPI_Pack` и `MPI_Unpack`). Работа с ними инкапсулирована в функциях `packque` и `unpackque` соответственно. При упаковке каждого запроса из очереди формируется заголовок, состоящий из нескольких целых чисел. Среди них тип запроса (создание актера или доставка сообщения), почтовый адрес, а также размеры областей данных идентификатора и параметра запроса. Сразу за заголовком следуют данные строки идентификатора запроса и данные его параметра.

За межпроцессную проверку занятости системы актеров ответственна функция `busy`. Ей на вход передается признак занятости актеров текущего процесса, на выходе — признак занятости хотя бы одного процесса в группе. Пока есть хотя бы один занятый процесс, работа всей системы должна продолжаться.

Для использования диспетчера в класс планировщика вносятся следующие изменения. Прежде всего, появляется внутренний объект диспетчера, а также меняется функция `new_actor`:

```
class scheduler_type
{
// ...
 // диспетчер запросов
 dispatcher_type m_disp;

 // === интерфейс для actor_type ===
 friend class actor_type;
 // добавление запроса на создание актера
 address_type new_actor(const behaviour_type &behaviour)
 {
  // формирование нового адреса
  address_type newaddr = m_disp.newaddr();
```

```
  // добавление запроса
  m_actorqueue.push_back(
   actorqueue_type::value_type(newaddr, behaviour));
  return newaddr;
 }
 // добавление запроса на доставку сообщения
 void new_task(const task_type &task)
 {
  m_taskqueue.push_back(task);
 }
// ...
};
```

Изменения касаются генерации нового почтового адреса, которой теперь занимается диспетчер. Помимо этого, существенно преобразуется функция `evolve`, поскольку именно на нее, в основном, возлагается задача взаимодействия с диспетчером:

```
class scheduler_type
{
// ...
 // полный цикл развития системы актеров
 void evolve(const factory_type &factory)
 {
  bool busy;
  do
  {
   // репликация новых почтовых адресов
   m_disp.replicate();

   // отправка всех запросов
   while (!m_actorqueue.empty())
   {
    // изымаем из очереди новый адрес и поведение
    dispatcher_type::request_type req = {
     dispatcher_type::REQ_CREATE,
     m_actorqueue.front().first,
     m_actorqueue.front().second.defid,
     m_actorqueue.front().second.chunk
    };
    m_actorqueue.pop_front();
    // передаем запрос диспетчеру
    m_disp.send(req);
   };
   while (!m_taskqueue.empty())
   {
    // изымаем из очереди сообщение
    dispatcher_type::request_type req = {
     dispatcher_type::REQ_SEND,
     m_taskqueue.front().address,
     m_taskqueue.front().patid,
     m_taskqueue.front().chunk
    };
    m_taskqueue.pop_front();
    // передаем запрос диспетчеру
    m_disp.send(req);
   };
```

```
    // распределение всех запросов между процессами
    m_disp.dispatch();

    // получение и обработка всех своих запросов
    dispatcher_type::request_type req;
    while (m_disp.recv(req))
    {
     if (req.type == dispatcher_type::REQ_CREATE)
     {
      // обработка запроса на создание актера
      behaviour_type behaviour = { req.id, req.body };
      // создаем арбитра
      assert(
       size_t(m_disp.index(req.addr)) ==
       m_arbiterlist.size());
      m_arbiterlist.push_back(
       new arbiter_type(factory, req.addr, behaviour, this));
     }
     else
     {
      // обработка запроса на доставку сообщения
      task_type task = { req.addr, req.id, req.body };
      // помещаем его в соответствующий почтовый ящик
      m_arbiterlist[m_disp.index(req.addr)]->deliver(task);
     };
    };

    // цикл работы системы актеров
    busy = false;
    int allnum = m_arbiterlist.size();
    for (int addr = 0; addr < allnum; ++addr)
    {
     // пополняем признак наличия сообщений в системе
     busy = busy || !m_arbiterlist[addr]->empty();
     // пытаемся получить и обработать сообщение
     task_type task;
     if (m_arbiterlist[addr]->retrieve(task))
      m_arbiterlist[addr]->process(task);
    };
    // выход, если в системе нет сообщений
   } while (m_disp.busy(busy));

   // уничтожение арбитров
   int allnum = m_arbiterlist.size();
   for (int addr = 0; addr < allnum; ++addr)
    delete m_arbiterlist[addr];
   m_arbiterlist.clear();
   m_disp = dispatcher_type();
  }
};
```

Работа жизненного цикла системы начинается с операции репликации адресов, поскольку некоторые адреса создаются еще до выполнения функции `evolve`. После ее репликации можно осуществлять отправку запросов, чем и занимаются последующие два цикла выбор-

ки запросов из очередей. Каждый запрос изымается из очереди и переводится в структуру `request_type`, объявленную в классе диспетчера, после чего передается последнему для помещения в выходную очередь. После опустошения обеих очередей запросов в планировщике выполняется межпроцессный вызов `dispatch`, в процессе которого все отправленные запросы пересылаются в процессы назначения, а в текущем процессе формируется входная очередь запросов. Следующим циклом производится выборка запросов из этой очереди, в рамках которой выполняются действия, соответствующие запросам, а именно создание новых арбитров или доставка сообщений существующим. При доставке сообщений используется информация от диспетчера о порядковом номере арбитра в рамках текущего процесса на основе его адреса.

Критерием завершения работы системы является отсутствие процессов с установленным признаком занятости актеров. Для определения наличия таковых используется межпроцессный вызов функции диспетчера `busy`, которой передается локальный признак занятости текущего процесса.

Построенный таким образом набор классов использовать не менее легко, чем код, приведенный в приложении Д. Клиентский код должен быть дополнен лишь вызовами `MPI_Init` и `MPI_Finalize`:

```
// ...

MPI_Init(&argc, &argv);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// регистрация актеров в фабрике
factory_type factory;
factory.add_definition<summator_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();

// планировщик
scheduler_type sched;
if (rank == 0)
{
 // инициализация системы актеров
 message_type msg = {
  2, 2,
  sched.system().create<print_type>()
 };
 sched.system().send(
  sched.system().create<summator_type>(),
  msg);
};
// жизненный цикл системы актеров
sched.evolve(factory);

MPI_Finalize();
```

Здесь приведен пример инициализации и выполнения системы актеров, изображенной на рис. 5.1. Как видно из примера, следует также проследить, чтобы первоначальное создание актеров и отправка им сообщений происходили лишь из одного процесса, иначе будет происходить многократное выполнение задачи системой, что сведет на нет выигрыш от распараллеливания.

### 5.2.3. Низкоуровневая многопоточная реализация

Обе описанные до сего момента реализации являются не совсем корректными, поскольку, по сути, вводят барьерную синхронизацию всех арбитров между последовательными выполнениями действий. Это не вносит существенных ограничений по возможностям моделирования различных систем актеров, поскольку каждый арбитр имеет возможность «пропуска хода», если очередь пуста или выбранное сообщение не подходит текущему актеру. Такой подход является частным случаем недетерминизма модели в отношении времени доставки, т.е. он является одним из возможных, отчего не мешает моделированию различных систем актеров. Однако само по себе наличие общей синхронизации и, как следствие, простоя некоторых арбитров противоречит описанию модели и вносит задержки при выполнении. К примеру, предположим, что у нас есть два актера, один из которых обрабатывает сообщения в два раза быстрее второго, но в его очереди в два раза больше сообщений. Тогда второй актер закончит работу гораздо раньше первого, поскольку первый после каждой обработки своего сообщения будет ожидать завершения обработки сообщения вторым актером. Если же обработка ведется независимо, оба актера могут закончить работу примерно в одно время.

В связи с этим попробуем обойтись без общей синхронизации между действиями актеров. Такая реализация снова потребует от нас использования более низкоуровневых средств, чем использованные до этого. Мы воспользуемся в этих целях программным интерфейсом pthreads.

Зададим схему распараллеливания, при которой планировщик (а именно функция `evolve`) работает в главном потоке, а для каждого нового арбитра создается по одному новому потоку. Разумеется, такой подход окажется недостаточно эффективным при большом количестве арбитров, поскольку вызовет большие накладные расходы на переключение потоков. Однако мы здесь приводим упрощенный вариант, более же сложные подходы могут быть реализованы самостоятельно.

Поток планировщика предоставляет доступ к некоторым своим данным потокам арбитров. Каждый арбитр, в свою очередь, также предоставляет доступ к своим данным планировщику. Потоки арбитров обращаются лишь к данным планировщика, к данным же друг друга они доступа не имеют. Таким образом, мы имеем топологию «звезда» с обращениями в обе стороны (рис. 5.5).

В предыдущих случаях выборка и обработка сообщений всеми арбитрами производилась потактно, и определение факта пустоты всех почтовых ящиков выполнялось на каждом такте путем опроса каждого. В нынешнем случае было бы неправильно реализовывать опрос всех арбитров из главного потока, поскольку время такого опроса пропорционально их количеству. К тому же, поскольку арбитр работает в другом потоке, опрос пустоты его почтового ящика потребует блокировки последнего, что может сильно замедлить цикл опроса всех ящиков в главном потоке. Вместо этого мы будем хранить в планировщике множество адресов занятых арбитров (т.е. арбитров, почтовый ящик которых не пуст). Каждый раз в момент помещения сообщения в ящик арбитра поток планировщика должен помещать в это множество соответствующий почтовый адрес. В свою очередь, между обработками сообщений арбитр проверяет свой почтовый ящик на пустоту. В случае, если он пуст, арбитр в рамках своего потока изымает свой адрес из множества занятых арбитров, хранимого планировщиком. Таким образом, проверка планировщиком факта пустоты всей системы заключается просто в проверке на пустоту множества занятых арбитров.

Для работы этого механизма необходимо обеспечить атомарность двух следующих пар операций:

- помещение планировщиком нового сообщения в почтовый ящик арбитра и добавление

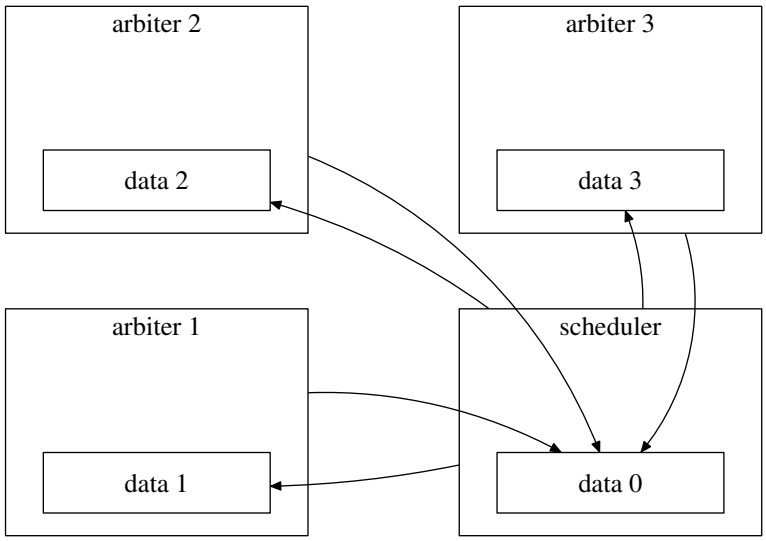



Рис. 5.5. Схема связей планировщика с арбитрами

соответствующего адреса в множество занятых арбитров;

- проверка арбитром пустоты почтового ящика и, в случае положительного результата, извлечение адреса из множества занятых арбитров в планировщике.

В обеих парах операций используются два разделяемых ресурса: множество занятых арбитров, хранимое в планировщике, и содержимое почтового ящика арбитра. Если не обеспечить атомарность обеих этих пар, могут возникать так называемые состояния гонки (race conditions). К примеру, может возникнуть ситуация, когда планировщик поместил адрес арбитра в множество занятых, в это время арбитр завершил выполнение предыдущих своих сообщений и изъял этот адрес, планировщик же добавил ему новое сообщение. Однако множество занятых оказалось пустым, вследствие чего планировщик по ошибке может закончить работу раньше, чем будут обработаны все сообщения в системе. Если же планировщик выполняет помещение адреса в множество занятых после передачи сообщения арбитру, последний может быстро обработать его и, если оно последнее, изъять свой адрес из множества занятых, после чего планировщик снова его туда добавляет. В результате, система давно оказалась пуста, но планировщик думает, что это не так, и вечно ожидает. Обратный пример возникает, если не обеспечить атомарность проверки пустоты ящика и извлечения адреса из множества занятых. Если проверка на пустоту ящика оказывается успешной, поток арбитра извлекает адрес из множества занятых. Если же перед этим извлечением поток планировщика поместит новые сообщения в ящик арбитра, они могут быть не обработаны, поскольку после того, как арбитр извлечет свой адрес из множества занятых, в случае пустоты последнего планировщик завершит работу системы.

Поскольку в обеих перечисленных парах операций используются одновременно два разделяемых ресурса, для обеспечения их атомарности требуется выполнять две блокировки. В такой ситуации может возникнуть взаимоблокировка (deadlock), если проявить неосторожность и не проследить за порядком выполнения блокирования ресурсов. Поскольку обращение к чужим ресурсам в нашей системе происходит в соответствии с топологией «звезда» (рис. 5.5), в нашем случае напрашивается вполне простое решение. Во избежание взаимоблокировки во всех потоках блокирование двух ресурсов должно осуществляться в одной последовательности, т.е. одним из двух вариантов: либо сначала блокируется ре-

сурс арбитра, потом ресурс планировщика, либо наоборот. Мы будем пользоваться первым вариантом.

Помимо множества занятых работ, у планировщика еще три разделяемых ресурса: очередь запросов на создание актеров `m_actorqueue`, очередь запросов на доставку сообщений `m_taskqueue` и массив указателей арбитров `m_arbiterlist`. Доступ к этим ресурсам также регулируется общей блокировкой данных планировщика.

Воспользуемся интерфейсом pthreads для многопоточного распараллеливания приведенных в приложении Д классов. Изменять нам потребуется лишь содержимое класса `scheduler_type`. Прежде всего, добавляется определение структуры данных, передаваемой каждому потоку арбитра, а также новые данные планировщика:

```
class scheduler_type
{
// ...
private:
 struct thrparam_type
 {
  scheduler_type *sched;
  const factory_type *factory;
  address_type address;
  behaviour_type behaviour;
  // объекты синхронизации арбитра
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  // признак завершения работы
  bool finish;
 };
 // объекты синхронизации планировщика
 pthread_mutex_t m_mutex;
 pthread_cond_t m_cond;
 // множество занятых арбитров
 std::set<address_type> m_busy;
// ...
};
```

Среди новых данных добавляются объекты синхронизации доступа к разделяемым данным планировщика и множество адресов занятых арбитров `m_busy`. Для инициализации и уничтожения объектов синхронизации пополняется конструктор и добавляется деструктор:

```
class scheduler_type
{
// ...
public:
 scheduler_type(void)
 {
  chkzero(::pthread_mutex_init(&m_mutex, NULL));
  chkzero(::pthread_cond_init(&m_cond, NULL));
  // псевдо-актер привязан только к планировщику
  // арбитра у него нет, как и почтового адреса
  m_origin.bind(this, NULL);
 }
 ~scheduler_type(void)
 {
  chkzero(::pthread_cond_destroy(&m_cond));
  chkzero(::pthread_mutex_destroy(&m_mutex));
```

```
 }
// ...
};
```

Меняются функции доступа к очередям запросов на создание актеров и доставку сообщений:

```
class scheduler_type
{
// ...
private:
 // === интерфейс для actor_type ===
 friend class actor_type;
 // добавление запроса на создание актера
 address_type new_actor(const behaviour_type &behaviour)
 {
  chkzero(::pthread_mutex_lock(&m_mutex));
  // формирование нового адреса
  address_type newaddr = m_arbiterlist.size() + m_actorqueue.size();
  // добавление запроса
  m_actorqueue.push_back(
   actorqueue_type::value_type(newaddr, behaviour));
  chkzero(::pthread_cond_signal(&m_cond));
  chkzero(::pthread_mutex_unlock(&m_mutex));
  return newaddr;
 }
 // добавление запроса на доставку сообщения
 void new_task(const task_type &task)
 {
  chkzero(::pthread_mutex_lock(&m_mutex));
  m_taskqueue.push_back(task);
  chkzero(::pthread_cond_signal(&m_cond));
  chkzero(::pthread_mutex_unlock(&m_mutex));
 }
// ...
};
```

Помимо этого, добавляется статическая функция потока, в рамках которого выполняется работа арбитра:

```
class scheduler_type
{
// ...
private:
 // проверка кода возврата функций pthread_xxx
 static
 void chkzero(int retcode) { assert(retcode == 0); }

 // функция потока, в рамках которого работает арбитр
 static
 void *thr_proc(void *param)
 {
  thrparam_type &p = *static_cast<thrparam_type *>(param);
  chkzero(::pthread_mutex_init(&p.mutex, NULL));
  chkzero(::pthread_cond_init(&p.cond, NULL));
```

```
  // создаем арбитра
  arbiter_type *arbiter = new arbiter_type(
   *p.factory,
   p.address,
   p.behaviour,
   p.sched);

  // сохраняем указатель и сообщаем планировщику
  chkzero(::pthread_mutex_lock(&p.sched->m_mutex));
  p.sched->m_arbiterlist[p.address] = arbiter;
  chkzero(::pthread_cond_signal(&p.sched->m_cond));
  chkzero(::pthread_mutex_unlock(&p.sched->m_mutex));

  bool finish = false;
  do
  {
   chkzero(::pthread_mutex_lock(&p.mutex));
   // пытаемся получить и обработать сообщение
   task_type task;
   if (arbiter->retrieve(task))
   {
    // на время обработки отпускаем блокировку
    chkzero(::pthread_mutex_unlock(&p.mutex));
    arbiter->process(task);
    chkzero(::pthread_mutex_lock(&p.mutex));
   };
   // если ящик пуст
   if (arbiter->empty())
   {
    // исключаем себя из списка занятых
    chkzero(::pthread_mutex_lock(&p.sched->m_mutex));
    p.sched->m_busy.erase(p.address);
    chkzero(::pthread_cond_signal(&p.sched->m_cond));
    chkzero(::pthread_mutex_unlock(&p.sched->m_mutex));
    // и ожидаем завершения или прихода новых сообщений
    while (!p.finish && arbiter->empty())
     chkzero(::pthread_cond_wait(&p.cond, &p.mutex));
    // в рамках блокировки читаем признак завершения
    finish = p.finish;
   };
   chkzero(::pthread_mutex_unlock(&p.mutex));
  } while (!finish);

  // при завершении уничтожаем арбитра
  delete arbiter;

  chkzero(::pthread_cond_destroy(&p.cond));
  chkzero(::pthread_mutex_destroy(&p.mutex));
  return NULL;
 }
// ...
};
```

В начале этой функции производится инициализация объектов синхронизации доступа к разделяемым данным арбитра, который будет работать в рамках текущего потока, и

создание этого арбитра. Указатель на созданного арбитра попадает в массив, хранимый планировщиком. Поскольку этот массив входит в разделяемые данные планировщика, доступ к нему оборачивается соответствующей блокировкой. После этого выполняется цикл работы арбитра.

Содержимое тела цикла оборачивается блокировкой данных арбитра. Блокировка распространяется на использование функций арбитра `retrieve` и `empty`, поскольку они работают с разделяемым ресурсом — содержимым почтового ящика. Функция `process` не общается с почтовым ящиком, поэтому на время ее выполнения блокировка арбитра временно снимается. Это важно, поскольку в рамках этой функции выполняются действия, задаваемые клиентским кодом, т.е. как таковые вычислительные действия актеров. Если бы эти действия выполнялись также в рамках блокировки, это могло бы создать «узкое горло» при выполнении, поскольку планировщик во время распределения сообщений между арбитрами вынужден был бы ожидать завершения каждого длительного действия соответствующим арбитром.

В случае, если после получения и, возможно, обработки сообщения почтовый ящик оказался пустым, выполняется изъятие своего адреса из множества занятых арбитров. Это множество относится к данным планировщика, поэтому на время изъятия выполняется блокирование его данных. После этого выполняется ожидание прихода новых сообщений или установки признака завершения работы системы. Чтение признака о необходимости завершения потока производится внутри области блокирования данных арбитра, поскольку в противном случае в потоке планировщика может возникнуть обращение объектам синхронизации арбитра после того, как они уже уничтожены. После выполнения цикла производится уничтожение текущего арбитра и его объектов синхронизации.

Наконец, изменений требует функция развития системы актеров:

```
class scheduler_type
{
// ...
public:
 // полный цикл развития системы актеров
 void evolve(const factory_type &factory)
 {
  // идентификаторы потоков
  std::vector<pthread_t> thr;
  // структуры параметров потоков
  std::deque<thrparam_type> thrparam;

  chkzero(::pthread_mutex_lock(&m_mutex));
  do
  {
   // обработка запросов на создание актеров
   while (!m_actorqueue.empty())
   {
    // изымаем из очереди адрес и соответствующий запрос
    address_type newaddr = m_actorqueue.front().first;
    behaviour_type behaviour = m_actorqueue.front().second;
    m_actorqueue.pop_front();
    // резервируем место для указателя на арбитра
    assert(size_t(newaddr) == m_arbiterlist.size());
    m_arbiterlist.push_back(NULL);
    // отпускаем блокировку
    chkzero(::pthread_mutex_unlock(&m_mutex));
```

```
    // резервируем место для параметров потока
    assert(size_t(newaddr) == thr.size());
    thr.push_back(pthread_t());
    assert(size_t(newaddr) == thrparam.size());
    thrparam.push_back(thrparam_type());
    // заполняем параметры потока
    thrparam[newaddr].sched = this;
    thrparam[newaddr].factory = &factory;
    thrparam[newaddr].address = newaddr;
    thrparam[newaddr].behaviour = behaviour;
    thrparam[newaddr].finish = false;

    // создаем поток
    chkzero(::pthread_create(
     &thr[newaddr], NULL, thr_proc, &thrparam[newaddr]));
    // забираем блокировку и ждем создания арбитра
    chkzero(::pthread_mutex_lock(&m_mutex));
    while (!m_arbiterlist[newaddr])
      chkzero(::pthread_cond_wait(&m_cond, &m_mutex));
  };

  // обработка запросов на доставку сообщений
  taskqueue_type skipqueue;
  while (!m_taskqueue.empty())
  {
    // изымаем сообщение
    task_type task = m_taskqueue.front();
    m_taskqueue.pop_front();
    // если соответствующий арбитр уже создан
    if (size_t(task.address) < m_arbiterlist.size())
    {
      // получаем его атрибуты
      arbiter_type *arbiter = m_arbiterlist[task.address];
      thrparam_type &p = thrparam[task.address];

      // временно разрываем блокировку планировщика
      chkzero(::pthread_mutex_unlock(&m_mutex));
      // блокируем арбитра и снова планировщика
      chkzero(::pthread_mutex_lock(&p.mutex));
      chkzero(::pthread_mutex_lock(&m_mutex));

      // помещаем сообщение в почтовый ящик
      arbiter->deliver(task);
      m_busy.insert(task.address);
      chkzero(::pthread_cond_signal(&p.cond));

      // отпускаем арбитра
      chkzero(::pthread_mutex_unlock(&p.mutex));
    }
    else
      // если арбитр еще не создан, откладываем запрос
      skipqueue.push_back(task);
  };
  m_taskqueue.swap(skipqueue);
```

```
    // ожидаем освобождения системы или прихода запроса
    while (!m_busy.empty() && m_actorqueue.empty() && m_taskqueue.empty())
     chkzero(::pthread_cond_wait(&m_cond, &m_mutex));

    // выход, если в системе нет сообщений
   } while (!m_busy.empty() ||!m_actorqueue.empty() ||!m_taskqueue.empty());
   chkzero(::pthread_mutex_unlock(&m_mutex));

   // взводим признаки для всех потоков о завершении работы
   int allnum = m_arbiterlist.size();
   for (int addr = 0; addr < allnum; ++addr)
   {
    thrparam_type &p = thrparam[addr];
    chkzero(::pthread_mutex_lock(&p.mutex));
    p.finish = true;
    chkzero(::pthread_cond_signal(&p.cond));
    chkzero(::pthread_mutex_unlock(&p.mutex));
   };
   // ждем завершения всех потоков
   for (int addr = 0; addr < allnum; ++addr)
    chkzero(::pthread_join(thr[addr], NULL));
   m_arbiterlist.clear();
  }
};
```

В начале функции создаются два контейнера для хранения идентификаторов созданных потоков и структур параметров, передаваемых им при создании. Оба контейнера обеспечивают возможность индексации по почтовому адресу арбитра. В качестве контейнера для хранения идентификаторов потоков мы выбрали `std::vector`. Для хранения структур параметров мы не можем сделать тот же выбор, поскольку при добавлении в `std::vector` новых элементов может происходить перевыделение памяти, в связи с чем ранее внесенные элементы могут переместиться в памяти на новое место. Постоянство адреса структуры параметров в памяти важно, поскольку мы передаем его создаваемому потоку. В связи с этим в качестве хранилища структур параметров мы выбрали контейнер `std::deque`, поскольку в соответствии со спецификацией добавление новых элементов в любой из его концов не меняет размещение элементов, внесенных ранее.

Весь цикл развития системы актеров охватывается блокировкой данных планировщика, поскольку обращение к ним происходит в критерии завершения цикла. Внутри цикла блокировка регулярно снимается для выполнения тех или иных действий, не затрагивающих ресурсов, используемых совместно разными потоками. В частности, перед созданием нового потока арбитра формируется структура его параметров. Эта структура добавляется в контейнер, который используется лишь потоком планировщика, поэтому фрагмент ее создания и заполнения, также как и создание соответствующего потока, вынесен из области блокирования планировщика, тем самым давая возможность потокам арбитров добавить новые запросы в соответствующие очереди.

В массиве указателей `m_arbiterlist` резервируется место для указателя на нового создаваемого арбитра. Сразу после создания потока осуществляется ожидание появления соответствующего указателя, который будет занесен туда потоком арбитра. Это гарантирует нам, что его объекты синхронизации уже созданы, и последующий код планировщика может к ним обращаться.

В цикле распределения сообщений выполняется проверка существования арбитра с соответствующим адресом назначения. Если этот арбитр еще не создан, сообщение должно

быть пропущено. С этой целью используется вспомогательная очередь `skipqueue`, в которую помещаются пропускаемые сообщения. Причину возникновения ситуации, когда планировщик обрабатывает сообщение для некоторого арбитра раньше, чем запрос на создание этого арбитра, мы обсудим позже.

Если арбитр, которому предназначено сообщение, уже создан, сообщение помещается в его почтовый ящик с помощью функции `deliver`, а его адрес добавляется в список занятых арбитров. При этом должно быть выполнено блокирование и соответствующего арбитра, и планировщика. Выше было оговорено, что во избежание взаимоблокировки в такой ситуации блокирование должно производиться во всех потоках в одном порядке: сначала блокирование арбитра, потом блокирование планировщика. С этой целью на время блокирования арбитра производится «разрыв» области блокирования планировщика. Использование такого разрыва не является корректным во всех ситуациях. К примеру, если бы мы использовали итератор, указывающий на какой-либо элемент из очереди `m_taskqueue`, а другие потоки могли удалять произвольные элементы из этой очереди, то после такого разрыва в общем случае использовать итератор было бы нельзя. В нашем же случае этот разрыв не разбивает никаких последовательностей действий, выполнение которых должно быть атомарным, таких как проверка очереди на пустоту и изъятие ее головного элемента, а потому в данном случае он допустим.

В момент разрыва области блокирования планировщика может произойти пополнение очередей запросов, в том числе могут прийти запросы на создание новых актеров и на доставку им же сообщений. Поскольку планировщик еще не покинул цикл обработки запросов на доставку, он продолжит выборку запросов из очереди и на следующих итерациях наткнется на те, которые адресованы еще не созданным арбитрам, запросы на создание которых как раз сейчас содержатся в очереди `m_actorqueue`. Именно такие запросы попадают в очередь пропуска `skipqueue` и обрабатываются в следующем цикле.

Завершение работы системы осуществляется в случае, если нет занятых арбитров и пусты обе очереди запросов. Каждый поток арбитра получает информацию о завершении по признаку в своей структуре параметров. Для каждого потока арбитра объявлен свой признак завершения для того, чтобы обеспечить возможность внести его взведение в область блокирования данных арбитра. Как уже говорилось выше, это делается для того, чтобы избежать обращения к объектам синхронизации арбитра из потока планировщика после того, как они уже уничтожены в потоке арбитра. Такая ситуация могла бы возникнуть при объявлении единого признака завершения в рамках планировщика. Если после его взведения вне области блокирования в цикле производить сигнализацию арбитрам, может возникнуть ситуация, в которой некоторый арбитр уже прочел признак (раньше сигнализации) и завершился, уничтожив объекты синхронизации. Поток арбитра, как мы видели выше, читает признак также в области блокирования, что гарантирует нам тот факт, что снятие блокировки арбитра из потока планировщика после взведения признака завершения арбитра будет последним обращением к соответствующим объектам синхронизации перед их уничтожением.

Следует отметить, что приведенная реализация порождает немало накладных расходов на переключение потоков при очень большом количестве актеров. Более оптимальным было бы использование пула потоков некоторого фиксированного размера, в рамках которых выполнялись бы действия всех актеров. Такой вариант, по сути, является комбинацией приведенной здесь реализации и последовательной версии, описанной ранее.

Другой момент, требующий оптимизации, заключается в следующем. Каждый арбитр имеет один разделяемый ресурс — содержимое почтового ящика. У планировщика же их четыре, а именно: массив указателей созданных арбитров, множество занятых арбитров и две очереди запросов. Доступ ко всем этим ресурсам регулируется одним объектом син-

хронизации, что приводит к наличию ненужных ожиданий между потоками арбитров, осуществляющих попытки одновременного доступа к разным ресурсам планировщика, к примеру, к разным очередям запросов. Чтобы этого не происходило, следует на каждый ресурс выделить свои объекты синхронизации, правильно организовав работу с ними.

### 5.2.4. Поддержка вложенных подсистем актеров

В этом разделе рассмотрим один из возможных вариантов расширения модели, который касается выполнения вложенных подсистем актеров.

Одной из характеристик классической модели актеров является монотонное возрастание системы. Новые актеры в системе постоянно создаются, однако старые не уничтожаются, даже если они системе уже не нужны и использованы ею в дальнейшем не будут. Это, разумеется, создает существенные сложности при попытке эффективной реализации таких систем по причине неосвобождения занятых актерами ресурсов. Более того, даже автоматическая сборка мусора не всегда способна решить такую проблему, поскольку ее работа напрямую связана с фактом отсутствия в системе ссылок (почтовых адресов) на неиспользуемых ею актеров. Но тот факт, что актер по логике программы больше не будет использован системой, в общем случае не означает, что на него в системе нет ссылок. Иначе говоря, чтобы работала автоматическая сборка мусора, программист при описании той или иной системы должен постоянно заботиться об отсутствии в системе адресов более ненужных актеров, что само по себе является «шагом назад» от идеи высокоуровневого программирования.

Для решения проблемы неосвобождения ресурсов мы расширим модель возможностью наличия подсистем актеров, существующих лишь некоторое время. В качестве реакции на приход некоторого сообщения актер может выполнить полный жизненный цикл внутренний подсистемы актеров. На этапе начальной инициализации этой подсистемы он отправляет в нее исходные данные, пришедшие ему, возможно, с сообщением, на которое он в данный момент реагирует. Результат вычислений, выполненных подсистемой, отправляется актеру-заказчику, почтовый адрес которого также был передан ей на этапе начальной инициализации. В качестве актера-заказчика может выступать текущий актер, инициирующий создание и выполнение подсистемы, или же какой-либо другой актер из внешней по отношению к подсистеме среды, адрес которого известен текущему актеру, к примеру, из пришедшего сообщения. В рамках выполнения подсистемы производятся вычисления на основе переданных входных данных, их результат отправляется актеру-заказчику, после чего работа подсистемы завершается, в связи с чем освобождаются занятые ей ресурсы. Завершение подсистемы, как и ранее, осуществляется тогда, когда в ней закончена обработка всех сообщений.

Может возникнуть вопрос, почему полный жизненный цикл подсистемы должен выполняться в рамках одного действия, а не, к примеру, растянуться на несколько действий некоторого актера, представляющего эту подсистему. При рассмотрении сетей конечных автоматов мы упомянули, что вследствие наличия синхронизации всех автоматов между тактами и, как следствие, наличия состояния сети, определяемого совокупностью состояний ее автоматов, такая сеть также представляет собой конечный автомат. В данном же случае провести полную аналогию не удается. Система взаимодействующих актеров не может в общем случае рассматриваться как актер более высокого уровня из-за отсутствия понятия ее общего глобального состояния и, соответственно, определяемого им поведения.

Более того, внесение возможности прихода в подсистему сообщений извне во время ее работы нарушает согласованность такой модели. Система (в том числе подсистема) актеров завершает работу, когда переходит в режим останова (halt), т.е. когда в системе не

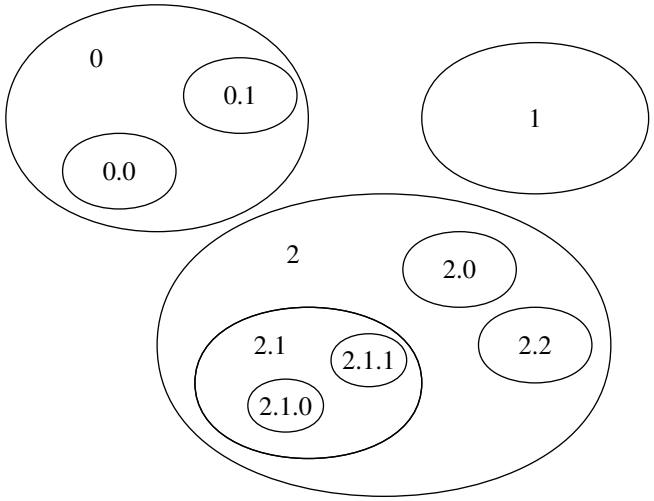



Рис. 5.6. Пример адресации в системе актеров с многоуровневой вложенностью

остается не обработанных сообщений. Однако, если некоторое внешнее сообщение, приход которого в подсистему необходим по логике программы, опоздает с приходом (а рано или поздно именно так и произойдет вследствие недетерминированности времени доставки), подсистема как раз окажется в состоянии, когда все ее актеры обработали сообщения, и новых нет. При этом задача подсистемы оказывается еще не выполненной, но она уже завершает работу.

По этим причинам мы используем подсистемы, существующие лишь в течение одного действия некоторого актера. Отправка сообщений внутренним актерам подсистемы возможна лишь на этапе ее начальной инициализации тем актером, который ее создает и выполняет. Никакие другие варианты отправки сообщений актерам подсистемы по указанным выше причинам невозможны, в связи с чем вводится запрет на пересылку ими своего почтового адреса во внешнюю среду. Если бы при отсутствии такого запрета по переданному во внешнюю среду почтовому адресу внутреннего актера подсистемы было бы отправлено сообщение, оно бы не смогло достичь адресата, поскольку такие актеры получают сообщения извне лишь при инициализации подсистемы. Таким образом, сообщение оставалось бы в системе постоянно, не давая ей завершить выполнение.

Для обеспечения возможности доставки сообщений между актерами разных уровней необходима единая система адресации. В одноуровневой реализации в качестве адреса актера нами было использовано целое число — индекс актера в рамках одноуровневой системы. Теперь же мы будем использовать последовательность чисел, характеризующих индексы соответствующих актеров на каждом уровне (рис. 5.6). Актеры верхнего уровня имеют адреса, состоящие из одного числа. В каждой вложенной подсистеме к адресу добавляется число, характеризующее индекс соответствующего актера в текущей подсистеме.

Обмен сообщениями между актерами может осуществляться только в направлении актеров подсистем верхнего уровня (к примеру, от `2.1.1` к `2.1` на рис. 5.6) или к актеру той же подсистемы (к примеру, от `2.1` к `2.0`). Также допустима отправка соседнему актеру подсистемы верхнего уровня (к примеру, от `2.1.1` к `2.0`). Вовнутрь подсистем (от `2.0` к `2.1.1`) по упомянутым выше причинам отправка невозможна.

Опишем изменения, которые требуется внести в набор классов из приложения Д для обеспечения поддержки вложенных подсистем актеров. Прежде всего, нам потребуется но-

вый тип адреса, содержащий последовательность чисел. Важно отметить, что, поскольку почтовые адреса включаются в отсылаемые сообщения, этот тип должен удовлетворять указанным ранее требованиям к содержимому сообщений, т.е. не должен содержать ссылок и указателей. Как следствие, адрес не может быть представлен каким-либо контейнером наподобие `std::vector` или `std::list`. Поэтому используем для представления адреса массив некоторого фиксированного размера. Таким образом, вместо объявления целочисленного типа `address_type` создадим класс со следующим содержимым:

```
// почтовый адрес актера
class address_type
{
private:
 enum { MAX_DEPTH = 15 };
 int m_size;
 int m_addr[MAX_DEPTH];

 // === интерфейс для scheduler_type ===
 friend class scheduler_type;
 address_type(const address_type &pre, int idx):
  m_size(pre.m_size + 1)
 {
  assert(pre.m_size < MAX_DEPTH);
  for (int i = 0; i < pre.m_size; ++i)
   m_addr[i] = pre.m_addr[i];
  m_addr[pre.m_size] = idx;
 }
 int depth(void) const
 {
  return m_size;
 }
 int index(void) const
 {
  assert(m_size > 0);
  return m_addr[m_size - 1];
 }

public:
 // === интерфейс для клиентского кода ===
 address_type(void):
  m_size(0)
 {}
 friend
 bool operator ==(const address_type &l, const address_type &r)
 {
  bool rc = (l.m_size == r.m_size);
  for (int i = 0; rc && i < l.m_size; ++i)
   rc = (l.m_addr[i] == l.m_addr[i]);
  return rc;
 }
};
```

В этом классе клиентскому коду, помимо неявных конструктора копирования и оператора присваивания, доступны только конструктор по умолчанию и оператор проверки на равенство с другим адресом. Остальные функции предназначены для использования планировщиком. Среди них следующие:

- конструктор `address_type` на основе переданного адреса некоторого актера, выполняющего работу вложенной подсистемы, и индекса внутреннего актера в рамках этой подсистемы;
- `depth` — определение глубины вложенности адреса;
- `index` — получение индекса актера в рамках подсистемы самого нижнего уровня.

Для использования такого типа адреса потребуются следующие изменения планировщика. Прежде всего, теперь класс планировщика будет унаследован от актера. Это укладывается в идею создателей модели о том, что все есть актер [65]. Также подлежат изменению функции пополнения очередей запросов:

```
class scheduler_type: public actor_type
{
// ...
 // === интерфейс для actor_type ===
 friend class actor_type;
 // добавление запроса на создание актера
 address_type new_actor(const behaviour_type &behaviour)
 {
  address_type newaddr;
  #pragma omp critical (creation)
  {
   // формирование нового адреса
   newaddr = address_type(
    self(),
    m_arbiterlist.size() + m_actorqueue.size());
   // добавление запроса
   m_actorqueue.push_back(
    actorqueue_type::value_type(newaddr, behaviour));
  };
  return newaddr;
 }
 // добавление запроса на доставку сообщения
 void new_task(const task_type &task)
 {
  if (task.address.depth() <= self().depth())
   raw_send(task);
  else
  {
   assert(task.address.depth() == self().depth() + 1);
   #pragma omp critical (sending)
   m_taskqueue.push_back(task);
  };
 }
// ...
};
```

Новый адрес создается на базе адреса текущего актера, выступающего в качестве планировщика для подсистемы, а также индекса нового создаваемого актера в рамках этой подсистемы. Помещение сообщения в очередь запросов текущего планировщика осуществляется лишь тогда, когда глубина адреса его назначения совпадает с глубиной адресов актеров этой подсистемы. В противном случае, когда глубина меньше, с помощью функции `raw_send` запрос передается «наверх», в вышестоящие подсистемы.

Наконец, снова меняется функция выполнения жизненного цикла подсистемы:

```
class scheduler_type: public actor_type
{
// ...
 // полный цикл развития системы актеров
 void evolve(const factory_type &factory)
 {
  bool busy;
  do
  {
   // обработка запросов на создание актеров
   while (!m_actorqueue.empty())
   {
    // изымаем из очереди адрес и соответствующий запрос
    address_type newaddr = m_actorqueue.front().first;
    behaviour_type behaviour = m_actorqueue.front().second;
    m_actorqueue.pop_front();
    // создаем арбитра
    assert(size_t(newaddr.index()) == m_arbiterlist.size());
    m_arbiterlist.push_back(
     new arbiter_type(factory, newaddr, behaviour, this));
   };
   // обработка запросов на доставку сообщений
   while (!m_taskqueue.empty())
   {
    // изымаем сообщение
    task_type task = m_taskqueue.front();
    m_taskqueue.pop_front();
    // помещаем его в соответствующий почтовый ящик
    m_arbiterlist[task.address.index()]->deliver(task);
   };
   // цикл работы системы актеров
   busy = false;
   int allnum = m_arbiterlist.size();
   #pragma omp parallel for schedule(guided) reduction(||: busy)
   for (int addr = 0; addr < allnum; ++addr)
   {
    // пополняем признак наличия сообщений в системе
    busy = busy || !m_arbiterlist[addr]->empty();
    // пытаемся получить и обработать сообщение
    task_type task;
    if (m_arbiterlist[addr]->retrieve(task))
     m_arbiterlist[addr]->process(task);
   };
   // выход, если в системе нет сообщений
  } while (busy);

  // уничтожение арбитров
  int allnum = m_arbiterlist.size();
  for (int addr = 0; addr < allnum; ++addr)
   delete m_arbiterlist[addr];
  m_arbiterlist.clear();
 }
};
```

Отличия от кода, приведенного в приложении Д, заключаются лишь в использовании функции `index` при индексации в массиве `m_arbiterlist`.

Можно заметить, что функция `self` при пополнении очередей запросов вызывается в том числе и планировщиком самого верхнего уровня, создаваемого клиентским кодом. Поскольку на его уровне брать собственный адрес неоткуда (нет арбитра, в рамках которого выполняется планировщик), для такого случая модифицируется метод актера `raw_self`:

```
address_type actor_type::raw_self(void) const
{
 return (m_arbiter != NULL) ? m_arbiter->address() : address_type();
}
```

Это все изменения, которые требуется внести в описанный ранее набор классов из приложения Д для поддержки вложенных подсистем. Отметим однако, что при отключенном режиме вложенного параллелизма OpenMP такой код будет распараллеливать лишь выполнение актеров системы самого верхнего уровня. Если же его включить, будет распараллеливаться выполнение и актеров вложенных подсистем, но возникает следующая проблема. Критическая секция с именем `creation`, использованная в функции пополнения очереди запросов на создание, будет взаимоисключать одновременный доступ из различных потоков не к одной очереди, а ко всем очередям запросов на создание из различных планировщиков. Та же ситуация будет и с критической секцией `sending`. Опасности возникновения взаимоблокировки нет, поскольку критические секции не вкладываются одна в другую, однако такая ситуация создает «узкое горло» при работе большого количества подсистем. Поскольку это обязательно скажется на производительности, предложим вариант модификации построенной только что реализации с использованием runtime-функций блокировки вместо критических секций:

```
class scheduler_type: public actor_type
{
// ...
 omp_lock_t m_lockcrea, m_locksend;

 // === интерфейс для actor_type ===
 friend class actor_type;
 // добавление запроса на создание актера
 address_type new_actor(const behaviour_type &behaviour)
 {
  address_type newaddr;
  omp_set_lock(&m_lockcrea);
  // формирование нового адреса
  newaddr = address_type(
   self(),
   m_arbiterlist.size() + m_actorqueue.size());
  // добавление запроса
  m_actorqueue.push_back(
   actorqueue_type::value_type(newaddr, behaviour));
  omp_unset_lock(&m_lockcrea);
  return newaddr;
 }
 // добавление запроса на доставку сообщения
 void new_task(const task_type &task)
 {
  if (task.address.depth() <= self().depth())
   raw_send(task);
```

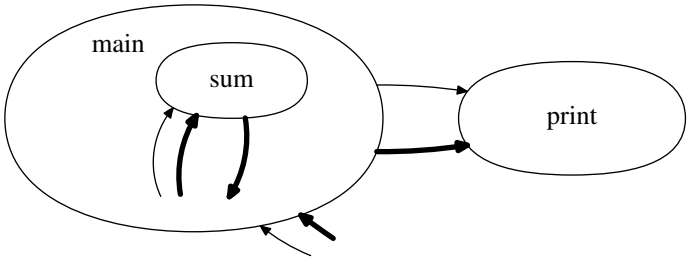



Рис. 5.7. Простая система актеров с вложенностью

```
  else
  {
   assert(task.address.depth() == self().depth() + 1);
   omp_set_lock(&m_locksend);
   m_taskqueue.push_back(task);
   omp_unset_lock(&m_locksend);
  };
 }

public:
 // === интерфейс для клиентского кода ===

 scheduler_type(void)
 {
  // псевдо-актер привязан только к планировщику
  // арбитра у него нет, как и почтового адреса
  m_origin.bind(this, NULL);

  omp_init_lock(&m_lockcrea);
  omp_init_lock(&m_locksend);
 }
 ~scheduler_type(void)
 {
  omp_destroy_lock(&m_locksend);
  omp_destroy_lock(&m_lockcrea);
 }
// ...
};
```

Объект планировщика содержит две переменных блокировки, по одной на каждую очередь запросов. Для выполнения их инициализации и уничтожения пополняется конструктор и добавляется деструктор. Обе эти переменные используются вместо задействованных ранее критических секций для синхронизации доступа к очередям запросов в соответствующих функциях `new_actor` и `new_task`.

В качестве примера клиентского кода, построенного с использованием внесенной поддержки вложенных подсистем, рассмотрим усложненный вариант системы актеров, изображенной на рис. 5.1. На этот раз суммирование будем выполнять в рамках вложенной подсистемы, при этом результат суммирования должен передаваться в систему верхнего уровня (рис. 5.7). Полученный от вложенной подсистемы результат вычислений передается создаваемому в системе верхнего уровня актеру `print`.

Такая система реализуется следующим кодом:

```
struct message_type
{
 int arg1, arg2;
 address_type cust;
};

class summator_type: public actor_type
{
public:
 summator_type(const empty_type &init)
 {
  add_action<summator_type, message_type>();
 }
 void action(const message_type &msg)
 {
  send(msg.cust, msg.arg1 + msg.arg2);
 }
};

class main_type: public scheduler_type
{
public:
 struct start_type {};

 main_type(const empty_type &init)
 {
  add_action<main_type, start_type>();
  add_action<main_type, int>();
 }
 void action(const start_type &start)
 {
  // фабрика текущего планировщика
  factory_type factory;
  factory.add_definition<summator_type, actor_type::empty_type>();
  // инициализация вложенной системы актеров
  message_type msg = { 2, 2, self() };
  system().send(
   system().create<summator_type>(),
   msg);
  // жизненный цикл вложенной системы актеров
  evolve(factory);
 }
 void action(const int &result)
 {
  send(create<print_type>(), result);
 }
};

// ...

// регистрация актеров в фабрике
factory_type factory;
factory.add_definition<main_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();
```

```
// планировщик
scheduler_type sched;
// инициализация системы актеров
sched.system().send(
 sched.system().create<main_type>(),
 main_type::start_type());
// жизненный цикл системы актеров
sched.evolve(factory);
```

В приведенном примере вложенная подсистема содержит лишь одного актера, вследствие чего пример не демонстрирует никакой экономии. В общем же случае количество создаваемых в процессе вычисления промежуточного результата актеров может быть довольно велико. Тогда использование вложенных подсистем, завершающих работу созданных в процессе ее выполнения актеров, может существенно повысить эффективность использования вычислительных ресурсов.

## 5.3. Примеры решения некоторых задач

Далее будет приведено несколько примеров решения вычислительных задач на основе модели актеров. Описание некоторых из них взято из оригинальных источников [48, 51, 66], другие просто подходят для решения с использованием естественного параллелизма модели. Практически все приведенные примеры носят иллюстративный характер, т.е. предназначены именно показать, как можно решить поставленную задачу с использованием актеров. В то же время становится ясно, что вычислять, к примеру, факториал приведенным способам в практических задачах несколько обременительно, и проще оказывается использовать другие подходы. Однако приведенные примеры призваны показать именно возможности построения систем актеров и их взаимодействия, выбор же размеров выполняемых ими в рамках каждого действия задач остается на усмотрение разработчика конкретной системы.

### 5.3.1. Вычисление факториала

В качестве первого базового примера использования модели актеров обычно приводится описание рекурсивной процедуры вычисления факториала. При этом описывается механизм вычислений, эквивалентный выполняемому следующей функцией:

```
int factorial(int n)
{
 assert(n >= 0);
 return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

При выполнении такой функции осуществляется неявное создание последовательности значений $n, n-1, \ldots, 1$, хранимых в стеке, после чего происходит их перемножение, начиная с конца. Такой же механизм предложен и для рекурсивного вычисления факториала, реализованного с помощью актеров [48]:

```
def customer (mul, cust)
[res]
    send [mul * res] to cust
end def

def factorial ()
```

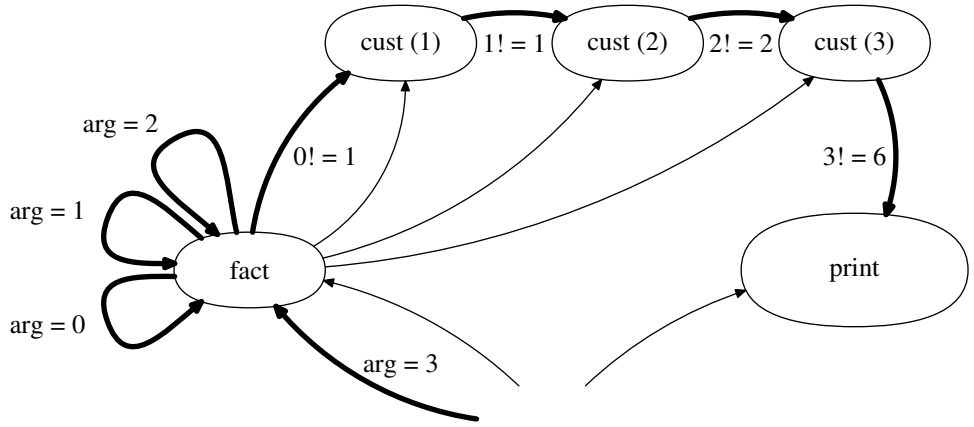



Рис. 5.8. Схема взаимодействия актеров во время рекурсивного вычисления факториала

```
[arg, cust]
    if arg = 0
    then
        send [1] to cust
    else
        let c = create customer (arg, cust)
        in { send [arg - 1, c] to self }
    fi
end def

let f = create factorial (),
    p = create print ()
in { send [num, p] to f }
```

В задачи актера с поведением `customer` входит ожидание сообщения с числом и, когда оно получено, перемножение его с другим числом, полученным при инициализации, с последующей отправкой результата актеру, адрес которого также получен при инициализации.

Актер с поведением `factorial` получает сообщение с неотрицательным аргументом, от которого следует вычислить значение факториала, и адресом актера, которому нужно отправить результат. В рамках реакции на такое сообщение актер проверяет значение аргумента на равенство нулю и, в случае успеха, отправляет заказчику единицу. В противном случае он создает нового актера `customer` с параметрами, в которых передает на хранение текущее значение аргумента, а также адрес текущего актера-заказчика. После этого актер выполняет отправку себе другого сообщения, в котором передает уменьшенное на единицу значение аргумента для вычисления факториала, а также адрес нового только что созданного актера-заказчика.

В рамках начальной инициализации системы создается актер `factorial` и актер `print`, после чего первому отправляется сообщение со значением аргумента и адресом второго. В процессе выполнения процедуры вычисления осуществляется создание цепочки актеров `customer`, хранящих последовательность множителей, после чего выполняется их перемножение и отправка результата актеру `print` (рис. 5.8).

Рекурсивное вычисление факториала, как справедливо замечено в [48], является по природе своей последовательным. Поэтому однократное вычисление с помощью приведенного алгоритма не будет выполняться в параллельной среде быстрее, чем в последовательной.

Однако полезным в параллельной среде свойством описанного актера `factorial` является возможность обрабатывать множество запросов на вычисление факториала одновременно.

Система актеров, выполняющая описанные вычисления на основе построенного нами ранее набора классов, реализуется следующим кодом:

```
struct calcrequest_type
{
 int argument;
 address_type customer;
};

class customer_type: public actor_type
{
private:
 int m_multiplier;
 address_type m_customer;

public:
 struct init_type
 {
  int multiplier;
  address_type customer;
 };

 customer_type(const init_type &init):
  m_multiplier(init.multiplier),
  m_customer(init.customer)
 {
  add_action<customer_type, int>();
 }
 void action(const int &msg)
 {
  send(m_customer, m_multiplier * msg);
 }
};

class factorial_type: public actor_type
{
public:
 factorial_type(const empty_type &init)
 {
  add_action<factorial_type, calcrequest_type>();
 }
 void action(const calcrequest_type &msg)
 {
  assert(msg.argument >= 0);
  if (msg.argument == 0)
   send(msg.customer, 1);
  else
  {
   customer_type::init_type init = {
    msg.argument, msg.customer
   };
   calcrequest_type req = {
    msg.argument - 1,
    create<customer_type>(init)
```

```
    };
    send(self(), req);
   };
  }
};

// ...

const int num = /*...*/;

factory_type factory;
factory.add_definition<customer_type, customer_type::init_type>();
factory.add_definition<factorial_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();

scheduler_type sched;
calcrequest_type req = {
 num,
 sched.system().create<print_type>()
};
sched.system().send(
 sched.system().create<factorial_type>(),
 req);
sched.evolve(factory);
```

Приведенный код содержит два класса, соответствующих определениям поведений актеров `customer` и `factorial` соответственно, а также фрагмент, осуществляющий начальную инициализацию системы актеров и ее выполнение в соответствии с описанной выше программой на языке SAL.

Помимо рекурсивного, нередко рассматривается также итеративный вариант вычисления факториала [66], эквивалентный реализуемому следующей функцией:

```
int factorial(int n)
{
 int p = 1;
 assert(n >= 0);
 while (n)
  p *= n--;
 return p;
}
```

В отличие от предыдущего случая, здесь не осуществляется явного или неявного построения цепочки хранимых множителей, перемножение осуществляется «на лету». Выполняемое этой функцией вычисление реализуется с помощью актеров следующей программой:

```
def loop (cust)
[arg, prod]
    if arg = 0
    then
        send [prod] to cust
    else
        send [arg - 1, arg * prod] to self
    fi
end def

def factorial ()
```

```
[arg, cust]
    let l = create loop (cust)
    in { send [arg, 1] to l }
end def

let f = create factorial (),
    p = create print ()
in { send [num, p] to f }
```

Актер с поведением `loop` ответственен за выполнение цикла. Через этот цикл в параметрах сообщений передаются текущие значения аргумента и накопленного произведения. Если аргумент равен нулю, произведение отсылается актеру-заказчику. В противном случае актер посылает себе сообщение для следующей итерации, в котором передает уменьшенный на единицу аргумент и перемноженное с аргументом значение накопленного произведения. В задачу актера `factorial` входит лишь создание актера `loop` с адресом актера-заказчика в качестве параметра и отправка ему начального сообщения. Таким образом, в итеративной реализации при вычислении факториала создается всего два актера.

Следующий код представляет собой реализацию приведенной программы итеративного вычисления факториала на основе построенного нами набора классов:

```
struct calcrequest_type
{
 int argument;
 address_type customer;
};

class loop_type: public actor_type
{
public:
 struct next_type
 {
  int argument;
  int product;
  address_type customer;
 };

 loop_type(const empty_type &init)
 {
  add_action<loop_type, next_type>();
 }
 void action(const next_type &msg)
 {
  assert(msg.argument >= 0);
  if (msg.argument == 0)
   send(msg.customer, msg.product);
  else
  {
   next_type next = {
    msg.argument - 1,
    msg.argument * msg.product,
    msg.customer
   };
   send(self(), next);
  };
 }
```

```
};

class factorial_type: public actor_type
{
public:
 factorial_type(const empty_type &init)
 {
  add_action<factorial_type, calcrequest_type>();
 }
 void action(const calcrequest_type &msg)
 {
  loop_type::next_type next = {
   msg.argument, 1, msg.customer
  };
  send(create<loop_type>(), next);
 }
};

// ...

const int num = /*...*/;

factory_type factory;
factory.add_definition<loop_type, actor_type::empty_type>();
factory.add_definition<factorial_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();

scheduler_type sched;
calcrequest_type req = {
 num,
 sched.system().create<print_type>()
};
sched.system().send(
 sched.system().create<factorial_type>(),
 req);
sched.evolve(factory);
```

Как и в рекурсивном случае, итеративный актер `factorial` может обрабатывать одновременно несколько запросов на вычисление факториала. При этом для обработки каждого запроса им каждый раз будет создаваться новый актер `loop`.

### 5.3.2. Числа Фибоначчи

Другим примером, близким к вычислению факториала, является вычисление чисел Фибоначчи. Как известно, последовательность этих чисел определена следующим образом:

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, \quad F_1 = 1.$$

Самый простой рекурсивный механизм вычисления очередного числа Фибоначчи с некоторым неотрицательным номером выражается следующей функцией:

```
int fibonacci(int n)
{
 assert(n >= 0);
```

```
 return n <= 1 ? n : (fibonacci(n - 1) + fibonacci(n - 2));
}
```

В процессе вычисления функция рекурсивно вызывает себя дважды. Это порождает известную проблему, а именно экспоненциальный рост времени вычисления в зависимости от номера требуемого числа.

Система актеров, выполняющая таким способом вычисление числа Фибоначчи с некоторым номером, описывается на языке SAL следующей программой:

```
def customer (val, cust)
[res]
    if val = NIL
    then
        become customer (res, cust)
    else
        send [res + val] to cust
        become sink ()
    fi
end def

def fibonacci ()
[arg, cust]
    if arg <= 1
    then
        send [arg] to cust
    else
        let c = create customer (NIL, cust),
            f = create fibonacci ()
        in {
            send [arg - 1, c] to self
            send [arg - 2, c] to f
        }
    fi
end def

let f = create fibonacci (),
    p = create print ()
in { send [num, p] to f }
```

Актер с поведением `customer` ожидает прихода двух чисел, которые ему предстоит просуммировать. В момент прихода первого из них он сохраняет его с помощью операции `become`, а именно создает себе на замену актера с тем же поведением, но другими параметрами инициализации, в которых передает пришедшее число. Когда приходит второе число, он отправляет сумму обоих чисел актеру-заказчику, после чего определяет свое дальнейшее поведение как `sink`, т.е. как актер, игнорирующий все принятые сообщения. Эта операция выполняется для того, чтобы актер-заказчик получил лишь одно сообщение от текущего актера, даже если последнему по ошибке придет более двух сообщений. Если же достоверно известно, что больше сообщений с числами не будет, операцию `become sink ()` можно опустить.

Если номер требуемого числа Фибоначчи больше единицы, актер с поведением `fibonacci` выполняет два рекурсивных вызова. Создается промежуточный актер `customer`, адрес которого передается в оба рекурсивных вызова с тем, чтобы результат выполнения обоих был им просуммирован. Рекурсивный вызов может быть осуществлен как путем отправки сообщения себе, так и путем создания нового актера с аналогичным поведением и

отправки сообщения ему. В нашем случае использованы оба варианта.

Ниже представлен код, реализующий такую систему актеров:

```
struct calcrequest_type
{
 int argument;
 address_type customer;
};

class customer_type: public actor_type
{
private:
 address_type m_customer;
 bool m_hasvalue;
 int m_value;

public:
 struct init_type
 {
  address_type customer;
  bool hasvalue;
  int value;
 };

 customer_type(const init_type &init):
  m_customer(init.customer),
  m_hasvalue(init.hasvalue),
  m_value(init.value)
 {
  add_action<customer_type, int>();
 }
 void action(const int &msg)
 {
  if (!m_hasvalue)
  {
   init_type init = { m_customer, true, msg };
   become<customer_type>(init);
  }
  else
  {
   send(m_customer, msg + m_value);
   become<sink_type>();
  };
 }
};

class fibonacci_type: public actor_type
{
public:
 fibonacci_type(const empty_type &init)
 {
  add_action<fibonacci_type, calcrequest_type>();
 }
 void action(const calcrequest_type &msg)
 {
  assert(msg.argument >= 0);
```

```
  if (msg.argument <= 1)
   send(msg.customer, msg.argument);
  else
  {
   customer_type::init_type init = { msg.customer, false, 0 };
   address_type customer = create<customer_type>(init);

   calcrequest_type req1 = { msg.argument - 1, customer };
   send(self(), req1);
   calcrequest_type req2 = { msg.argument - 2, customer };
   send(create<fibonacci_type>(), req2);
  };
 }
};

// ...

const int num = /*...*/;

factory_type factory;
factory.add_definition<customer_type, customer_type::init_type>();
factory.add_definition<fibonacci_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();
factory.add_definition<sink_type, actor_type::empty_type>();

scheduler_type sched;
calcrequest_type req = {
 num,
 sched.system().create<print_type>()
};
sched.system().send(
 sched.system().create<fibonacci_type>(),
 req);
sched.evolve(factory);
```

Может возникнуть вопрос, почему в приведенном коде класса `customer_type` для задания новых значений внутренним переменным объекта используется вызов `become` с вытекающим из него пересозданием объекта. Вместо этого можно было просто присвоить переменным новые значения, и результат был бы тем же, и выполнялся бы такой код быстрее. Да, так и было бы. Но в модели актеров любое изменение состояния актера выполняется через операцию `become`, а в наши задачи входит именно иллюстрация использования классической модели актеров. Упростить же ее, спроецировав на наличие возможности многократного присваивания, каждый всегда сможет сам.

Выше упоминалась проблема, возникающая при вычислении чисел Фибоначчи описанным путем, а именно экспоненциальный рост времени, необходимого для вычислений, возникающий вследствие двойного рекурсивного вызова. В качестве альтернативного пути зачастую используется функция, возвращающая пару чисел, а именно числа Фибоначчи с заданным и предыдущим номерами:

```
std::pair<int, int> fibpair(int n)
{
 assert(n > 0);
 std::pair<int, int> fib(1, 0);
 if (n > 1)
 {
```

```
  std::pair<int, int> pre = fibpair(n - 1);
  fib = std::make_pair(pre.first + pre.second, pre.first);
 };
 return fib;
}

int fibonacci(int n)
{
 assert(n >= 0);
 return n == 0 ? 0 : fibpair(n).first;
}
```

Рекурсивный вызов в такой реализации всего один, в связи с чем время вычисления зависит от номера числа Фибоначчи линейно. Такой механизм реализуется с помощью актеров следующей программой:

```
def customer (cust)
[res, pre]
    send [res + pre, res] to cust
end def

def takeone (cust)
[res, pre]
    send [res] to cust
end def

def fibpair ()
[arg, cust]
    if arg = 1
    then
        send [1, 0] to cust
    else
        let c = create customer (cust)
        in { send [arg - 1, c] to self }
    fi
end def

def fibonacci ()
[arg, cust]
    if arg = 0
    then
        send [0] to cust
    else
        let t = create takeone (cust)
        in { send [arg, t] to create fibpair () }
    fi
end def

let f = create fibonacci (),
    p = create print ()
in { send [num, p] to f }
```

Актер `customer` получает пару соседних чисел Фибоначчи, формирует из них следующую пару и отправляет ее дальше по цепочке.

Поскольку `customer` отправляет пару чисел, в то время как актеру-заказчику требуется лишь одно, введен еще один актер с поведением `takeone`. Он ожидает прихода пары чисел и

отправляет одно из них по адресу, полученному им при инициализации. Этот актер призван быть последним элементом в цепочке актеров `customer`, связывающим ее непосредственно с актером-заказчиком конечного результата вычислений.

Сообщение, принимаемое актером с поведением `fibpair`, представляет собой заказ на вычисление и содержит значение аргумента и адрес актера-заказчика промежуточного результата вычислений. Если аргумент равен единице, актеру-заказчику направляется соответствующая пара чисел Фибоначчи, в противном случае осуществляется рекурсивный вызов.

Наконец, актер `fibonacci` в ответ на приход запроса на вычисление создает актера `takeone`, передавая ему адрес актера-заказчика, а также актера `fibpair`, после чего отправляет последнему задание на вычисление пары чисел с указанием в качестве получателя результата адреса созданного только что актера `takeone`.

Описанная система реализуется следующим кодом:

```
struct calcrequest_type
{
 int argument;
 address_type customer;
};

class customer_type: public actor_type
{
private:
 address_type m_customer;

public:
 struct init_type
 {
  address_type customer;
 };
 struct result_type
 {
  int result;
  int previous;
 };

 customer_type(const init_type &init):
  m_customer(init.customer)
 {
  add_action<customer_type, result_type>();
 }
 void action(const result_type &msg)
 {
  result_type res = {
   msg.result + msg.previous,
   msg.result
  };
  send(m_customer, res);
 }
};

class fibpair_type: public actor_type
{
private:
```

```
 address_type m_customer;

public:
 struct init_type
 {
  address_type customer;
 };

 fibpair_type(const init_type &init):
  m_customer(init.customer)
 {
  add_action<fibpair_type, calcrequest_type>();
  add_action<fibpair_type, customer_type::result_type>();
 }
 void action(const calcrequest_type &msg)
 {
  assert(msg.argument > 0);
  if (msg.argument == 1)
  {
   customer_type::result_type res = { 1, 0 };
   send(msg.customer, res);
  }
  else
  {
   customer_type::init_type init = { msg.customer };
   address_type customer = create<customer_type>(init);

   calcrequest_type res = { msg.argument - 1, customer };
   send(self(), res);
  };
 }
 void action(const customer_type::result_type &msg)
 {
  send(m_customer, msg.result);
 }
};

class fibonacci_type: public actor_type
{
public:
 fibonacci_type(const empty_type &init)
 {
  add_action<fibonacci_type, calcrequest_type>();
 }
 void action(const calcrequest_type &msg)
 {
  assert(msg.argument >= 0);
  if (msg.argument == 0)
   send(msg.customer, 0);
  else
  {
   fibpair_type::init_type init = { msg.customer };
   address_type fibpair = create<fibpair_type>(init);

   calcrequest_type req = { msg.argument, fibpair };
```

```
    send(fibpair, req);
   };
  }
};

// ...

const int num = /*...*/;

factory_type factory;
factory.add_definition<customer_type, customer_type::init_type>();
factory.add_definition<fibpair_type, fibpair_type::init_type>();
factory.add_definition<fibonacci_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();

scheduler_type sched;
calcrequest_type req = {
 num,
 sched.system().create<print_type>()
};
sched.system().send(
 sched.system().create<fibonacci_type>(),
 req);
sched.evolve(factory);
```

Вследствие отсутствия двойной рекурсии такая система актеров растет гораздо медленнее, чем в предыдущем варианте, и позволяет вычислять числа Фибоначчи с номерами гораздо большей величины. Однако этот алгоритм, как уже говорилось выше, имеет линейную сложность, в то время как возможно построение еще более быстрой реализации с логарифмической сложностью. На основе рекуррентного выражения пары соседних чисел Фибоначчи $(F_n, F_{n-1})$ через предыдущую пару $(F_{n-1}, F_{n-2})$ получаем:

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n-1} + F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = A \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = A^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}.$$

Видно, что любая пара чисел Фибоначчи представляется в виде произведения начальной пары $(F_1, F_0)$ и известной матрицы $A$ размером $2 \times 2$, возведенной в некоторую степень. После возведения $A$ в заданную степень путем умножения на вектор $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ получаем первый столбец матрицы результата, из которого нас интересует лишь число в первой строке. Таким образом, для вычисления числа $F_n$ нам следует возвести матрицу $A$ в степень $n-1$ и взять из полученного результата левое верхнее число.

В свою очередь, для операции возведения в степень известен простой и быстрый алгоритм, основанный на двоичном представлении показателя степени. Степень $A^n$ с четным показателем может быть представлена в виде квадрата степени с показателем, в два раза меньшим $n$. Степень с нечетным показателем может быть выражена через четную с умножением на основание $A$:

$$A^n = \begin{cases} (A^{\frac{n}{2}})^2, & n \bmod 2 = 0; \\ (A^{\frac{n-1}{2}})^2 A, & n \bmod 2 = 1. \end{cases}$$

Основанное на таких соотношениях вычисление степени требует количества операций, пропорциональное логарифму ее показателя $n$. Ниже приведен пример реализации на его основе вычисления числа Фибоначчи с использованием шаблона класса из приложения А:

```
matrix_type<int> mtxpow(const matrix_type<int> &mtx, int n)
{
 assert(mtx.vsize() == mtx.hsize());
 assert(n >= 0);
 if (n == 0)
 {
  matrix_type<int> e(mtx.vsize(), mtx.hsize());
  for (int i = 0; i < e.vsize(); ++i)
   for (int j = 0; j < e.hsize(); ++j)
    e(i, j) = (i == j) ? 1 : 0;
  return e;
 }
 else
 {
  matrix_type<int> r = mtxpow(mtx, n >> 1);
  return (n & 1) ? (r * r * mtx) : (r * r);
 };
}

int fibonacci(int n)
{
 assert(n >= 0);
 matrix_type<int> a(2, 2);
 for (int i = 0; i < a.vsize(); ++i)
  for (int j = 0; j < a.hsize(); ++j)
   a(i, j) = (i && j) ? 0 : 1;
 return n == 0 ? 0 : mtxpow(a, n - 1)(0, 0);
}
```

Функция `mtxpow` осуществляет возведение переданной квадратной матрицы в заданную неотрицательную степень. В случае, если показатель степени равен нулю, она возвращает единичную матрицу заданных размеров. В противном случае рекурсивно производится вычисление для степени, равной половине от ближайшей снизу четной степени, после чего результат возводится в квадрат и, если текущий показатель был нечетным, домножается на основание. В задачи функции `fibonacci` входит лишь формирование матрицы $A$, возведение ее в степень $n-1$ и вычленение из результата первого элемента первой строки.

Система актеров, реализующая такой алгоритм, по структуре взаимодействия несущественно отличается от описанной выше системы с линейным ростом:

```
def customer (mul, cust)
[res]
    send [res * res * mul] to cust
end def

def takeone (cust)
[res]
    send [res[1 1]] to cust
end def

def mtxpow ()
[mtx, pow, cust]
    let e = [ [ 1 0 ] [ 0 1 ] ]
    in {
        if pow = 0
```

```
        then
            send [e] to cust
        else
            if pow is odd
            then
                let c = create customer (mtx, cust)
                in { send [mtx, (arg - 1) / 2, c] to self }
            else
                let c = create customer (e, cust)
                in { send [mtx, arg / 2, c] to self }
            fi
        fi
    }
end def

def fibonacci ()
[arg, cust]
    if arg = 0
    then
        send [0] to cust
    else
        let t = create takeone (cust),
            m = [ [ 1 1 ] [ 1 0 ] ]
        in { send [m, arg - 1, t] to create mtxpow () }
    fi
end def

let f = create fibonacci (),
    p = create print ()
in { send [num, p] to f }
```

Актер с поведением `mtxpow` осуществляет возведение в степень переданной квадратной матрицы. В случае нулевой степени он возвращает единичную матрицу, иначе осуществляет рекурсивный вызов, в рамках которого запрашивает вычисление в два раза меньшей степени. Для выполнения действий после рекурсивного вызова создается актер `customer`, которому в параметрах инициализации передается исходная возводимая в степень матрица, если текущая степень нечетная, и единичная матрица в противном случае. Актер `customer` получает некоторую матрицу, возводит ее в квадрат, умножает на матрицу-множитель, полученную через параметры инициализации, и отправляет результат далее по цепочке. Актер `fibonacci` запрашивает возведение заданной матрицы в указанную степень и извлечение из результата первого элемента первой строки средствами актера `takeone`.

Несмотря на то, что структура взаимодействия системы актеров схожа с предыдущей, растет она пропорционально логарифму номера запрашиваемого числа Фибоначчи, что обеспечивает гораздо более высокую эффективность на больших номерах.

### 5.3.3. Задача чтения-записи

При рассмотрении сетей Петри нами была приведена в пример задача чтения-записи. Покажем теперь, как эта же задача может быть решена с помощью актеров [51].

Предположим, у нас есть некий ресурс, допускающий выполнение различных операций получения данных (чтение) и их модификации (запись). Под различными операциями имеется в виду, к примеру, чтение или запись различных свойств или частей сложного ресурса. Такой ресурс может быть представлен неким актером, принимающим два соот-

ветствующих типа сообщений (запросы на чтение и запись). В запросе передается тег (tag), указывающий, какую часть данных (или же какое свойство ресурса) требуется записать или прочитать.

Оба типа операций могут быть сложными и выполняться, к примеру, с использованием итерации или рекурсии. Это приводит к тому, что выполнение каждой такой операции в рамках ресурса перестает быть атомарным и может нарушить целостность ресурса, если из множества одновременно обрабатываемых запросов хотя бы один является запросом на запись.

К примеру, рассмотрим ресурс со следующим поведением:

```
def resource (...)
[rrequest, cust, tag]
    ...
    send [rreply, tag, data] to cust
[wrequest, cust, tag, data]
    ...
    send [wreply, tag] to cust
end def

let r = create resource (...)
in { ... }
```

Здесь выполнение отправки ответов в рамках реакции именно на соответствующий запрос указано весьма условно. Вследствие возможного использования итерации или рекурсии отправка ответов в общем случае может происходить в качестве реакции на другие сообщения, и даже может выполняться другими актерами.

Для обеспечения корректного доступа к такому ресурсу предлагается использование так называемого сериализатора [51]. Им является некий промежуточный актер, который выступает от имени ресурса и предоставляет актерам-заказчикам тот же интерфейс, что и сам ресурс. Все запросы следуют сначала на сериализатор, который далее пересылает их на ресурс в корректной последовательности (рис. 5.9). Ответы от ресурса также следуют через сериализатор, вследствие чего он может делать выводы о завершении выполнения тех или иных операций (на рис. 5.9 не обозначены).

Одним из предложенных в [51] подходов является содержание сериализатором единой внутренней очереди, в которую помещаются все приходящие на сериализатор запросы. Такой подход решает в том числе проблему голодания, когда во время постоянной перекрывающейся обработки запросов на чтение не выполняется обработка запросов на запись. Проблема решается за счет того, что одновременно на ресурс подаются лишь те запросы на чтение, между которыми в очереди нет запросов на запись, т.е. параллельно обрабатываются ресурсом лишь рядом стоящие запросы на чтение (рис. 5.10).

Внутренняя очередь запросов позволяет выполнять следующие операции:

- добавление нового элемента в хвост очереди (`enqueue`);
- обращение к головному элементу непустой очереди без его изъятия (`head`);
- изъятие из непустой очереди головного элемента (`dequeue`);
- проверка очереди на пустоту (`empty`).

Выполнение этих и других операций над внутренними объектами для наглядности будет нами синтаксически представлено как непосредственный вызов их методов, хотя, как

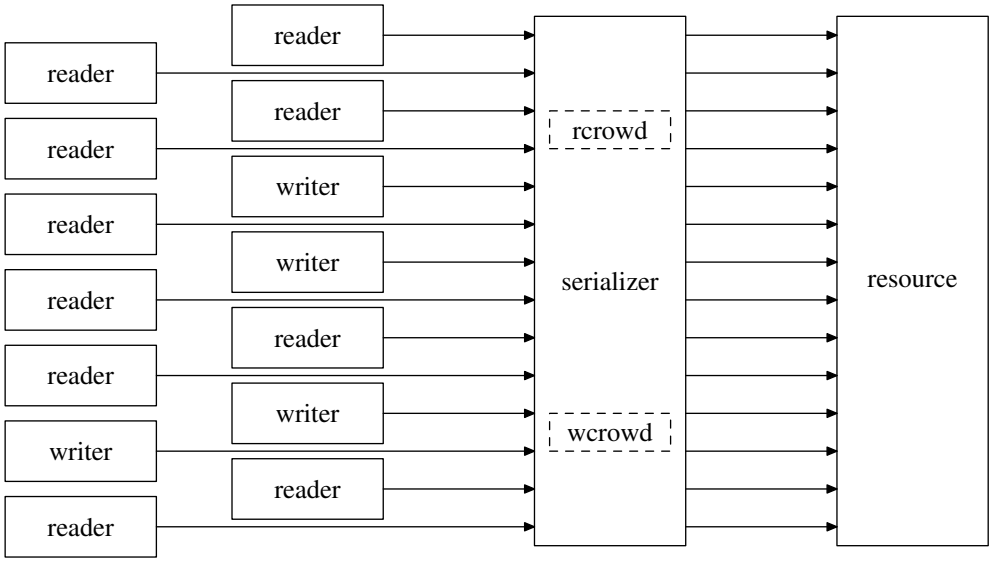



Рис. 5.9. Схема движения запросов на критический ресурс

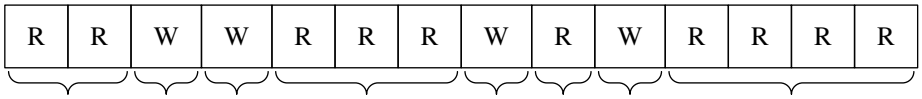



Рис. 5.10. Группировка запросов на чтение и запись из очереди

говорилось ранее, в соответствии с моделью актеров модификация внутренних объектов на низком уровне реализуется с использованием операции `become`.

Помимо очереди, сериализатор содержит две группы (crowd) адресов читателей и писателей (т.е. актеров, отправивших на ресурс соответствующий запрос), обработка запросов от которых выполняется ресурсом в данный момент. По условиям задачи группа читателей может содержать произвольное количество элементов, при этом группа писателей в этот момент должна быть пуста. В свою очередь, группа писателей может состоять максимум из одного элемента, при этом должна быть пуста группа читателей.

При отправке сериализатором на ресурс очередного запроса на чтение или запись пополняется соответствующая группа. В момент переправки актеру-заказчику ответа от ресурса соответствующий адрес из группы изымается.

Изъятие запросов из очереди и переправка на ресурс осуществляется при выполнении необходимых условий пустоты групп писателей и/или читателей. Если головным элементом очереди является запрос на чтение, переправка его на ресурс осуществляется лишь в случае, если пуста группа писателей. В противном случае, если головной элемент — запрос на запись, для переправки его на ресурс должны быть пусты обе группы.

Структура данных, представляющая группу, предоставляет возможность выполнения следующих операций:

- включение нового элемента, характеризующего адрес актера-заказчика и тег запрошенной им операции (`include`);
- поиск и исключение элемента по тегу выполненной операции, возвращается адрес актера-заказчика (`exclude`);

- проверка группы актеров-заказчиков на пустоту (`empty`).

Запросы на каждый тип операции различаются по тегу, при этом ответы на запросы от разных актеров с одинаковым тегом по смыслу и содержимому не отличаются. Это дает нам возможность извлекать адрес актера-заказчика из соответствующей группы на основе тега пришедшего ответа.

Приведем код описания поведения актера-сериализатора:

```
def serializer (res)
[rrequest, cust, tag]
    queue.enqueue [rrequest, cust, tag, NIL]
    send [next] to self
[wrequest, cust, tag, data]
    queue.enqueue [wrequest, cust, tag, data]
    send [next] to self
[next]
    if not queue.empty
    then
        let [reqtype, cust, tag, data] = queue.head
        in {
            case reqtype of
                rrequest:
                    if wcrowd.empty
                    then
                        send [rrequest, self, tag] to res
                        rcrowd.include [tag, cust]
                        queue.dequeue
                        send [next] to self
                    fi
                wrequest:
                    if wcrowd.empty and rcrowd.empty
                    then
                        send [wrequest, self, tag, data] to res
                        wcrowd.include [tag, cust]
                        queue.dequeue
                        send [next] to self
                    fi
            end case
        }
    fi
[rreply, tag, data]
    let cust = rcrowd.exclude [tag]
    in { send [rreply, tag, data] to cust }
    send [next] to self
[wreply, tag]
    let cust = wcrowd.exclude [tag]
    in { send [wreply, tag] to cust }
    send [next] to self
end def

let u = create resource ()
in {
    let r = create serializer (u)
    in { ... }
}
```

Актер `serializer` содержит три внутренних объекта — очередь запросов `queue`, группу читателей `rcrowd` и группу писателей `wcrowd`. Описанием поведения предусмотрена возможность принятия пяти образцов сообщений, а именно запросов двух типов, ответов на эти запросы, а также переход к следующему элементу внутренней очереди запросов. Последний упомянутый образец сообщения введен по той причине, что изъятие запросов из очереди и отправка их на ресурс — процесс итеративный. В момент прихода запросов на чтение или запись они помещаются в конец внутренней очереди `queue`, после чего актером инициируется процесс прокачки очереди путем отправки себе сообщения `next`.

При приходе сообщения `next` очередь `queue` проверяется на пустоту. Если она не пуста, проверяется тип запроса в голове очереди. Если головной элемент очереди является запросом на чтение и группа писателей пуста, запрос переправляется на ресурс и удаляется из очереди, в группу читателей добавляется новый элемент, после чего осуществляется переход к следующему элементу очереди. В случае, когда в голове очереди содержится запрос на запись, проверяется пустота обеих групп заказчиков.

Когда от ресурса приходит сообщение о завершении выполнения некоторой операции, по ее тегу из соответствующей группы извлекается адрес актера-заказчика, после чего ему переправляется принятое сообщение. Далее инициируется процесс прокачки очереди, поскольку могли измениться условия пустоты групп актеров-заказчиков.

Актер с описанным поведением может быть реализован следующим кодом:

```
struct rrequest_type
{
 address_type customer;
 tag_type tag;
};
struct wrequest_type
{
 address_type customer;
 tag_type tag;
 data_type data;
};

struct rreply_type
{
 tag_type tag;
 data_type data;
};
struct wreply_type
{
 tag_type tag;
};

class resource_type: public actor_type
{
// ...
public:
 void action(const rrequest_type &msg)
 {
  // ...
 }
 void action(const wrequest_type &msg)
 {
  // ...
 }
```

```
};

class serializer_type: public actor_type
{
private:
 // почтовый адрес контролируемого ресурса
 address_type m_resource;


 // группы читателей и писателей
 typedef multimap<tag_type, address_type> crowd_type;
 crowd_type m_rcrowd, m_wcrowd;

 // элемент очереди запросов на обслуживание
 struct request_type
 {
  enum { READ, WRITE } type;
  address_type customer;
  tag_type tag;
  data_type data;
 };
 // очередь запросов на обслуживание
 typedef list<request_type> queue_type;
 queue_type m_queue;

public:
 struct init_type
 {
  address_type resource;
 };

 struct next_type {};

 serializer_type(const init_type &init):
  m_resource(init.resource)
 {
  add_action<serializer_type, rrequest_type>();
  add_action<serializer_type, wrequest_type>();
  add_action<serializer_type, next_type>();
  add_action<serializer_type, rreply_type>();
  add_action<serializer_type, wreply_type>();
 }

 // прием и помещение в очередь запросов на чтение и запись
 void action(const rrequest_type &msg)
 {
  request_type r = {
   request_type::READ,
   msg.customer,
   msg.tag
  };
  m_queue.push_back(r);

  send(self(), next_type());
 }
```

```
void action(const wrequest_type &msg)
{
 request_type r = {
  request_type::WRITE,
  msg.customer,
  msg.tag,
  msg.data
 };
 m_queue.push_back(r);

 send(self(), next_type());
}

// попытка отправки очередного запроса ресурсу
void action(const next_type &msg)
{
 // в очереди должен быть хотя бы один элемент
 if (!m_queue.empty())
 {
  request_type &r = m_queue.front();
  switch (r.type)
  {
  case request_type::READ:
   // если запрос на чтение, группа писателей должна быть пустой
   if (m_wcrowd.empty())
   {
    rrequest_type req = { self(), r.tag };
    send(m_resource, req);

    m_rcrowd.insert(crowd_type::value_type(r.tag, r.customer));
    m_queue.pop_front();

    send(self(), next_type());
   };
   break;
  case request_type::WRITE:
   // если запрос на запись, обе группы должны быть пустыми
   if (m_wcrowd.empty() && m_rcrowd.empty())
   {
    wrequest_type req = { self(), r.tag, r.data };
    send(m_resource, req);

    m_wcrowd.insert(crowd_type::value_type(r.tag, r.customer));
    m_queue.pop_front();

    send(self(), next_type());
   };
   break;
  };
 };
}

// прием и переправка ответов от ресурса
void action(const rreply_type &msg)
{
```

```
  crowd_type::iterator it = m_rcrowd.find(msg.tag);
  assert(it != m_rcrowd.end());
  // переправляем ответ читателю
  send(it->second, msg);
  // удаляем читателя из группы
  m_rcrowd.erase(it);

  send(self(), next_type());
 }
 void action(const wreply_type &msg)
 {
  crowd_type::iterator it = m_wcrowd.find(msg.tag);
  assert(it != m_wcrowd.end());
  // переправляем ответ писателю
  send(it->second, msg);
  // удаляем писателя из группы
  m_wcrowd.erase(it);

  send(self(), next_type());
 }
};
```

Класс `serializer_type` содержит описание типов группы заказчиков и очереди запросов. Тип группы определен как отображение с повторами (`std::multimap`), поскольку на ресурс может быть отправлено много запросов с одинаковым тегом. Для содержания очереди запросов в едином контейнере определена структура `request_type`, в которой могут содержаться запросы обоих типов. Объект сериализатора при создании регистрирует пять обработчиков сообщений, в которых осуществляет работу с очередью запросов и группами заказчиков в соответствии с описанием поведения, приведенным выше.

### 5.3.4. Вычисление количества максимальных значений

В этом примере мы реализуем параллельное выполнение редукционной операции для большого количества аргументов. Допустим, дана некоторая пронумерованная последовательность чисел, и стоит задача определить, сколько в ней содержится элементов, равных их максимальному значению. Количество элементов на заданном интервале последовательности, равных некоторому значению, назовем весом соответствующего значения на этом интервале.

Система актеров, решающая такую задачу, во многом сходна с изображенной на рис. 5.2:

```
def customer (maxval, weight, cust)
[val, num]
    if maxval = NIL
    then
        become customer (val, num, cust)
    else
        if maxval = val
        then
            send [maxval, weight + num] to cust
        else
            if maxval > val
            then
                send [maxval, weight] to cust
```

```
            else
                send [val, num] to cust
            fi
        fi
    fi
end def

def nummax ()
[b, e, cust]
    if e - b = 1
    then
        let v = sequence(b)
        in { send [v, 1] to cust }
    else
        let c = create customer (NIL, NIL, cust),
            n = create nummax (),
            h = floor((e - b) / 2)
        in {
            send [b, b + h, c] to n
            send [b + h, e, c] to self
        }
    fi
end def

def main ()
[start]
    let n = create nummax ()
    in { send [1, 1000, self] to n }
[val, num]
    send [num] to create print ()
end def

send [start] to create main ()
```

Интервал номеров элементов последовательности задается актером `main` и отправляется в сообщении созданному им же актеру с поведением `nummax`. Результат вычислений приходит обратно в виде пары чисел, из которых одно, интересующее нас, выводится актером `print`.

Актер с поведением `nummax`, получив очередное задание на вычисление, проверяет ширину переданного интервала. Если в интервал попадает более одного значения, он может быть разбит на два. В этом случае выполняется два рекурсивных вызова для вычисления максимальных значений и их весов в двух подынтервалах. Для одного рекурсивного вызова создается еще один актер с поведением `nummax`, для другого используется текущий. Если же текущий интервал содержит всего один элемент, он и является единственным максимальным элементом на этом интервале, в связи с чем соответствующее сообщение отсылается актеру-заказчику.

Дерево актеров с поведением `customer` передает пары чисел, содержащие максимальное значение на некотором интервале и его вес. Каждый актер в дереве ожидает прихода двух таких пар, из которых формирует одну и отправляет далее к корню дерева. Формирование отправляемой пары осуществляется на основе оценки содержимого двух пришедших пар. Если максимальные значения в обеих парах отличаются, отправляется та, в которой максимальное значение больше. В противном случае, когда максимальные значения одинаковы, отправляется пара с суммарным весом.

Для получения значения элемента последовательности с заданным номером используется вызов функции `sequence`, которая является общей для всех созданных актеров `nummax`. Может показаться, что это противоречит модели актеров, поскольку в ней не может быть данных, общих для всех. Однако здесь следует учитывать, что эти данные не меняются в процессе работы программы, поэтому не могут в полной мере расцениваться как общие разделяемые данные. Вследствие своей неизменности они могут быть представлены как часть описания поведения актера, что демонстрирует отсутствие противоречия с моделью. К примеру, вместо строки `let v = sequence(b)` при описании программы мы можем использовать следующий фрагмент:

```
let v = case b of
    1: value1
    2: value2
    ...
end case
```

Приведенная выше программа, описанная на языке SAL, реализуется на основе построенных нами классов следующим кодом:

```
int sequence(int i)
{
 return /* ... */;
}

struct calcrequest_type
{
 int begin;
 int end;
 address_type customer;
};
struct result_type
{
 int maxval;
 int weight;
};

class customer_type: public actor_type
{
private:
 address_type m_customer;
 bool m_hasresult;
 result_type m_result;

public:
 struct init_type
 {
  address_type customer;
  bool hasresult;
  result_type result;
 };

 customer_type(const init_type &init):
  m_customer(init.customer),
  m_hasresult(init.hasresult),
  m_result(init.result)
 {
```

```
  add_action<customer_type, result_type>();
 }
 void action(const result_type &msg)
 {
  if (!m_hasresult)
  {
   init_type init = { m_customer, true, msg };
   become<customer_type>(init);
  }
  else
  {
   result_type res =
    (msg.maxval > m_result.maxval) ? msg : m_result;
   if (msg.maxval == m_result.maxval)
    res.weight = msg.weight + m_result.weight;
   send(m_customer, res);
  };
 }
};

class nummax_type: public actor_type
{
public:
 nummax_type(const empty_type &init)
 {
  add_action<nummax_type, calcrequest_type>();
 }
 void action(const calcrequest_type &msg)
 {
  assert(msg.end - msg.begin > 0);
  if (msg.end - msg.begin == 1)
  {
   result_type res = { sequence(msg.begin), 1 };
   send(msg.customer, res);
  }
  else
  {
   customer_type::init_type init = { msg.customer, false, {0} };
   address_type customer = create<customer_type>(init);

   int half = (msg.end - msg.begin) / 2;
   calcrequest_type req1 = { msg.begin, msg.begin + half, customer };
   send(create<nummax_type>(), req1);
   calcrequest_type req2 = { msg.begin + half, msg.end, customer };
   send(self(), req2);
  };
 }
};

class main_type: public actor_type
{
public:
 struct start_type {};

 main_type(const empty_type &init)
```

```
 {
  add_action<main_type, start_type>();
  add_action<main_type, result_type>();
 }
 void action(const start_type &msg)
 {
  calcrequest_type req = { 1, 1000, self() };
  send(create<nummax_type>(), req);
 }
 void action(const result_type &msg)
 {
  send(create<print_type>(), msg.weight);
 }
};

// ...

factory_type factory;
factory.add_definition<customer_type, customer_type::init_type>();
factory.add_definition<nummax_type, actor_type::empty_type>();
factory.add_definition<main_type, actor_type::empty_type>();
factory.add_definition<print_type, actor_type::empty_type>();

scheduler_type sched;
sched.system().send(
 sched.system().create<main_type>(),
 main_type::start_type());
sched.evolve(factory);
```

Поскольку в языке C++ нет средств представления значения `NIL` в целочисленной переменной, в классе `customer_type` предусмотрена переменная `m_hasresult`, говорящая о наличии пришедшего ранее результата. С другой стороны, наличие в языке операции целочисленного деления делает ненужным использование операции `floor` в классе `nummax_type`. Класс `main_type`, как и само описание поведения актера `main`, введен для перенаправления актеру `print` лишь одного числа из пришедшей пары.

### 5.3.5. Поиск выхода из лабиринта

Напоследок покажем, как с помощью актеров может быть решена задача поиска выхода из односвязного лабиринта. Односвязным лабиринтом называется такой, в котором нет отдельно стоящих стенок и замкнутых маршрутов. Структура коридоров такого лабиринта представляет собой дерево, и задача поиска выхода из него эквивалентна обходу этого дерева. Из лабиринта оказывается тем сложнее найти выход, чем больше в нем ветвлений, т.е. чем больше вариантов различных путей от входа в лабиринт (вершины дерева) до очередного тупика (листа дерева). И именно обилие ветвлений в таком лабиринте является фактором, определяющим высокую степень параллелизма задачи поиска выхода из него.

Для решения задачи поиска нами будут использованы актеры, определяемые одним поведением. Каждый актер характеризуется своей позицией в лабиринте и на основе нее принимает решение о дальнейших направлениях движения.

Конфигурация лабиринта считается известной всем актерам. Это не противоречит утверждению об отсутствии общих данных, разделяемых между актерами, поскольку под разделяемыми данными обычно подразумевают такие, которые могут меняться в процессе работы. В нашем случае конфигурация лабиринта не изменяется, поэтому может быть со-

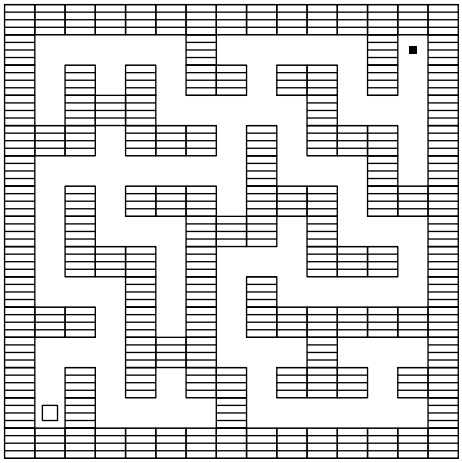


Рис. 5.11. Пример односвязного лабиринта

отнесена не с разделяемыми данными, а с некими правилами, реализованными на уровне описания поведения актеров.

### Объект конфигурации лабиринта

Для предоставления удобного интерфейса актерам конфигурация лабиринта инкапсулирована в соответствующий неизменяемый объект, хранящий данные о стенах, коридорах, входе и выходе. Этот объект позволяет выполнение следующих операций:

- получение начальной позиции путника (`get_in`);
- определение невозможности перемещения в заданную соседнюю позицию из текущей (`is_wall`);
- проверка наличия выхода в текущей позиции (`is_out`).

Позиция путника в лабиринте характеризуется его координатами и направлением движения. Операция `get_in` возвращает позицию путника, обращенного спиной к одной из прилегающих стен.

Для любой позиции предусмотрены операции получения следующей позиции при движении вперед, налево и направо (соответственно, `forward`, `left` и `right`). При движении прямо направление путника в следующей позиции совпадает с текущей, в остальных двух меняется соответствующим образом. Четвертое направление (назад) не рассматривается, поскольку считается, что путник с этого направления пришел, и потому исследовать его незачем. Именно по этой причине путник в начальной позиции встает спиной к одной из прилегающих стен.

Мы приводим здесь довольно упрощенный вариант реализации такого объекта, что налагает на конфигурацию лабиринта некоторые ограничения. В частности, по всей внешней границе должна быть непрерывная стена (рис. 5.11), поскольку иначе на некотором шаге возникнет выход очередной позиции за границы лабиринта. Также в лабиринте не должно быть коридоров шире одной клетки, поскольку широкий участок такого коридора является замкнутым маршрутом, а их в односвязном лабиринте быть не должно.

Поскольку мы предполагаем, что актер движется лишь в трех направлениях, позиция входа должна прилегать хотя бы к одной стене, к которой актер в начальной позиции

становится спиной. В случае, когда вход не прилегает ни к одной стене, т.е. находится на перекрестке с четырьмя возможными направлениями, для решения задачи достаточно внести модификацию, в соответствии с которой в начале пути (на первом шаге) актер исследует не три, а все четыре направления. Есть и более простой вариант: запустить из начальной позиции не одного, а двух актеров в любых двух направлениях, хотя это потребует в среднем в полтора раза больше актеров из-за частичного дублирования исследуемых направлений.

Описанный интерфейс, предоставляющий информацию о конфигурации лабиринта, реализуется следующим классом:

```
// лабиринт
class maze_type
{
private:
 // направления движения и их количество
 enum { NORTH, EAST, SOUTH, WEST, DIR_NUM };
 // конфигурация лабиринта
 const int m_width;
 const string m_data;

public:
 // позиция в лабиринте
 class position_type
 {
 private:
  friend class maze_type;
  // координаты
  int m_x, m_y;
  // направление
  int m_dir;

 public:
  // следующая позиция в прямом направлении
  position_type forward(void) const
  {
   position_type pos = *this;
   if (m_dir == NORTH || m_dir == SOUTH)
    pos.m_y += (m_dir == NORTH) ? -1 : 1;
   if (m_dir == WEST || m_dir == EAST)
    pos.m_x += (m_dir == WEST) ? -1 : 1;
   return pos;
  }
  // следующая позиция при движении налево
  position_type left(void) const
  {
   position_type pos = *this;
   pos.m_dir = (m_dir + DIR_NUM - 1) % DIR_NUM;
   return pos.forward();
  }
  // следующая позиция при движении направо
  position_type right(void) const
  {
   position_type pos = *this;
   pos.m_dir = (m_dir + 1) % DIR_NUM;
   return pos.forward();
```

```
  }
  // оператор вывода информации о позиции
  friend
  ostream & operator <<(ostream &o, const position_type &pos)
  {
   return (o << '(' << pos.m_x << ';' << pos.m_y << ')');
  }
 };

 // обозначения в data: i - вход, o - выход, # - стена;
 // по всей границе должна быть непрерывная стена;
 // вход должен прилегать хотя бы к одной стене;
 // не должно быть замкнутых контуров;
 // не должно быть коридоров шире одной клетки.
 maze_type(int width, const string &data):
  m_width(width), m_data(data)
 {
  assert(m_width > 0);
  assert(m_data.size() % m_width == 0);
  assert(m_data.find_first_not_of(" io#") == string::npos);
 }
 // получение начальной позиции
 position_type get_in(void) const
 {
  // найдем вход
  int i = m_data.find_first_of('i');
  assert(string::size_type(i) != string::npos);
  // вычислим координаты
  position_type pos;
  pos.m_x = i % m_width;
  pos.m_y = i / m_width;
  // найдем прилегающую стену
  for (i = 0; i < DIR_NUM; ++i)
  {
   pos.m_dir = i;
   if (is_wall(pos.forward()))
    break;
  };
  assert(i != DIR_NUM);
  // встанем к ней спиной
  pos.m_dir = (i + DIR_NUM / 2) % DIR_NUM;
  return pos;
 }
 // есть ли тут препятствие?
 bool is_wall(const position_type &pos) const
 {
  return m_data[pos.m_y * m_width + pos.m_x] == '#';
 }
 // тут ли выход?
 bool is_out(const position_type &pos) const
 {
  return m_data[pos.m_y * m_width + pos.m_x] == 'o';
 }
};
```

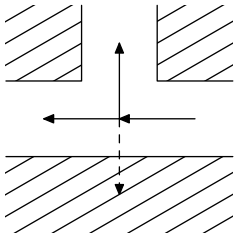


Рис. 5.12. Возможные направления движения путника

Конструктор класса `maze_type` принимает два параметра: ширину лабиринта и строку символов, характеризующих его конфигурацию. Длина строки должна быть кратна ширине лабиринта. В содержимом строки может фигурировать лишь четыре символа, характеризующих соответственно коридор, вход, выход и стену.

Функция `get_in` осуществляет поиск входа в лабиринт и вычисляет его координаты. Далее осуществляется поиск соседней стены, после чего начальной позиции задается направление, повернутое от этой стены на половину полного оборота. Функции `is_wall` и `is_out` просто проверяют наличие для переданной позиции соответствующего символа в строке конфигурации.

Позиция реализуется вложенным классом `position_type`, который определяет три перечисленных ранее операции. Функция `forward` возвращает новую позицию, сдвинутую относительно текущей на единицу в том направлении, которое задано текущей позицией. Функции `left` и `right` делают то же самое, но сменив предварительно направление на соответствующее.

### Поведение путника

Система актеров, решающая задачу поиска выхода из лабиринта, состоит из актеров-путников с одинаковым поведением. Путником такой актер назван весьма условно, поскольку свою позицию в лабиринте он получает при инициализации и далее не меняет, т.е. никуда не движется. В качестве «шага» путник создает нового актера в соседней позиции. На каждом шаге путник помещает новых путников во все направления, куда можно пройти, кроме того, откуда пришел сам (рис. 5.12).

Каждый путник хранит адрес предыдущего, т.е. породившего его, путника, а также адрес актера, которому надлежит отправить информацию о пройденном до выхода пути. При достижении выхода путник отправляет последнему информацию о своей позиции, после чего это делает предыдущий путник и т.д. Описывается соответствующая система актеров следующим образом:

```
def tracer (prn, prev, depth, pos)
[success]
    send [depth, pos] to prn
    send [success] to prev
[move]
    if maze.is_out(pos)
    then
        send [success] to self
    else
        if not maze.is_wall(pos.forward)
        then
            let t = create tracer (prn, self, depth + 1, pos.forward)
```

```
            in { send [move] to t }
        fi
        if not maze.is_wall(pos.left)
        then
            let t = create tracer (prn, self, depth + 1, pos.left)
            in { send [move] to t }
        fi
        if not maze.is_wall(pos.right)
        then
            let t = create tracer (prn, self, depth + 1, pos.right)
            in { send [move] to t }
        fi
    fi
end def

let s = create sink (),
    p = create print ()
in {
    let t = create tracer (p, s, 0, maze.get_in)
    in { send [move] to t }
}
```

Путник описывается поведением `tracer`, в параметрах инициализации принимает адрес актера, получающего информацию о пути до выхода, адрес предыдущего путника, глубину пройденного пути и позицию в лабиринте. В ответ на сообщение `success` путник сообщает свою позицию и глубину пройденного пути, после чего отправляет такое же сообщение предыдущему актеру.

При получении сообщения `move` путник, прежде всего, выясняет, находится ли выход в его позиции, и в случае положительного результата отправляет себе сообщение `success`. Если же выхода тут нет, и предстоит движение дальше, путник проверяет наличие препятствий в трех возможных направлениях. В каждом случае, если препятствия нет, в соответствующей позиции создается новый путник, которому, среди прочих параметров, передается увеличенная на единицу глубина пути.

В начале работы системы создаются актеры `print` и `sink`. Далее создается один путник в начальной позиции, ему назначается нулевая глубина пути и в качестве предыдущего актера передается адрес актера `sink`. Актер `print` получает всю последовательность позиций найденного пути в произвольном порядке (из-за неопределенности порядка доставки), но, поскольку вместе с каждой позицией получает соответствующее значение глубины, имеет возможность их отсортировать.

Реализуется такая программа следующим образом:

```
// конфигурация лабиринта
enum { WIDTH = 15 };
const char c_mazestr[] =
 "###############"
 "#     #     #i#"
 "# # # ## ## # #"
 "# ###     #   #"
 "### ### # ### #"
 "#       #   # #"
 "# # ### ### ###"
 "# #   ### #   #"
 "# ### #   ### #"
 "#   # # #     #"
```

```
 "### # # ########"
 "#   ###   #   #"
 "# # # ## ### ##"
 "#o#     #      #"
 "################";
// объект–лабиринт
const maze_type c_maze(WIDTH, c_mazestr);

// актер–путник
class tracer_type: public actor_type
{
private:
 // актер, которому надлежит послать результат
 address_type m_prn;
 // актер–путник на предыдущем шаге
 address_type m_prev;
 // глубина пройденного пути
 int m_depth;
 // позиция текущего актера
 maze_type::position_type m_pos;

public:
 struct init_type
 {
  address_type print;
  address_type prev;
  int depth;
  maze_type::position_type pos;
 };
 struct success_type {};
 struct move_type {};

 tracer_type(const init_type &init):
  m_prn(init.print),
  m_prev(init.prev),
  m_depth(init.depth),
  m_pos(init.pos)
 {
  add_action<tracer_type, success_type>();
  add_action<tracer_type, move_type>();
 }

 void action(const success_type &msg)
 {
  // формируем информационное сообщение
  print_type::strmsg_type strmsg;
  ostringstream ostr;
  ostr << "step " << setw(3) << m_depth << ": " << m_pos << ends;
  ostr.str().copy(strmsg.str, print_type::strmsg_type::MAX_SIZE);
  // отправляем его актеру print
  send(m_prn, strmsg);
  // передаем сообщение об успехе дальше по цепочке
  send(m_prev, success_type());
 }
```

```
 void action(const move_type &msg)
 {
  // если выход найден, пошлем себе сообщение об успехе
  if (c_maze.is_out(m_pos))
   send(self(), success_type());
  else
  {
   // иначе инициируем проработку до трех новых направлений
   init_type init = { m_prn, self(), m_depth + 1, m_pos };
   if (!c_maze.is_wall(m_pos.forward()))
   {
    init.pos = m_pos.forward();
    send(create<tracer_type>(init), move_type());
   };
   if (!c_maze.is_wall(m_pos.left()))
   {
    init.pos = m_pos.left();
    send(create<tracer_type>(init), move_type());
   };
   if (!c_maze.is_wall(m_pos.right()))
   {
    init.pos = m_pos.right();
    send(create<tracer_type>(init), move_type());
   };
  };
 }
};

// ...

factory_type factory;
factory.add_definition<tracer_type, tracer_type::init_type>();
factory.add_definition<print_type, actor_type::empty_type>();
factory.add_definition<sink_type, actor_type::empty_type>();

scheduler_type sched;
address_type print = sched.system().create<print_type>();
address_type sink = sched.system().create<sink_type>();
tracer_type::init_type init = {
 print, sink, 0, c_maze.get_in()
};
sched.system().send(
 sched.system().create<tracer_type>(init),
 tracer_type::move_type());
sched.evolve(factory);
```

Конфигурация лабиринта в примере соответствует изображенному на рис. 5.11. При отправке актеру `print` текущей позиции и соответствующей глубины пути формируется строка для печати. В остальном программа соответствует приведенному выше описанию.

# Глава 6.

# Квантовые вычисления

Настоящая глава не имеет непосредственного отношения к моделям параллельного программирования, актуальным сегодня, поскольку является скорее обзорной и описывает принципы программирования для аппаратуры, практически полезных экземпляров которой на сегодняшний момент пока не существует. В связи с этим говорить о необходимости реализации сегодня описанных ниже алгоритмов не приходится. Тем не менее, автор счел возможным и целесообразным добавить эту главу, поскольку именно параллелизм вычислений является основой столь высокой производительности квантового компьютера. Понимание того, откуда возникает этот параллелизм и как он используется при выполнении вычислений, может также поспособствовать развитию «параллельного» мышления программиста.

В целях обеспечения наглядности описания будет приведена простая реализация симулятора квантового компьютера. В отличие от предыдущих глав, она не будет распараллеливаться с помощью различных программных интерфейсов, поскольку сама по себе идея создания такого симулятора обладает в основном лишь академической, а не практической, полезностью и преследует иллюстративные цели совершенно иного характера.

## 6.1. Описание вычислительной модели

Обычно, освещая те или иные аспекты квантовых вычислений, рассматривают вопрос с нескольких во многом перекрывающихся точек зрения, а именно:

- инженерно-физические вопросы, касающиеся реализации квантовых компьютеров (на основе ионных ловушек, спинов атомных ядер, поляризованных фотонов и т.п.), а также проблемы подавления воздействия окружающей среды и других сопутствующих вопросов;

- вопросы квантовой механики, включающие в себя мощный математический аппарат, используемый для описания соответствующих задач теоретической физики;

- вопросы квантовой информатики, касающиеся построения логических схем (алгоритмов) в рамках некой абстрактной вычислительной модели, базирующейся на принципах квантовой механики.

Мы не будем затрагивать вопросы физической реализации, упомянем лишь, что прототипы малой разрядности уже создавались. Про квантовые компьютеры с высокой разрядностью говорить пока не приходится, в связи с чем разработка квантовых алгоритмов

на текущий момент актуальна, к сожалению, лишь на бумаге. Более того, даже о принципиальной возможности реализации квантовых компьютеров высокой разрядности ведутся, порой, жаркие дискуссии, однако энтузиазм разработчиков не угасает. Подробнее о вариантах реализации, предложенных на текущий момент, можно прочитать в [6].

Также мы постараемся минимизировать описание используемого в квантовой механике математического аппарата, поскольку нецелесообразно повторять здесь содержимое большого количества литературы, специально посвященной этой дисциплине. Разумеется, нам не удастся полностью исключить математические обоснования описываемых вычислений, в связи с чем необходимый для понимания минимум будет приведен. В свою очередь, этот минимум потребует от читателя базовых знаний линейной алгебры.

В основном, рассмотрение будет посвящено вопросам квантовой информатики. Модель квантовых вычислений будет рассмотрена как некая абстракция, удовлетворяющая заданному набору правил, при этом она будет рассматриваться без привязки к вопросам физической реализации. Такой подход подобен тому, как при построении классических вычислительных схем на основе, к примеру, вентилей «И-НЕ» нам не приходится задумываться об их полупроводниковой реализации.

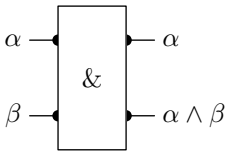
### 6.1.1. Классические обратимые вычисления

Прежде, чем переходить к описанию квантовых вычислений, рассмотрим так называемую модель классических обратимых вычислений, которая к квантовым имеет весьма опосредованное отношение.

Считается, что одним из препятствий к наращиванию производительности классических компьютеров является чрезмерное выделение тепла в процессе вычислений. В соответствии с принципом Ландауэра, любое необратимое вычисление, такое как уничтожение бита, сопровождается выделением некоторого количества энергии [54]. В то же время, если стремиться физически выполнять вычисления обратимым способом, можно избежать выделения тепла и преодолеть барьер производительности. Разумеется, невозможно достичь абсолютной физической обратимости, но путем изолирования системы от внешней среды можно достичь высокой степени приближения к идеалу. С другой стороны, чтобы вычислительный процесс мог быть реализован физически обратимым, он должен быть также обратим логически. Эти обстоятельства подтолкнули ученых к разработке модели обратимых вычислений.

Логическая обратимость вычислительного процесса означает, что любое преобразование в рамках процесса должно взаимно однозначно отображать множество исходных состояний на множество последующих, т.е. не только каждому возможному исходному состоянию должно соответствовать лишь одно состояние-результат, но и, обязательно, наоборот. В частности, это подразумевает, что любое логически обратимое преобразование (вентиль или схема вентилей) имеет на выходе ровно столько же битов, сколько на входе.

В традиционной модели необратимых вычислений базовыми являются операции «И», «ИЛИ», «НЕ». Из них обратимой является лишь одна: операция «НЕ». Каждая из остальных двух операций отображает три разных исходных состояния на одно состояние-результат, т.е. преобразование не является взаимно однозначным. Казалось бы, это очевидно, поскольку у каждой из этих операций по два входа и лишь один выход. Но даже если добавить один дополнительный выход (к примеру, равный одному из входов), ситуация не разрешается, поскольку одному выходному состоянию соответствуют два входных (рис. 6.1). На схеме слева отражены входные биты, справа — выходные; предполагается, что вентили в схемах срабатывают последовательно во времени слева направо, что является стандартным способом изображения обратимых схем.




| $\alpha$ | $\beta$ | $\alpha$ | $\alpha \wedge \beta$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

Рис. 6.1. Классический вентиль «И» с дополнительным выходом

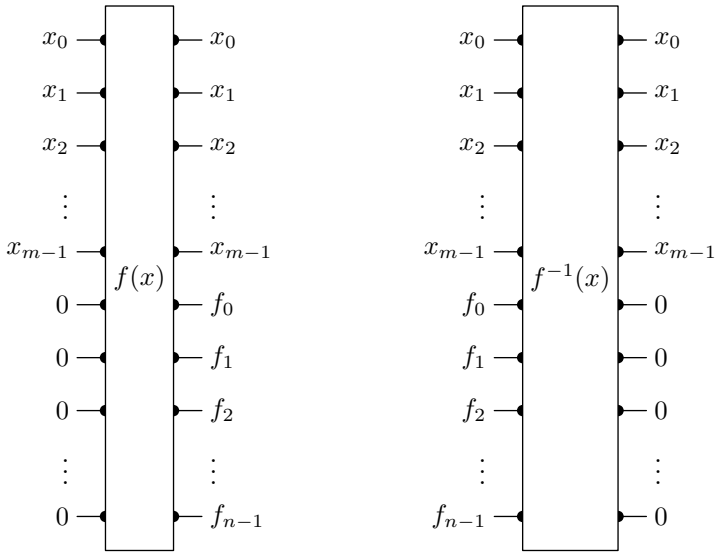
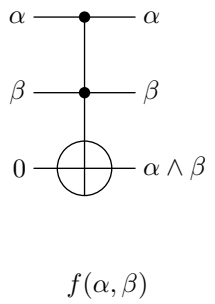



Рис. 6.2. Общая схема обратимого вычисления необратимой функции и последующей очистки результата

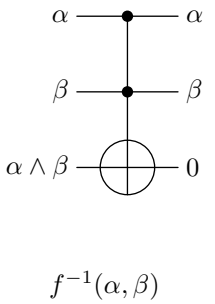
Доказано, что любое вычисление может быть выполнено обратимым способом [61]. В связи с этим в рамках модели обратимых вычислений возникает множество приемов, характерных именно для нее и призванных обеспечить возможность выполнения произвольных вычислений. В частности, для выполнения вычислений, которым принципиально необходимо быть необратимыми (к примеру, вычисление какой-либо периодической функции $f(x)$), увеличивают количество битов, к которым подключается схема (рис. 6.2). Множество битов разделено на два битовых поля (регистра): регистр аргумента $x$ вычисляемой функции и вспомогательный регистр, в который в результате вычислений будет помещено значение функции от соответствующего аргумента. Каждый бит вспомогательного регистра (служебный бит, ancilla) задается на входе равным определенному значению (константе). На выходе первый регистр не меняется и содержит значение аргумента $x$, во втором регистре содержится значение $f(x)$.

Поскольку значение аргумента $x$ содержится в выходных битах, возможно однозначное определение первоначального набора битов на основе набора битов результата. Т.е. отображение множества входных состояний на множество выходных может быть построено взаимно однозначным, а значит обратимым. Здесь следует иметь в виду, что вычисления в этом случае выполняются корректно лишь при условии, что регистр, предназначенный для значения $f(x)$, на входе всегда содержит наперед заданное число (к примеру, ноль), и именно с этим расчетом составляется схема вентилей. В остальных случаях результат на выходе окажется некорректным, хотя этот вопрос нас не интересует.

Есть еще один тип вспомогательных битов: временные (черновые, scratch). Они, как и




$f(\alpha, \beta)$




$f^{-1}(\alpha, \beta)$

| $\alpha$ | $\beta$ | in | $\alpha$ | $\beta$ | out |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| ~~0~~ | ~~0~~ | ~~1~~ | ~~0~~ | ~~0~~ | ~~1~~ |
| 0 | 1 | 0 | 0 | 1 | 0 |
| ~~0~~ | ~~1~~ | ~~1~~ | ~~0~~ | ~~1~~ | ~~1~~ |
| 1 | 0 | 0 | 1 | 0 | 0 |
| ~~1~~ | ~~0~~ | ~~1~~ | ~~1~~ | ~~0~~ | ~~1~~ |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

Рис. 6.3. Обратимая схема «И»

в модели необратимых вычислений, используются для хранения промежуточных данных. К примеру, для накопления суммы в цикле вычисления ряда. В необратимых вычислениях временные данные могут быть просто «забыты» и впоследствии перезаписаны другими данными независимо от их значения. Однако в случае обратимых вычислений это неприемлемо, что обуславливает необходимость выполнения другого характерного для них приема, а именно очистки промежуточных результатов (обратное вычисление, uncomputing).

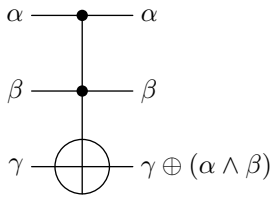
Если значение $f(x)$ является временным (к примеру, так может быть, если схема на рис. 6.2 — часть более крупной схемы), после его вычисления и использования следует сбросить содержимое второго регистра в исходное значение (в нашем примере ноль). В некоторый момент нам снова могут потребоваться временные данные (к примеру, чтобы вычислить ту же функцию от другого аргумента и поместить результат в этот же регистр). Если мы не очистим его после первичного использования, результат вторичного вычисления функции окажется некорректным. Очистка результата вычисления $f(x)$ выполняется другой схемой (как правило, не менее громоздкой), выполняющей обратный переход к начальному состоянию и обозначаемой зачастую $f^{-1}(x)$ (рис. 6.2).

Одни и те же временные биты могут потребоваться для выполнения разных схем, поэтому удобно всегда ориентироваться на унифицированные их входные значения, в связи с чем обычно используют 0. Если конкретной схеме требуется другое значение во временном регистре, его всегда можно сформировать путем использования вентилей NOT.

В качестве простейшего примера схемы, изображенной на рис. 6.2, рассмотрим обратимую реализацию операции «И» (рис. 6.3). Схема подключается к трем битам (как уже было видно выше, двух оказывается недостаточно). Первые два бита — регистр аргумента, последний — регистр результата, который для получения корректного значения должен быть задан на входе равным нулю.

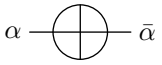
Схема содержит единственный вентиль, подключенный ко всем трем битам. Этот вентиль называется вентилем Тоффоли (CCNOT, controlled-controlled-NOT) и выполняет инверсию одного из битов (нижнего на рис. 6.3) в случае, если два остальных бита установлены в единицу. Таким образом, схема осуществляет вычисление функции $f(x) = 0 \oplus (\alpha \wedge \beta) = \alpha \wedge \beta$. Очистка бита результата при необходимости может быть произведена с помощью той же схемы, поскольку $f^{-1}(x) = f(x) \oplus (\alpha \wedge \beta) = (\alpha \wedge \beta) \oplus (\alpha \wedge \beta) = 0$. В таблице истинности на рис. 6.3 вычеркнуты строки, которые не задействуются в обоих режимах, поскольку в этих режимах соответствующие наборы значений на вход схемы не поступают. Последняя строка таблицы задействуется только в режиме очистки результата.

Вентиль Тоффоли (рис. 6.4) является универсальным элементом, на основе которого в принципе можно произвести любые классические обратимые вычисления, подобно тому, как в модели необратимых вычислений универсальными вычислительными элементами являются вентили «И-НЕ» (NAND, операция «штрих Шеффера») и «ИЛИ-НЕ» (NOR,




| $\alpha$ | $\beta$ | $\gamma$ | $\alpha$ | $\beta$ | $\gamma \oplus (\alpha \wedge \beta)$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

Рис. 6.4. Обратимый вентиль Toffoli




| $\alpha$ | $\bar{\alpha}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

Рис. 6.5. Обратимый вентиль NOT

операция «стрелка Пирса»). Также при построении обратимых схем используются некоторые вентили, не являющиеся универсальными, к примеру, однобитный вентиль NOT, реализующий операцию «НЕ» (рис. 6.5), и двубитный вентиль CNOT (controlled-NOT), выполняющий инверсию одного бита, если другой установлен в единицу (рис. 6.6).

Чуть более сложным примером построения обратимой схемы является реализация операции «ИЛИ» (рис. 6.7). Схема снова содержит вентиль Тоффоли, а также четыре вентиля NOT. Обозначение последних на схеме является альтернативным приведенному на рис. 6.5, и происходит от обозначения одного из операторов Паули [35]. Вентили NOT на схеме служат для временной инверсии входов, благодаря чему вентиль Тоффоли инвертирует входную единицу лишь в случае, когда оба остальные входа равны нулю.

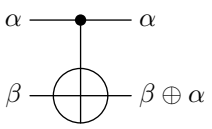
В качестве другого примера рассмотрим двубитную схему SWAP, выполняющую обмен значениями двух входных битов (рис. 6.8).

На рисунке приведено обозначение схемы и ее реализация на основе трех вентилей CNOT. Поскольку вентиль CNOT выполняет операцию «ИСКЛЮЧАЮЩЕЕ ИЛИ», эта схема работает аналогично известному способу обмена значениями двух целочисленных переменных без использования третьей:

```
inline
void swap(int &a, int &b)
{
 a ^= b;
 b ^= a;
 a ^= b;
}
```

К любой схеме может быть подключен дополнительный управляющий бит. Такая схема работает в штатном режиме, лишь если управляющий бит на входе установлен в единицу, противном же случае схема не производит никаких изменений (не работает). Таким образом строится вентиль CNOT на основе NOT, вентиль Тоффоли на основе CNOT и т.д. В общем случае подобная схема CC...CCNOT, как и любая другая управляемая схема, может иметь произвольное количество управляющих битов.

Если мы добавим управляющий бит к вентилю SWAP, мы получим вентиль Фредкина (CSWAP, controlled-SWAP), производящий обмен значениями двух входных битов в случае, если третий установлен в единицу (рис. 6.9). Вентиль Фредкина, как и вентиль Тоффоли,




| $\alpha$ | $\beta$ | $\alpha$ | $\beta \oplus \alpha$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 |

Рис. 6.6. Обратимый вентиль CNOT

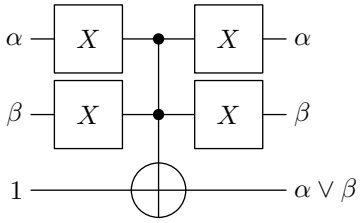



Рис. 6.7. Обратимая схема «ИЛИ»

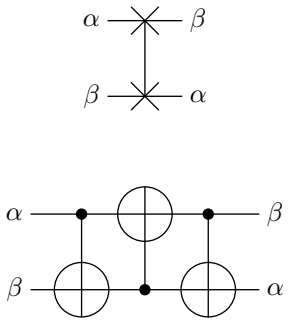
также является универсальным вентилем, на основе которого можно построить любое классическое обратимое вычисление. На рис. 6.9 приведено обозначение вентиля Фредкина, на рис. 6.10 — его логическая схема на основе управляемого вентиля SWAP и реализация с помощью трех вентилей Тоффоли. Здесь виден общий подход к реализации управляемых схем: если схема управляется некоторым битом, контроль от этого бита передается на каждый составляющий вентиль этой схемы.

Другим общим подходом, также широко применяемым в построении обратимых схем, является изменение схемой значения одного вспомогательного бита без изменения остальных (рис. 6.11). Примеры таких схем уже были приведены выше (реализация обратимых схем «И» и «ИЛИ»). В общем случае схема, реализующая некоторую функцию $f(x)$ (рис. 6.2), может быть построена в виде последовательности из $n$ таких схем. При этом схемой в процессе вычисления может быть использовано некоторое количество дополнительных битов для хранения каких-либо временных данных (scratch register), которые зачастую на схемах верхнего уровня не изображаются. После завершения вычислений эти биты должны быть возвращены схемой в исходное состояние.

### 6.1.2. Квантовый бит и принцип суперпозиции

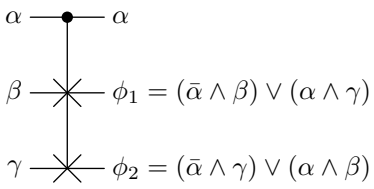
Как известно, состояние классического компьютера описывается набором битов, т.е. совокупностью состояний каждого бита в отдельности. Каждый классический бит может находиться в одном из двух состояний: 0 или 1. Таким образом, если компьютер содержит $n$ битов, он может находиться в одном из $2^n$ состояний — комбинаций состояний отдельных битов. При этом состояние, в котором пребывает компьютер, в любой момент четко определено. При выполнении любой логической операции (срабатывании вентиля) система переходит из одного состояния в другое (рис. 6.12).

Состояние квантового компьютера также описывается набором битов. Однако, в отличие от классического бита, квантовый бит (кубит, quantum bit, q-bit, qubit) может находиться не только в одном из двух базисных состояний, но также в их суперпозиции (нормированной линейной комбинации). При этом кубит в какой-то степени находится и в состоянии 0, и в состоянии 1. Соответственно, возможные состояния квантового компьютера, содержащего $n$ кубитов, описываются непрерывным множеством суперпозиций из $2^n$ взаимоисключающих (несовместных) базисных состояний. Выполнение любой нетривиаль-




| $\alpha$ | $\beta$ | $\beta$ | $\alpha$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Рис. 6.8. Схема SWAP




| $\alpha$ | $\beta$ | $\gamma$ | $\alpha$ | $\phi_1$ | $\phi_2$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

Рис. 6.9. Вентиль Фредкина (controlled-SWAP)

ной вычислительной операции предполагает в общем случае переход системы кубитов из одной суперпозиции состояний в другую (рис. 6.13).

Квантовомеханический принцип суперпозиции гласит, что если квантовая система может пребывать в некотором наборе различных состояний, она может пребывать также и в состоянии, описываемом их линейной комбинацией с произвольными комплексными коэффициентами. Квадрат модуля коэффициента определяет вероятность нахождения системы в соответствующем состоянии. Поскольку возможные состояния компьютера взаимоисключают друг друга, сумма вероятностей, т.е. квадратов модулей коэффициентов, должна быть равна единице.

Опуская множество не имеющих прямого отношения к теме изложения деталей, скажем, что существует ряд физических объектов (двухуровневых квантовых систем), поведение которых описывается столь непривычной для нас моделью кубита [82, 6]. Произвольное состояние квантовой системы отражается вектором в гильбертовом пространстве состояний $\mathcal{H}$ соответствующей системы, в котором выбирается некоторый ортонормированный вычислительный базис. В случае кубита рассматривают вычислительный базис из двух векторов $\{|0\rangle, |1\rangle\}$, на основе которого произвольное состояние кубита описывается двумерным комплексным вектором:

$$\begin{gathered} |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}; \\ |\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}, \quad \langle\psi|\psi\rangle = |\alpha_0|^2 + |\alpha_1|^2 = 1. \end{gathered} \tag{6.1}$$

Комплексные коэффициенты $\alpha_0$ и $\alpha_1$ называются амплитудами соответствующих базисных состояний $|0\rangle$ и $|1\rangle$. В случае, когда один из них равен нулю, состояние кубита эквивалентно одному из базисных состояний.

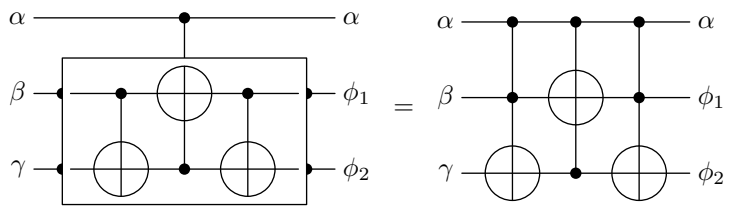



Рис. 6.10. Реализация вентиля Фредкина

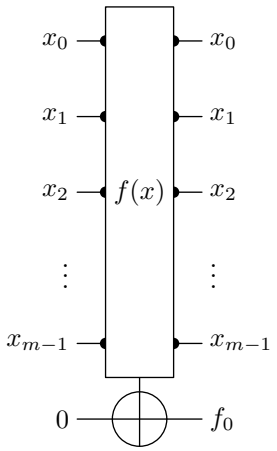



Рис. 6.11. Обратимая схема вычисления необратимой функции с двумя возможными значениями

Обозначение $|\cdot\rangle$ является общепринятым для представления векторов квантового состояния (нотация Дирака). Оно происходит из разбиения на две части обозначения скалярного произведения: $\langle\cdot|\cdot\rangle = \langle\cdot|\,|\cdot\rangle$ (bracket = bra ket). Поскольку по смыслу скалярное произведение двух векторов есть результат умножения сопряженного вектора на вектор, векторы-столбцы обозначаются $|\cdot\rangle$ (кет-векторы), тогда как векторы-строки (сопряженные векторы-столбцы) обозначаются $\langle\cdot|$ (бра-векторы):

$$|\psi\rangle = \begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix}, \quad \langle\psi| = (|\psi\rangle)^\dagger = \begin{pmatrix}\alpha_0^* & \alpha_1^*\end{pmatrix}.$$

Ортонормированный набор базисных векторов в общем случае может быть выбран произвольно, хотя в нашем случае это не представляет существенного интереса.

Все пространство состояний кубита представляется четырехмерной единичной сферой (вследствие наличия четырех действительных параметров). Представив амплитуды в показательной форме, получим:

$$\begin{aligned}|\psi\rangle = \alpha_0\,|0\rangle + \alpha_1\,|1\rangle = \rho_0 e^{i\varphi_0}\,|0\rangle + \rho_1 e^{i\varphi_1}\,|1\rangle = \\ = e^{i\varphi_0}(\rho_0\,|0\rangle + \rho_1 e^{i(\varphi_1-\varphi_0)}\,|1\rangle) = e^{i\varphi_0}\,|\psi'\rangle\,.\end{aligned}$$

Общий фазовый сдвиг всего набора коэффициентов $\{\alpha_0, \alpha_1\}$ не имеет для нас значения, поскольку не может быть измерен. Имеет значение лишь соотношение сдвигов коэффициентов между собой, т.к. оно, хоть и также не может быть напрямую измерено, может, тем не менее, отразиться на результате дальнейших вычислений в виде квантовой интерференции

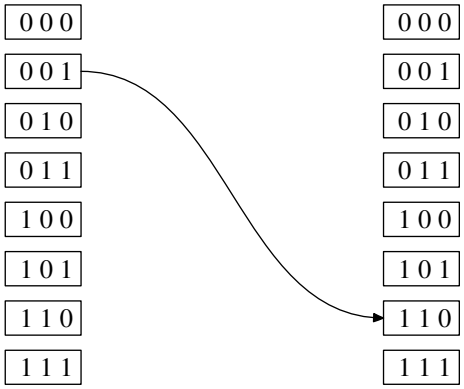



Рис. 6.12. Пример смены состояния классического компьютера

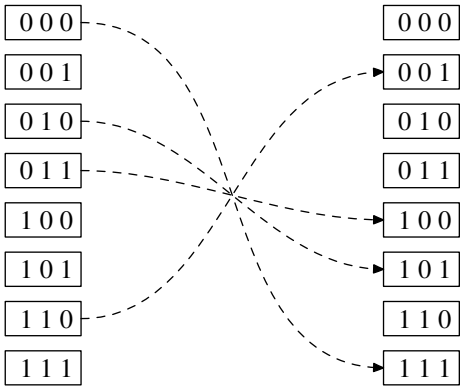



Рис. 6.13. Пример смены суперпозиции состояний квантового компьютера

[82]. Поскольку общий фазовый сдвиг не важен, квантовые состояния $|\psi\rangle$ и $|\psi'\rangle$ эквивалентны, что дает нам возможность исключить один из четырех параметров и однозначно представить любое состояние кубита $|\psi\rangle$ (кроме одного) в виде вектора, направленного в точку трехмерной полусферы (рис. 6.14). В данном случае трехмерное пространство получено объединением одномерного пространства неотрицательных коэффициентов $\rho_0$ (вдоль вектора $|0\rangle$) и двумерного пространства коэффициентов $\rho_1 e^{i(\varphi_1-\varphi_0)}$ (лежащих на плоскости, ортогональной $|0\rangle$). Любое множество эквивалентных состояний кубита, кроме одного, однозначно соответствует точке на такой полусфере (и окружности на четырехмерной сфере, вследствие наличия произвольного общего фазового сдвига $e^{i\varphi_0}$). Исключением в нашем случае является состояние $|1\rangle$, для которого множество эквивалентных состояний на рис. 6.14 представлено не точкой, а окружностью (вследствие того, что угол $\varphi_0$ не определен и, соответственно, не определена разница $\varphi_1 - \varphi_0$).

Чаще используется другое представление, когда пространство состояний двухуровневой квантовой системы отображается на сферу (сфера Блоха). Такая сфера, по сути, получается путем «растягивания» приведенной выше полусферы, при этом ее граничная окружность стягивается в точку. Мы не приводим такое представление, поскольку оно не отражает ортогональности базисных векторов и потому отчасти теряет в наглядности. Стоит отметить однако, что представление в виде сферы Блоха имеет и свои удобства. К примеру, любое множество эквивалентных состояний, в том числе $|1\rangle$, отображается на ней

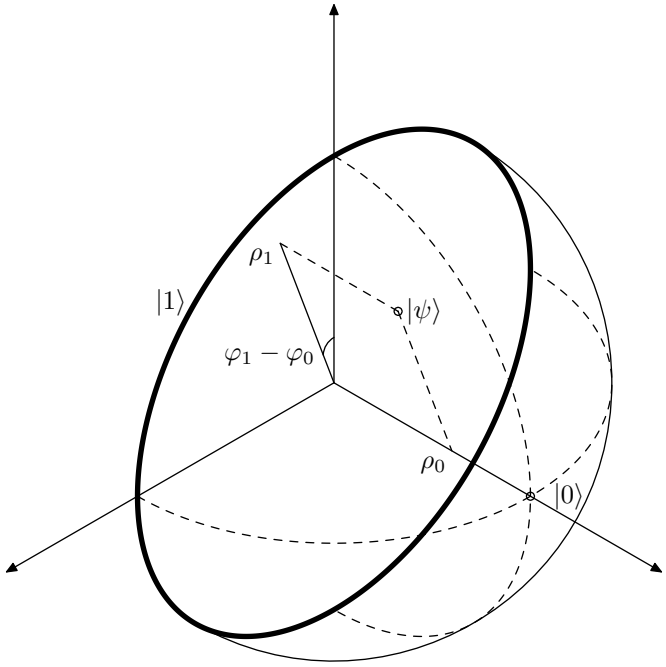



Рис. 6.14. Пространство состояний кубита

лишь одной точкой, а любой базис из двух ортогональных векторов — противоположными точками сферы.

### 6.1.3. Системы кубитов и квантовая запутанность

Многоразрядный квантовый компьютер строится на основе системы из $n$ кубитов. Вычислительный базис $2^n$-мерного гильбертова пространства состояний такой системы представляет собой ортонормированный набор векторов $\{|k\rangle\}_{k=0}^{2^n-1}$. Каждый вектор $|k\rangle$ вычислительного базиса является тензорным произведением $n$ базисных векторов отдельных кубитов и характеризует состояние, при котором каждый кубит пребывает в состоянии $|k_i\rangle, i = 0, \ldots, n-1$, где $k_i \in \{0, 1\}$ — значение соответствующего бита в двоичном разложении числа $k$:

$$|k\rangle = |k_{n-1} \cdots k_1 k_0\rangle = |k_0\rangle \otimes |k_1\rangle \otimes \cdots \otimes |k_{n-1}\rangle, \\ k = 0, \ldots, 2^n - 1.$$

Следует обратить внимание на то, что биты в числе, как обычно, нумеруются справа налево, тогда как базисные векторы соответствующих подсистем в произведении следуют слева направо. Выбор того или иного порядка зависит от соглашений, мы же в дальнейшем будем придерживаться именно такого порядка. Кубиты в изображениях схем будем нумеровать сверху вниз.

Произвольное состояние $\psi$ квантовой системы описывается суперпозицией базисных векторов:

$$|\psi\rangle = \sum_{k=0}^{2^n-1} \alpha_k |k\rangle, \quad \langle\psi|\psi\rangle = \sum_{k=0}^{2^n-1} |\alpha_k|^2 = 1. \tag{6.2}$$

Любое состояние однозначно описывается $2^n$-мерным комплексным вектором $|\psi\rangle$:

$$|\psi\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \cdots \\ \alpha_k \\ \cdots \\ \alpha_{2^n-1} \end{pmatrix} \begin{array}{|l} 00\ldots00 \\ 00\ldots01 \\ \cdots \\ k_{n-1}\cdots k_0 \\ \cdots \\ 11\ldots11 \end{array}$$

Нумерация элементов вектора состояния выполняется нами с нуля. Это не согласуется с общепринятым подходом, однако позволяет нам провести естественное соответствие между номером элемента и базисным состоянием системы кубитов. Как и в случае одного кубита, общий фазовый сдвиг всей системы физического смысла не несет, т.е. домножение суперпозиции $|\psi\rangle$ на произвольный множитель $e^{i\varphi}$ состояния не меняет. Элементы векторов вычислительного базиса определяются значениями символа Кронекера, т.е. $|k\rangle = (\delta_{lk})_{l=0}^{2^n-1}$:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \\ \cdots \\ 0 \\ \cdots \\ 0 \end{pmatrix}; \quad \ldots \quad |k\rangle = \begin{pmatrix} 0 \\ 0 \\ \cdots \\ 1 \\ \cdots \\ 0 \end{pmatrix}; \quad \ldots \quad |2^n-1\rangle = \begin{pmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ \cdots \\ 1 \end{pmatrix}. \tag{6.3}$$

Важный момент, который необходимо понимать, заключается в следующем. Некоторое состояние системы может быть тензорным произведением состояний ее подсистем. Однако в общем случае произвольная суперпозиция состояний квантовой системы в виде произведения суперпозиций отдельных ее кубитов не представляется. Допустим, к примеру, что квантовая система содержит два независимых кубита, находящихся соответственно в состояниях $\alpha'_0|0\rangle + \alpha'_1|1\rangle$ и $\alpha''_0|0\rangle + \alpha''_1|1\rangle$. Тогда состояние всей системы описывается как:

$$\begin{gathered}(\alpha'_0|0\rangle + \alpha'_1|1\rangle) \otimes (\alpha''_0|0\rangle + \alpha''_1|1\rangle) = \\ = \alpha'_0\alpha''_0|0\rangle \otimes |0\rangle + \alpha'_1\alpha''_0|1\rangle \otimes |0\rangle + \alpha'_0\alpha''_1|0\rangle \otimes |1\rangle + \alpha'_1\alpha''_1|1\rangle \otimes |1\rangle = \\ = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle.\end{gathered}$$

Казалось бы, интуиция подсказывает, что допустима и обратная операция — декомпозиция произвольного состояния системы на состояния подсистем, но это не так. К примеру, состояние $\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle$ со всеми ненулевыми коэффициентами не может быть представлено в виде тензорного произведения состояний двух отдельных кубитов, поскольку тогда равенство нулю коэффициента $\alpha_{11}$ говорит о равенстве нулю как минимум одного из коэффициентов $\alpha'_1$ или $\alpha''_1$, а это противоречит тому, что не равны нулю коэффициенты $\alpha_{10}$ и $\alpha_{01}$.

Такое явление, когда состояние системы не может быть представлено в виде произведения состояний подсистем, т.е. подсистемы не имеют четко определенного состояния, носит название запутанности (сцепленности, entanglement) квантового состояния. Понятие запутанного состояния характерно только для систем кубитов, поскольку является следствием зависимости состояний одних кубитов от состояний других.

Состояние системы, представимое в виде некоторого вектора $|\psi\rangle$, называется чистым (pure). Если такое состояние может быть представлено в виде произведения состояний подсистем, т.е. не является запутанным, оно называется разложимым (separable). Если чистое состояние системы разложимо, состояния подсистем, на которые оно раскладывается,

также являются чистыми. Если система находится в запутанном состоянии, каждая ее подсистема пребывает в смешанном (mixed) состоянии, т.е. с некоторой вероятностью пребывает в различных чистых состояниях (которые, в свою очередь, могут быть суперпозициями базисных состояний). Чем выше степень смешанности (минимальная шенноновская энтропия) состояния подсистемы некоторой системы, находящейся в чистом состоянии, тем сильнее сцепленность ее с другими подсистемами. Смешанное состояние системы не может быть представлено вектором $|\psi\rangle$, в связи с чем пользуются представлением в виде матрицы плотности $\rho$ [35]. Чем выше ее ранг, тем выше степень запутанности подсистемы с другими подсистемами. Ранг матрицы плотности, построенной для чистого состояния, равен единице.

Запутанность, явно или неявно, является одним из основных инструментов в квантовых вычислениях. Если состояния двух подсистем сцеплены, измерение одного из них повлияет на состояние другой: она перейдет из смешанного состояния в чистое. Как следствие, результат измерения состояния второй будет коррелировать с результатом измерения первой.

Запутанные состояния кубитов, удаленных друг от друга в пространстве, используются, в частности, в квантовой телепортации и сверхплотном кодировании. О квантовой телепортации и сверхплотном кодировании можно прочесть в [35, 76, 82], мы же опустим эти вопросы, поскольку они не имеют непосредственного отношения к теме нашего рассмотрения — квантовому параллелизму.

### 6.1.4. Унитарные преобразования и квантовые схемы

Из квантовой механики известно, что любое изменение состояния изолированной системы описывается унитарным преобразованием, т.е. таким обратимым линейным преобразованием $\hat{U}$, при котором длина преобразуемого вектора остается неизменной. Это представляется логичным, если вспомнить, что квадрат длины вектора в нашем случае равен полной вероятности пребывания системы в каких-либо ее базисных состояниях, т.е. всегда равен единице. Обратимость преобразования также является естественным следствием того факта, что законы квантовой физики проявляются лишь на микроскопическом уровне, на котором все процессы являются обратимыми.

Любое изменение, выполняемое квантовым компьютером в процессе вычислений, производит унитарное преобразование состояния системы $|\psi\rangle$ (вращение вектора в пространстве состояний). Унитарные преобразования в конечномерном гильбертовом пространстве состояний могут быть представлены в виде унитарной матрицы, т.е. такой квадратной матрицы $\hat{U}$, что сопряженная к ней является обратной ($\hat{U}^\dagger = \hat{U}^{-1}$). Допустим, к примеру, что преобразование $\hat{U}$ переводит каждый базисный вектор пространства состояний в некоторую суперпозицию с коэффициентами $\alpha_{lk}$:

$$|k\rangle \quad \mapsto \quad \hat{U}\,|k\rangle = \alpha_{0k}\,|0\rangle + \cdots + \alpha_{2^n-1k}\,|2^n-1\rangle = \sum_{l=0}^{2^n-1} \alpha_{lk}\,|l\rangle\,,$$
$$k = 0, \ldots, 2^n - 1.$$

Тогда эти коэффициенты составляют столбцы матрицы $\hat{U}$:

$$|k\rangle \mapsto \hat{U}\,|k\rangle\,, \quad \hat{U} = (\alpha_{lk})_{l,k=0}^{2^n-1} = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots & \alpha_{02^n-1} \\ \alpha_{10} & \alpha_{11} & \cdots & \alpha_{12^n-1} \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_{2^n-10} & \alpha_{2^n-11} & \cdots & \alpha_{2^n-12^n-1} \end{pmatrix}.$$

Описание преобразования над состояниями в виде матриц является аналогом таблиц истинности для вентилей и схем, используемых в классических вычислениях. При этом битовое представление каждого номера строки или столбца соответствует набору базисных состояний соответствующих кубитов (при условии нумерации строк и столбцов с нуля).

Унитарность преобразований предполагает, что все квантовые вентили и схемы обратимы, что, в свою очередь, дает возможность использовать наработки классической модели обратимых вычислений. Сама по себе она не имеет прямого отношения к квантовому компьютеру, однако ее достижения естественным образом распространяются и на модель квантовых вычислений [60], поскольку классические вычисления рассматриваются как частный случай. Матрица унитарного преобразования, выполняемого классическими обратимыми вентилями, имеет в каждой строке и в каждом столбце лишь одно ненулевое значение, равное единице. К примеру, так выглядят матрицы преобразований, выполняемых вентилями NOT и CNOT (во втором случае воздействие производится на нулевой кубит):

$$\hat{U}_{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \quad \hat{U}_{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

В отличие от классической модели, в квантовой реализации классические обратимые вентили производят преобразование в общем случае над суперпозицией состояний. Следует отметить, что вентили Тоффоли и Фредкина в модели квантовых вычислений уже не являются универсальными.

Часто прибегают к представлению операторов, основанном на внешнем произведении базисных бра-векторов с кет-векторами — результатами преобразования. К примеру, приведенные выше операторы могут быть описаны так:

$$\hat{U}_{NOT} = |0\rangle\langle 1| + |1\rangle\langle 0|;$$
$$\hat{U}_{CNOT} = |00\rangle\langle 00| + |01\rangle\langle 01| + |11\rangle\langle 10| + |10\rangle\langle 11|.$$

В этом случае произвольный вектор, преобразуемый оператором, с помощью бра-векторов раскладывается по проекциям на базисные векторы, после чего новый вектор формируется в виде линейной комбинации кет-векторов на основе длины полученных проекций.

В дополнение к вентилям, унаследованным из классической модели, в квантовых вычислениях используются несколько новых вентилей, не реализуемых классически. Преобразования сразу над большим количеством кубитов сложно реализуются физически, вследствие чего для построения таких схем используют последовательности однокубитных и двухкубитных вентилей. Доказано, что любое квантовое вычисление может быть сколь угодно точно выполнено на базе универсального набора из классического вентиля CNOT и множества однокубитных квантовых вентилей [52]. Из квантовых вентилей нами будут использованы в дальнейшем вентили Адамара и фазового сдвига.

Вентиль H выполняет над кубитом преобразование Адамара (Уолша-Адамара, Walsh–Hadamard transform), переводя каждое базисное состояние в равновзвешенную суперпозицию обоих базисных состояний:

$$|0\rangle \mapsto \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle, \quad |1\rangle \mapsto \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle;$$
$$\hat{H} = \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right)\langle 0| + \left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)\langle 1| = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}.$$

Вентиль фазового сдвига оставляет неизменным состояние $|0\rangle$, а состоянию $|1\rangle$ смещает фазу на заданный угол $\theta$:

$$|0\rangle \mapsto |0\rangle, \quad |1\rangle \mapsto e^{i\theta}|1\rangle;$$
$$\hat{R}(\theta) = |0\rangle\langle 0| + e^{i\theta}|1\rangle\langle 1| = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}.$$

В процессе вычислений, как правило, на каждом шаге выполняется преобразование не над всеми кубитами системы, а лишь над некоторым их подмножеством. Допустим, к примеру, что выполняется некоторое преобразование $\hat{U}'$ над первыми $m$ кубитами системы, оставляя без изменений остальные. В таком случае полный оператор $\hat{U}$, воздействующий на систему, определяется тензорным произведением оператора $\hat{U}'$ и тождественного (единичного) оператора $\hat{I}$, воздействующего на каждый из остальных $n-m$ кубитов системы:

$$\hat{U} = \hat{U}' \otimes \hat{I}_{n-m} = \hat{U}' \otimes \hat{I} \otimes \cdots \otimes \hat{I} = \begin{pmatrix} U' & 0 & & 0 \\ 0 & U' & & 0 \\ & & \ddots & \\ 0 & 0 & & U' \end{pmatrix}; \quad \hat{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Преобразование, выполняемое таким оператором над вектором состояния $|\psi\rangle$, может быть представлено через суперпозицию разложимых состояний, где каждый элемент суперпозиции есть тензорное произведение состояния набора кубитов, подвергаемых преобразованию, и базисного состояния остальных кубитов:

$$|\psi\rangle = \sum_{k=0}^{2^n-1} \alpha_k |k\rangle = \sum_{l=0}^{2^{n-m}-1} |\psi'_l\rangle \otimes |l\rangle \quad \mapsto \quad \hat{U}|\psi\rangle = \sum_{l=0}^{2^{n-m}-1} \hat{U}|\psi'_l\rangle \otimes |l\rangle. \tag{6.4}$$

Другой пример, требующий рассмотрения — произвольная схема $\hat{U}''$, воздействующая на $m$ кубитов и управляемая еще одним. Матрица такого преобразования строится путем совмещения матрицы $\hat{U}''$, единичной и двух нулевых матриц:

$$\hat{U} = \begin{pmatrix} I_m & 0 \\ 0 & U'' \end{pmatrix}, \quad \hat{I}_m = \hat{I} \otimes \cdots \otimes \hat{I}.$$

Выполняемое с помощью такого оператора преобразование над системой из $m+1$ кубитов, находящейся в состоянии $|\psi\rangle$, описывается на основе суперпозиции разложимых состояний следующим образом:

$$|\psi\rangle = |\psi''_0\rangle \otimes |0\rangle + |\psi''_1\rangle \otimes |1\rangle \quad \mapsto \quad \hat{U}|\psi\rangle = |\psi''_0\rangle \otimes |0\rangle + \hat{U}''|\psi''_1\rangle \otimes |1\rangle. \tag{6.5}$$

В обоих рассмотренных случаях предполагалось, что подсхема, выполняющая преобразование $\hat{U}'$ или $\hat{U}''$, подключается строго по порядку к первым $m$ кубитам системы. В общем

случае подсхема может быть подключена к произвольным кубитам в произвольном порядке. Для построения матрицы такого преобразования удобно допустить временную перестановку кубитов так, чтобы кубиты, к которым подключается подсхема, оказались в нужном порядке в начале системы. Допустим, некоторая целочисленная обратимая функция $t(k)$ осуществляет соответствующее преобразование номера базисного состояния $k$ (требуемую перестановку битов в двоичном представлении $k$). Тогда матрица выполняемого подсхемой преобразования формируется в виде произведения матрицы $\hat{U}$, построенной для описанного выше случая подключения к первым $m$ кубитам, и двух матриц перестановки:

$$|\psi\rangle \mapsto \hat{P}^\dagger \hat{U} \hat{P} |\psi\rangle; \quad \hat{P} = \sum_{k=0}^{2^n-1} |t(k)\rangle \langle k| .$$

Матрица $\hat{P}$ осуществляет перестановку элементов вектора состояния $|\psi\rangle$ так, чтобы обеспечить для каждого номера базисного состояния $k$ требуемую перестановку значений битов. Матрица $\hat{P}^\dagger$, соответственно, выполняет обратное преобразование. Такой механизм построения матрицы преобразования эквивалентен явному осуществлению перестановки кубитов (к примеру, путем подключения некоторого количества вентилей SWAP) и подключению интересующей подсхемы к первым $m$ кубитам системы.

Алгоритм, выполняемый квантовым компьютером, определяется схемой квантовых вентилей, подключенных к его кубитам и выполняющих соответствующие унитарные преобразования. Перед выполнением схемы компьютер «подготавливается», т.е. приводится к заданному начальному состоянию $\psi_0$. Вентили, срабатывающие в процессе выполнения программы, последовательно преобразуют вектор текущего состояния:

$$\begin{aligned} |\psi_0\rangle &\mapsto |\psi_1\rangle = \hat{U}_1 |\psi_0\rangle , \\ |\psi_1\rangle &\mapsto |\psi_2\rangle = \hat{U}_2 |\psi_1\rangle , \\ &\cdots \\ |\psi_{L-1}\rangle &\mapsto |\psi_L\rangle = \hat{U}_L |\psi_{L-1}\rangle . \end{aligned} \tag{6.6}$$

Состояние, получаемое в результате выполнения всей программы, определяется унитарным оператором $\hat{U}$, равным произведению всех матриц преобразования в обратном порядке:

$$|\psi_L\rangle = \hat{U}_L \cdots \hat{U}_2 \hat{U}_1 |\psi_0\rangle = \hat{U} |\psi_0\rangle ,$$
$$\hat{U} = \hat{U}_L \cdots \hat{U}_2 \hat{U}_1 .$$

Полученное на финальном этапе состояние $|\psi_L\rangle$ подлежит измерению, в результате которого система в общем случае снова меняет состояние. Однако такое преобразование уже не является унитарным.

### 6.1.5. Измерение результата вычислений

Получение результатов квантовых вычислений производится с помощью так называемого измерения (measurement). Напомним, что система из $n$ кубитов пребывает в общем случае в суперпозиции $2^n$ базисных состояний:

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_{2^n-1} |2^n - 1\rangle ,$$
$$|\alpha_0|^2 + |\alpha_1|^2 + \cdots + |\alpha_{2^n-1}|^2 = 1.$$

При выполнении полного измерения состояния системы может быть получено одно из $2^n$ числовых значений, соответствующих совокупностям базисных состояний отдельных кубитов. Любое возможное число $k = 0, \ldots, 2^n - 1$ будет получено с вероятностью, равной $|\alpha_k|^2$. При этом система перейдет в соответствующее базисное состояние $|k\rangle$:

$$|\psi\rangle \mapsto \begin{cases} |0\rangle, & \Pr = |\alpha_0|^2; \\ |1\rangle, & \Pr = |\alpha_1|^2; \\ \ldots & \\ |2^n - 1\rangle, & \Pr = |\alpha_{2^n-1}|^2. \end{cases} \tag{6.7}$$

Таким образом, первоначальное состояние $|\psi\rangle$ в общем случае (если оно не совпадает с одним из базисных состояний) будет потеряно. Все последующие измерения дадут тот же результат с полной вероятностью.

В более общем случае, помимо полного измерения, возможно также выполнение частичного измерения (partial measurement), т.е. измерения лишь части кубитов всей системы. Допустим, в системе из $n$ кубитов требуется измерить лишь $m$ из них. Если принять за измерительный базис набор векторов $|k\rangle, k = 0, \ldots, 2^m - 1$, состояние системы может быть представлено как суперпозиция разложимых состояний:

$$|\psi\rangle = \alpha_0 |\psi_0\rangle \otimes |0\rangle + \alpha_1 |\psi_1\rangle \otimes |1\rangle + \cdots + \alpha_{2^m-1} |\psi_{2^m-1}\rangle \otimes |2^m - 1\rangle,$$
$$|\alpha_0|^2 + |\alpha_1|^2 + \cdots + |\alpha_{2^m-1}|^2 = 1,$$
$$\langle\psi_k|\psi_k\rangle = 1, \quad k = 0, \ldots, 2^m - 1.$$

Нормированные $2^{n-m}$-мерные векторы $|\psi_0\rangle, \ldots, |\psi_{2^m-1}\rangle$ отражают возможные альтернативные состояния подсистемы из $n-m$ кубитов, которые не подлежат измерению. Тогда при измерении система переходит в одно из таких разложимых состояний:

$$|\psi\rangle \mapsto \begin{cases} |\psi_0\rangle \otimes |0\rangle, & \Pr = |\alpha_0|^2; \\ |\psi_1\rangle \otimes |1\rangle, & \Pr = |\alpha_1|^2; \\ \ldots & \\ |\psi_{2^m-1}\rangle \otimes |2^m - 1\rangle, & \Pr = |\alpha_{2^m-1}|^2. \end{cases}$$

В случае, если с точностью до множителя векторы $|\psi_0\rangle, \ldots, |\psi_{2^m-1}\rangle$ равны между собой, исходное состояние $|\psi\rangle$ является разложимым, т.е. состояние не подлежащей измерению подсистемы до измерения является чистым, а не смешанным, и при измерении не меняется.

В общем случае возможно выполнение измерений в произвольном ортонормированном базисе. Это эквивалентно выполнению унитарного преобразования над состоянием системы, осуществляющего поворот от желаемого измерительного базиса к стандартному вычислительному, с последующим измерением в вычислительном базисе.

Измерительные приборы, особенно те, результаты которых воздействуют на последующие квантовые вычисления, часто изображают на схемах наряду с вентилями и схемами. Однако следует их отличать от обычных квантовых вентилей и схем по той причине, что действия измерителей не являются обратимыми.

Известен так называемый принцип отложенного измерения (principle of deferred measurement) [55]. В соответствии с ним всякая квантовая схема с промежуточными измерениями может быть преобразована к эквивалентной квантовой схеме с измерениями лишь в конце вычислительного процесса. Это касается даже случаев, когда последующие части схемы строятся на основе результата промежуточных измерений (рис. 6.15).

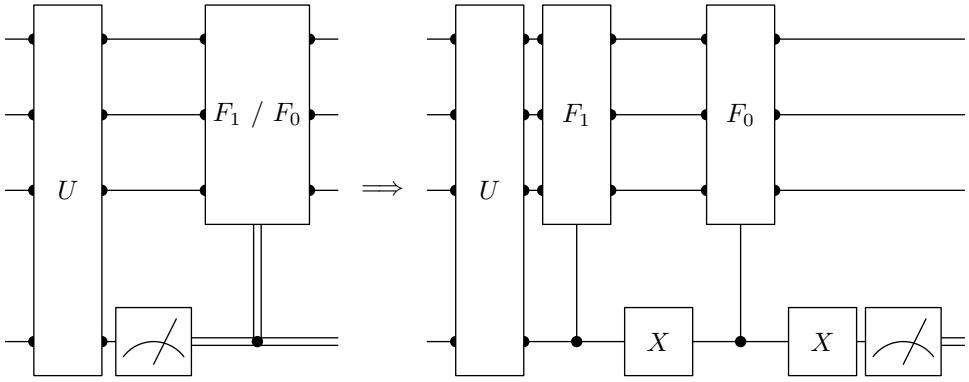



Рис. 6.15. Преобразование схемы с промежуточным измерением к эквивалентной схеме с отложенным измерением

К примеру, слева на рис. 6.15 изображена схема, на которой выполняемое преобразование зависит от результата промежуточного измерения. Допустим, в результате выполнения схемы $\hat{U}$ общее состояние системы представляется вектором $|\psi\rangle$:

$$|\psi\rangle = \alpha_0 |0\psi_0\rangle + \alpha_1 |1\psi_1\rangle = \alpha_0 |\psi_0\rangle \otimes |0\rangle + \alpha_1 |\psi_1\rangle \otimes |1\rangle .$$

Здесь $|0\rangle$ и $|1\rangle$ — состояния нижнего (управляющего) кубита, $|\psi_0\rangle$ и $|\psi_1\rangle$ — альтернативные состояния верхней подсистемы кубитов. В зависимости от значения классического бита, полученного в результате измерения управляющего кубита, дальнейшие вычисления выполняются по двум разным веткам, в рамках которых над остальными кубитами выполняется преобразование одной из схем $\hat{F}_0$ или $\hat{F}_1$:

$$|\psi\rangle \quad \mapsto \quad \begin{cases} |\psi_0\rangle \otimes |0\rangle \mapsto \hat{F}_0 |\psi_0\rangle \otimes |0\rangle , & \Pr = |\alpha_0|^2; \\ |\psi_1\rangle \otimes |1\rangle \mapsto \hat{F}_1 |\psi_1\rangle \otimes |1\rangle , & \Pr = |\alpha_1|^2. \end{cases}$$

Тогда эквивалентная схема (рис. 6.15, справа) может быть построена на основе двух управляемых схем и двух вентилей NOT. Первая управляемая схема выполняется, если измеряемый кубит в состоянии $|1\rangle$. Вторая должна выполняться в противном случае, в связи с чем ее управляющий вход временно инвертируется с помощью двух отрицаний. Последовательность преобразований, выполняемых такой схемой, выглядит следующим образом:

$$\begin{gathered} |\psi\rangle = \alpha_0 |\psi_0\rangle \otimes |0\rangle + \alpha_1 |\psi_1\rangle \otimes |1\rangle \quad \mapsto \\ \mapsto \quad \alpha_0 |\psi_0\rangle \otimes |0\rangle + \alpha_1 \hat{F}_1 |\psi_1\rangle \otimes |1\rangle \quad \mapsto \quad \alpha_0 |\psi_0\rangle \otimes |1\rangle + \alpha_1 \hat{F}_1 |\psi_1\rangle \otimes |0\rangle \quad \mapsto \\ \mapsto \quad \alpha_0 \hat{F}_0 |\psi_0\rangle \otimes |1\rangle + \alpha_1 \hat{F}_1 |\psi_1\rangle \otimes |0\rangle \quad \mapsto \quad \alpha_0 \hat{F}_0 |\psi_0\rangle \otimes |0\rangle + \alpha_1 \hat{F}_1 |\psi_1\rangle \otimes |1\rangle \quad \mapsto \\ \mapsto \quad \begin{cases} \hat{F}_0 |\psi_0\rangle \otimes |0\rangle , & \Pr = |\alpha_0|^2; \\ \hat{F}_1 |\psi_1\rangle \otimes |1\rangle , & \Pr = |\alpha_1|^2. \end{cases} \end{gathered}$$

Видно, что результат в обоих случаях совпадает. Второй вентиль NOT нужен для того, чтобы вернуть значение кубита в первоначальное состояние, хотя для приведенного случая оно роли в квантовой схеме уже не играет и влияет лишь на результат измерения.

Помимо получения результатов вычислений, измерения могут использоваться также для промежуточного контроля системы. К примеру, временные регистры после очистки могут быть измерены, чтобы убедиться, что они пребывают в обнуленном состоянии. Это позволяет обнаружить наличие вычислительных ошибок, вызванных необратимым воздействием внешней среды (проблема декогеренции).

### 6.1.6. Параллелизм в квантовых вычислениях

Выполнение вычислений на квантовом компьютере осуществляется в три этапа:

1) подготовка первоначального состояния системы кубитов;

2) выполнение последовательности унитарных преобразований путем применения схемы квантовых вентилей;

3) измерение состояния системы в вычислительном базисе и интерпретация результата.

Сама по себе такая схема не предлагает ничего нового по сравнению с классическими компьютерами. Преимущество становится понятным, если вспомнить, что состояние системы кубитов может быть суперпозицией базисных состояний. Таким образом, если на начальном этапе вычислений состояние системы будет переведено в суперпозицию исходных данных, дальнейшие вычисления будут производиться для всего полученного набора данных параллельно. При этом количество параллельных ветвей вычислений зависит от количества входных кубитов экспоненциально. Это явление носит название квантового параллелизма.

Однако преимущества, получаемые от квантового параллелизма, оказываются в некотором роде скрытыми от нас, поскольку, хоть система и получает суперпозицию результатов вычислений, т.е. множество значений, при измерении мы все равно получим лишь одно из них. Может возникнуть предложение скопировать состояние системы во временный регистр и измерить его состояние, после чего снова скопировать в него исходное состояние и т.д., получив таким образом множество значений. Однако существует теорема о невозможности клонирования состояния (no-cloning theorem), в соответствии с которой такое действие выполнить невозможно, что следует из унитарности выполняемых преобразований. Скопировано может быть классическое состояние, квантовое же состояние можно лишь переместить из одной подсистемы в другую (телепортация). Таким образом, суперпозиция результатов все равно будет потеряна при первом же измерении.

Кажется, что упомянутый параллелизм, позволяющий вычислить некоторую функцию от множества аргументов, оказывается для нас бесполезен. Но это не так. Во многих случаях нас в конечном итоге интересует не вся последовательность значений функции, а лишь какое-то ее свойство или фиксированный их набор, который мы могли бы узнать, имея эту последовательность. К примеру, это может быть период функции (алгоритм Шора [77]) или же величина аргумента, для которого функция принимает заданное значение (алгоритм Гровера [63]). Тогда на основе вычисленной суперпозиции значений функции путем последующих квантовых преобразований зачастую оказывается возможным сформировать состояние, из которого легко получить требуемое свойство.

Таким образом, обычная схема использования квантового параллелизма принимает следующий общий вид (рис. 6.16). Вначале формируется состояние, представляемое суперпозицией аргументов исследуемой функции. Как правило, для этого используется преобразование Уолша-Адамара. Затем над этим состоянием единожды выполняется преобразование, реализующее интересующую нас функцию. На выходе получается состояние, содержащее суперпозицию ее значений для различных аргументов. Наконец, последний этап —

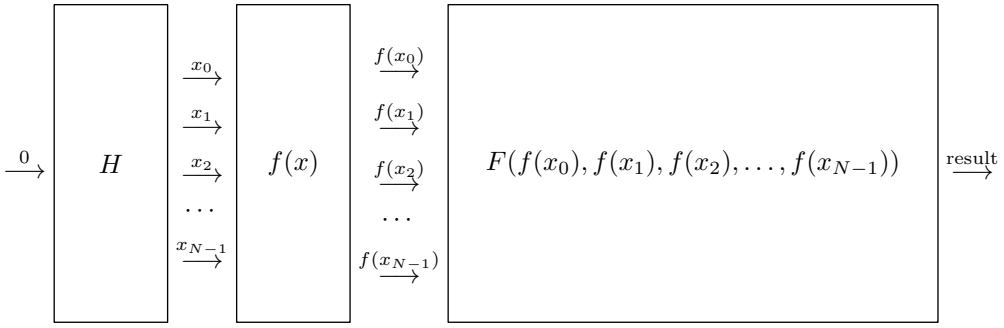



Рис. 6.16. Использование параллелизма в квантовых вычислениях

выполнение квантового преобразования, формирующего из суперпозиции значений функции некоторое состояние, измерив которое, можно извлечь необходимую информацию.

Именно последний этап обычно представляется весьма непростым, поскольку далеко не во всех случаях удается построить эффективную схему такого преобразования. Однако известен ряд довольно сложных задач, для которых предложен эффективный квантовый алгоритм.

Простейшим примером, иллюстрирующим описанный подход, является алгоритм Дойча, который будет рассмотрен нами ниже. Несмотря на то, что решаемая им задача сама по себе выглядит весьма надуманно, сам механизм ее решения показывает принципиальную возможность использования квантового параллелизма, если искомая величина может быть получена на основе всего множества значений некоторой функции, тогда как сами ее значения прямого интереса не представляют.

## 6.2. Симулятор квантового компьютера

В дальнейшем для иллюстративной реализации квантовых алгоритмов нам потребуется классическая реализация симулятора квантового компьютера. Разумеется, вследствие экспоненциальной временной сложности работы такого симулятора размеры потенциально решаемых им задач оказываются смехотворными, и их решение не приносит практической пользы. Тем не менее, использование симуляторов, реализованных на классических компьютерах, является обычной практикой как в исследовательских целях (при проверке работоспособности квантовых алгоритмов), так и в образовательных (как в нашем случае). Симуляторов зачастую оказывается достаточно для исследования задач малых размеров. В частности, используемый нами в дальнейшем симулятор оказывается в состоянии решать задачи на регистрах длиной более двух десятков кубитов. Более того, при использовании симуляторов появляются дополнительные возможности, удобные при составлении и анализе квантовых схем, но не реализуемые на настоящих квантовых компьютерах. Это касается возможностей получения вектора квантового состояния без его изменения и построения матрицы преобразования.

Многие классические реализации квантовых алгоритмов, часто приводимые в образовательной литературе, иллюстрируют математические преобразования, выполняемые над вектором состояния, без явного использования квантовых схем. Описываемый же здесь симулятор, наоборот, основан на построении квантовой схемы и последующем ее выполнении, в связи с чем выполняемые математические преобразования оказываются не столь очевидными и требуют отдельного описания. Автор не берется утверждать, что такой подход объективно лучше или нагляднее, просто он оказался более удобным лично ему.

### 6.2.1. Виртуальный квантовый вычислитель

Код используемого нами в дальнейшем симулятора квантового компьютера приводится в приложении Е. Он содержит описание абстрактного класса произвольной квантовой схемы, а также набор классов, унаследованных от него.

Абстрактный класс `circuit_abstract_type` описывает произвольную квантовую схему, которой может быть как однокубитный вентиль, так и полноценный квантовый компьютер, содержащий сложный набор квантовых схем. В классе содержатся описания используемых типов, а также описания двух абстрактных функций. Функция `qubit_number` возвращает количество кубитов квантового регистра, охватываемого соответствующей схемой. Функция `execute` получает на входе вектор квантового состояния и преобразует его соответствующим схеме образом. Тип `qustate_type`, описывающий квантовое состояние, основан на приведенных в приложении А шаблонах классов и является вектором комплексных чисел. Длина вектора состояния для каждой схемы возвращается `inline`-функцией `qustate_size`, возводящей двойку в степень числа, возвращаемого функцией `qubit_number`.

Несколько последующих классов реализуют абстрактный класс `circuit_abstract_type`. Далее опишем их, но прежде приведем описание вспомогательного шаблона `holder_type`.

#### Вспомогательный класс `holder_type`

В классе составной схемы, описание которого приводится ниже, для хранения информации о содержащихся подсхемах используется стандартный контейнер. Поскольку хранимые подсхемы имеют в общем случае разный тип, приходится хранить объекты не по значению, а по указателю абстрактного класса. В связи с этим для содержания своей собственной копии подсхемы возникает необходимость ее явного динамического создания и уничтожения. В то же время, для удобства написания клиентского кода удобно наличие у класса составной схемы конструктора копирования и оператора присваивания. Для их реализации необходимо наличие возможности создания копий объектов на основе абстрактных указателей. Обычно такая возможность обеспечивается с помощью какой-либо виртуальной функции (к примеру, `clone`), выполняющей создание нового объекта на основе существующего. Однако этот подход требует реализации соответствующей функции в каждом классе квантовой схемы, что усложняет написание клиентского кода. Чтобы избежать такого усложнения при реализации классов квантовых схем, мы используем вместо этого шаблон статической функции, инстанцируемый на основе реального типа подсхемы.

Для хранения в контейнере объектов по указателям создан специальный класс `holder_type`. Он реализован в виде шаблона, поскольку не привязан явно к конкретному базовому абстрактному типу и, в принципе, может быть использован при реализации других задач. В задачи класса `holder_type` входит хранение некоторого объекта по указателю на базовый тип и обеспечение корректного копирования объекта и его уничтожения в соответствии с его реальным типом. Ссылка на хранимый объект в любой момент может быть получена с помощью функции `ref`.

В момент создания объекта типа `holder_type` на основе некоторого объекта, унаследованного от указанного базового типа, выполняется получение адреса экземпляра шаблонной статической функции `borndie`, инстанцированного на основе реального типа переданного объекта. Эта функция в зависимости от переданного параметра выполняет одну из двух операций: создание нового объекта с вызовом конструктора копирования или уничтожение объекта. Логичнее было бы разнести обе эти операции в разные статические функции, а не совмещать в одну, однако тогда потребовалось бы в объекте `holder_type` хранить, помимо указателя на объект, два указателя на статические функции. В нашем

же случае приходится хранить лишь один, что сокращает расход памяти. Использование статической функции не только для создания, но и для уничтожения, избавляет от необходимости наличия в базовом классе виртуального деструктора.

С помощью функции `borndie` с параметром `BORN` осуществляется копирование объекта во всех конструкторах `holder_type`, а также в операторе присваивания. Соответственно, в деструкторе `holder_type`, а также в операторе присваивания перед копированием нового объекта, выполняется уничтожение текущего объекта путем вызова `borndie` с параметром `DIE`.

При помещении в контейнер объекта типа `holder_type`, владеющего некоторым объектом подсхемы, вызывается конструктор копирования `holder_type`, который, в свою очередь, опосредованно вызывает конструктор копирования реального типа содержащегося объекта. При очистке контейнера вызываются деструкторы объектов `holder_type`, в рамках которых выполняется уничтожение соответствующих объектов подсхем. Это позволяет нам не заботиться о копировании и удалении объектов подсхем, хранимых в контейнерах, несмотря на то, что они имеют в общем случае разный тип и хранятся по указателям.

### Однокубитный вентиль

Самой простой реализацией класса абстрактной схемы является класс `gate_unary_type`, представляющий произвольный однокубитный вентиль. Объект такого класса хранит матрицу выполняемого преобразования размерности $2 \times 2$, на которую умножает вектор преобразуемого состояния в функции `execute`.

Матрица преобразования заполняется наследующими классами с помощью функции `mapsto`. Эта функция реализована для удобства задания выполняемого преобразования в виде $|k\rangle \mapsto \alpha_0 |0\rangle + \alpha_1 |1\rangle, k = 0, 1$. Каждый наследующий класс должен вызвать ее в конструкторе дважды — для обоих базисных состояний. Примеры будут приведены ниже.

### Управляемая схема

Следующим классом, унаследованным от абстрактной квантовой схемы, является `circuit_controlled_type`. Он реализует работу некоторой схемы над произвольным количеством кубитов, управляемой еще одним кубитом, внешним по отношению к ней. Объект класса `circuit_controlled_type` хранит внутри себя объект схемы, которая подлежит управлению внешним кубитом.

Подлежащая управлению схема имеет произвольный тип (унаследованный от `circuit_abstract_type`), что осложняет описание соответствующего объекта. Можно реализовать класс `circuit_controlled_type` в виде шаблона, инстанцируемого на основе типа подлежащей управлению схемы. Однако мы можем себе позволить другой способ, поскольку нами уже реализован специальный класс `holder_type` для хранения указателя на произвольную подсхему. Этот способ несколько удобнее тем, что незначительно сокращается клиентский код за счет отсутствия указания типа подлежащей управлению схемы при инстанцировании шаблона. При этом от нас не требуется явное создание и уничтожение собственной копии схемы в рамках описания класса `circuit_controlled_type`.

Управляемая схема занимает на один кубит больше, чем соответствующая подлежащая управлению подсхема, что отражено в реализации функции `qubit_number`. Преобразование квантового состояния, выполняемое функцией `execute`, представлено выражением (6.5). В соответствии с ним, изменению подлежат амплитуды лишь тех базисных состояний, у которых последний кубит установлен в единицу. Им соответствует вторая половина входного вектора состояния, первая же остается без изменений. При извлечении вектора состояния подсистемы для преобразования подлежащей управлению схемой используется алгоритм

`std::copy`. Возможность его использования обусловлена тем, что, в соответствии с заявленными требованиями к реализации класса `vector_type`, содержимое вектора состояния хранится в непрерывном массиве.

### Составная квантовая схема

Составная схема представлена классом `circuit_type`. Она призвана объединять в едином объекте произвольный набор внутренних по отношению к ней схем и вентилей (подсхем).

Объект составной схемы содержит переменную-член, отражающую количество кубитов, к которым подключается схема, а также контейнер, содержащий последовательность выполняющихся внутри схемы подсхем. В рамках этого контейнера каждый элемент представлен совокупностью из объекта типа `holder_type`, содержащего соответствующий объект подсхемы, а также двух наборов битовых масок, о которых подробнее скажем позже. Использование в контейнере типа `holder_type` позволяет хранить в нем объекты разных типов и, в то же время, не заботиться об их корректном копировании и уничтожении.

Функция `execute` составной схемы выполняет преобразование над полным вектором состояния путем последовательного применения всего набора внутренних подсхем. С помощью каждой подсхемы состояние преобразуется в соответствии с выражением (6.4). Для этого входной вектор длиной $2^n$ разбивается на $2^{n-m}$ подвекторов длиной $2^m$, каждый из которых преобразуется подсхемой отдельно. Разбиение вектора выполняется в соответствии с номерами кубитов, к которым подключена подсхема. Тут возникает вопрос, каким именно образом осуществлять разбиение, поскольку выражение (6.4) иллюстрирует случай, когда подсхема подключена к нескольким первым кубитам системы, а остальные не затронуты. В общем же случае подсхема может быть подключена к любым кубитам системы в произвольном порядке. В связи с этим используются два набора битовых масок, один из которых отображает $2^m$ базисных состояний системы подключенных к подсхеме кубитов на соответствующие битовые маски базисного состояния всей составной схемы, другой выполняет аналогичное отображение для $2^{n-m}$ базисных состояний остального набора кубитов, не подключенных к подсхеме. Полное базисное состояние всего набора кубитов составной схемы формируется путем объединения битовых масок для соответствующих базисных состояний обеих подсистем. По сути, с помощью такого механизма осуществляется перестановка «на лету» элементов вектора состояния путем обращения сразу к нужным его элементам. Оба набора битовых масок строятся однократно при добавлении подсхемы и используются многократно при выполнении.

Добавление в схему некоторой подсхемы производится шаблонной функцией `add_circuit`. На входе передается объект добавляемой подсхемы, а также набор номеров кубитов в рамках текущей составной схемы, к которым будет подключаться соответствующая подсхема. На основе переданного набора и полного набора номеров кубитов составной схемы формируется набор номеров «остальных» кубитов, которые подсхемой не задействуются. Для каждого из двух взаимно дополняющих наборов с помощью битовых операций формируется отображение базисных состояний соответствующей подсистемы кубитов в битовую маску базисного состояния всей системы. Наконец, оба построенных отображения, вместе с копией объекта переданной схемы, помещаются в контейнер `m_circuits`.

Для повышения удобства при написании клиентского кода дополнительно введены функции `add_unary`, `add_binary` и `add_ternary`, которые позволяют добавлять в схему соответственно унарные, бинарные и тернарные вентили, минуя явное создание списка номеров кубитов.

Полная очистка схемы может быть выполнена с помощью функции `remove_all`, кото-

рая осуществляет очистку контейнера `m_circuits`. В этом случае в деструкторах соответствующих объектов `holder_type` уничтожаются подключенные к составной схеме объекты подсхем.

#### Квантовый компьютер

Квантовый компьютер является частным случаем составной схемы, вследствие чего класс `quantum_machine_type` унаследован от `circuit_type` и позволяет вызов функций составления квантовой схемы. Помимо схемы квантовых вентилей, объект квантового компьютера хранит вектор текущего состояния.

Функции `prepare`, `run` и `measure` осуществляют выполнение основных операций с квантовым компьютером: подготовку состояния, выполнение заложенной квантовой схемы и измерение результата. Подготовка состояния производится функцией `prepare` в соответствии с (6.3). Функция `run` меняет это состояние путем последовательного применения преобразований (6.6) в соответствии с заложенной схемой вентилей. Наконец, функция `measure` на основе датчика случайных чисел осуществляет выбор произвольного базисного состояния (с учетом его вероятности), после чего в соответствии с (6.7) переводит систему в это состояние и возвращает соответствующее значение. Поскольку принцип отложенного измерения позволяет обойтись без промежуточных измерений, для упрощения кода симулятора реализовано лишь полное измерение. Частичное измерение при желании может быть реализовано дополнительно.

Помимо естественных функций квантового компьютера, класс `quantum_machine_type` поддерживает еще две, возможность наличия которых характерна лишь для симуляторов. Функция `qustate` позволяет получить весь вектор текущего состояния квантовой системы, не изменяя его. Функция `build_matrix` возвращает матрицу преобразования, выполняемого текущей заложенной в квантовый компьютер схемой. Эта матрица строится путем последовательного выполнения преобразования над всеми возможными базисными состояниями квантовой системы.

### 6.2.2. Реализация базовых вентилей

Приведем примеры реализации вентилей, которые будут использованы нами в дальнейшем при выполнении вычислений в описанном симуляторе. Одним из основных квантовых вентилей, используемых практически во всех вычислениях, является вентиль Адамара:

```
// вентиль HADAMARD
class gate_hadamard_type: public gate_unary_type
{
public:
 gate_hadamard_type(void)
 {
  mapsto(0, M_SQRT1_2, M_SQRT1_2);
  mapsto(1, M_SQRT1_2, - M_SQRT1_2);
 }
};
```

Зачастую вентили Адамара устанавливаются сразу в большом количестве, поэтому для удобства реализована схема из их последовательности, выполняющая преобразование Уолша-Адамара над произвольным количеством кубитов:

```
// n-кубитная схема HADAMARD
class circuit_hadamard_type: public circuit_type
{
```

```
public:
 circuit_hadamard_type(int n):
  circuit_type(n)
 {
  for (int i = 0; i < qubit_number(); ++i)
   add_unary(gate_hadamard_type(), i);
 }
};
```

Аналогично вентилю Адамара строится вентиль фазового сдвига:

```
// вентиль PHASE SHIFT
// аргумент – часть полного оборота (поворот на frac * 2 * pi)
class gate_shift_type: public gate_unary_type
{
public:
 gate_shift_type(real_type frac)
 {
  mapsto(0, 1.0, 0.0);
  mapsto(1, 0.0, exp(element_type(0.0, 2.0) * M_PI * frac));
 }
};
```

Для реализации классических обратимых вентилей потребуется вентиль NOT:

```
// вентиль NOT
class gate_not_type: public gate_unary_type
{
public:
 gate_not_type(void)
 {
  mapsto(0, 0.0, 1.0);
  mapsto(1, 1.0, 0.0);
 }
};
```

С его использованием на основе класса управляемой схемы легко реализуется вентиль CNOT:

```
// двухкубитный вентиль CNOT
class gate_cnot_type: public circuit_controlled_type
{
public:
 gate_cnot_type(void):
  circuit_controlled_type(gate_not_type())
 {}
 gate_cnot_type(const gate_cnot_type &src):
  circuit_controlled_type(
   static_cast<const circuit_controlled_type &>(src))
 {}
};
```

На базе вентиля CNOT могут быть построены вентиль SWAP и вентиль Тоффоли:

```
// двухкубитный вентиль SWAP
class gate_swap_type: public circuit_type
{
public:
```

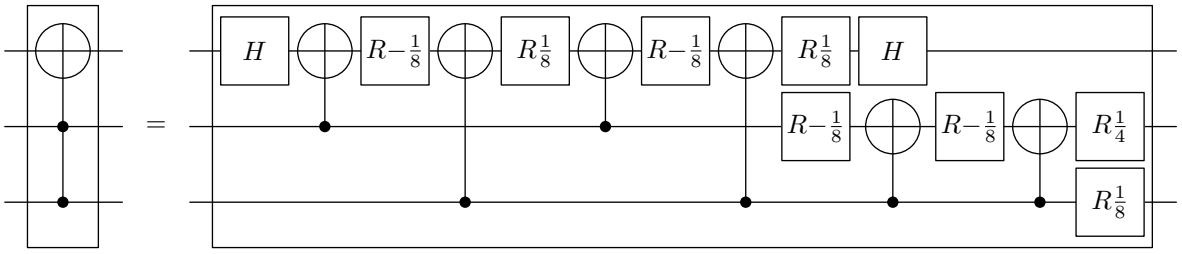


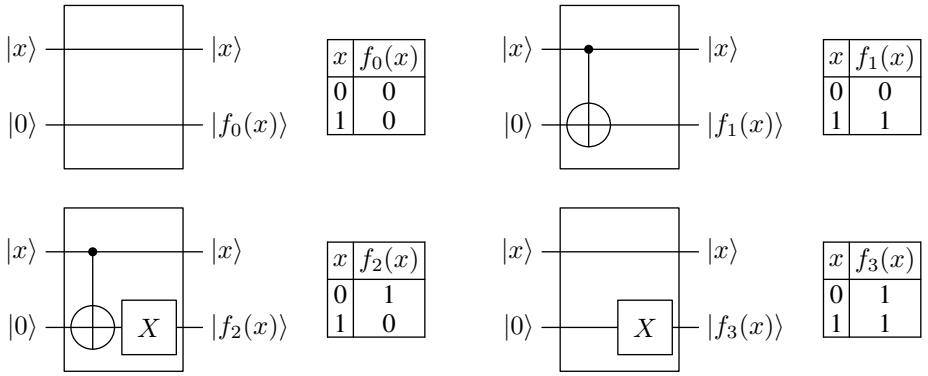

Рис. 6.17. Вентиль Тоффоли, реализованный на однокубитных и двухкубитных вентилях

```
 gate_swap_type(void):
  circuit_type(2)
 {
  add_binary(gate_cnot_type(), 0, 1);
  add_binary(gate_cnot_type(), 1, 0);
  add_binary(gate_cnot_type(), 0, 1);
 }
};

// трехкубитный вентиль TOFFOLI
class gate_toffoli_type: public circuit_controlled_type
{
public:
 gate_toffoli_type(void):
  circuit_controlled_type(gate_cnot_type())
 {}
 gate_toffoli_type(const gate_toffoli_type &src):
  circuit_controlled_type(
   static_cast<const circuit_controlled_type &>(src))
 {}
};
```

С использованием однокубитных вентилей Адамара и фазового сдвига, а также двухкубитного вентиля CNOT, может быть построена альтернативная реализация вентиля Тоффоли [55] (рис. 6.17). На схеме вентили сдвига фазы обозначены дробной частью полного оборота: $\hat{R}_\gamma = \hat{R}(2\pi \cdot \gamma)$.

```
// трехкубитный вентиль TOFFOLI - альтернативная реализация
class gate_toffoli_type: public circuit_type
{
public:
 gate_toffoli_type(void):
  circuit_type(3)
 {
  add_unary(gate_hadamard_type(), 0);
  add_binary(gate_cnot_type(), 0, 1);
  add_unary(gate_shift_type(0.875), 0);
  add_binary(gate_cnot_type(), 0, 2);
  add_unary(gate_shift_type(0.125), 0);
  add_binary(gate_cnot_type(), 0, 1);
  add_unary(gate_shift_type(0.875), 0);
  add_binary(gate_cnot_type(), 0, 2);
  add_unary(gate_shift_type(0.125), 0);
```




| $x$ | $f_0(x)$ |
|---|---|
| 0 | 0 |
| 1 | 0 |

| $x$ | $f_1(x)$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $x$ | $f_2(x)$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

| $x$ | $f_3(x)$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

Рис. 6.18. Варианты логической функции одного аргумента

```
  add_unary(gate_hadamard_type(), 0);
  add_unary(gate_shift_type(0.875), 1);
  add_binary(gate_cnot_type(), 1, 2);
  add_unary(gate_shift_type(0.875), 1);
  add_binary(gate_cnot_type(), 1, 2);
  add_unary(gate_shift_type(0.250), 1);
  add_unary(gate_shift_type(0.125), 2);
 }
};
```

По функциональности эта схема аналогична управляемому CNOT (в чем можно убедиться путем построения соответствующей матрицы преобразования с помощью функции `build_matrix`), хотя при использовании в симуляторе она гораздо менее производительна.

## 6.3. Алгоритм Дойча

В качестве простого примера алгоритма, использующего квантовый параллелизм, рассмотрим алгоритм Дойча (David Deutsch) [60, 69, 6]. Предположим, дана некая классическая функция от одного бита $f(x)$ и реализующая ее обратимая схема: $|x\rangle\,|y\rangle \mapsto |x\rangle\,|y \oplus f(x)\rangle$. Все четыре возможных варианта такой схемы изображены на рис. 6.18. Из этих вариантов два являются функциями, генерирующими константу, еще два порождают сбалансированное количество нулей и единиц для всего набора входных данных. В зависимости от этого каждую такую функцию можно назвать либо постоянной, либо сбалансированной.

Формулировка задачи Дойча такова, что при наличии некоторой произвольно взятой схемы из приведенного набора требуется узнать, реализует ли она постоянную или сбалансированную функцию. Классическое решение этой задачи требует двух вычислений значения функции. В квантовой реализации нужно лишь одно вычисление сразу для обоих вариантов входного значения (рис. 6.19).

В начале работы схемы путем применения вентиля Адамара формируется суперпозиция двух возможных входных значений. Далее полученная суперпозиция подвергается преобразованию, в рамках которого выполняется обращение знака у базисного состояния, соответствующего единичному значению исследуемой функции: $|x\rangle \mapsto (-1)^{f(x)}\,|x\rangle$ [55]. Наконец, последний вентиль Адамара возвращает суперпозицию к одному из состояний вычислительного базиса:

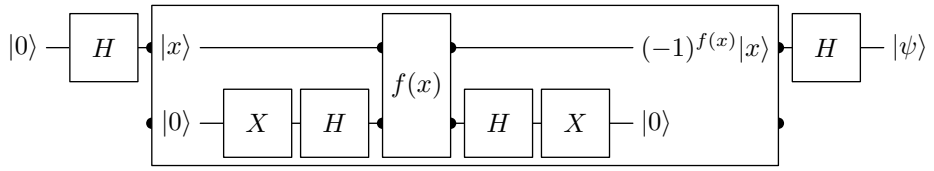



Рис. 6.19. Квантовая схема, реализующая алгоритм Дойча

$$|0\rangle \quad \mapsto \quad \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \mapsto \quad \frac{1}{\sqrt{2}}\left((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\right) \quad \mapsto$$
$$\mapsto \quad |\psi\rangle = \frac{1}{2}\left((-1)^{f(0)}(|0\rangle + |1\rangle) + (-1)^{f(1)}(|0\rangle - |1\rangle)\right).$$

Подставляя в результат выходные значения различных вариантов функции $f(x)$, видим, что состояние на выходе равно:

$$|\psi\rangle = \begin{cases} |0\rangle, & f(0) = 0, f(1) = 0; \\ |1\rangle, & f(0) = 0, f(1) = 1; \\ -|1\rangle, & f(0) = 1, f(1) = 0; \\ -|0\rangle, & f(0) = 1, f(1) = 1. \end{cases}$$

Проводя измерение в стандартном базисе, получаем 0, если значения функции при обоих аргументах одинаковы (функция постоянна), и 1 в противном случае (функция сбалансирована). В рамках выполнения схемы было выполнено три основных этапа, что соответствует обычному подходу к реализации квантовых алгоритмов (рис. 6.16), при этом было произведено лишь одно вычисление функции сразу для обоих аргументов.

Рассмотрим детально устройство схемы, выполняющей обращение знака у амплитуды состояния в зависимости от соответствующего значения функции (рис. 6.19). Такая схема использует один дополнительный временный кубит и выполняет над входным состоянием $|x\rangle|0\rangle$ последовательно следующие преобразования:

$$|x\rangle|0\rangle \quad \mapsto \quad |x\rangle|1\rangle \quad \mapsto \quad |x\rangle\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) \quad \mapsto$$
$$\mapsto \quad (-1)^{f(x)}|x\rangle\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right) \quad \mapsto \quad (-1)^{f(x)}|x\rangle|1\rangle \quad \mapsto \quad (-1)^{f(x)}|x\rangle|0\rangle.$$

Может возникнуть вопрос, как путем преобразования $|x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle$ получено изменение знака. Поясним это, раскрыв результат воздействия такого преобразования на состояние $1/\sqrt{2}\,|x\rangle(|0\rangle - |1\rangle)$ для каждого из двух различных значений $f(x)$ независимо от значения $x$:

$$|x\rangle\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} = |x\rangle\frac{|f(x)\rangle - \left|\overline{f(x)}\right\rangle}{\sqrt{2}} =$$
$$= \frac{1}{\sqrt{2}}\begin{cases} |x\rangle(|0\rangle - |1\rangle) \equiv (-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle), & f(x) = 0; \\ |x\rangle(|1\rangle - |0\rangle) \equiv (-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle), & f(x) = 1. \end{cases}$$

Обычно при описании алгоритма Дойча приводится другая схема, более компактная по сравнению с рис. 6.19. Она сокращена за счет того, что временный бит, во-первых, подается уже установленным в единицу, а во-вторых, не очищается после использования. По мнению автора настоящего издания, приведенный способ изображения схемы с разбиением по отдельным модулям более нагляден, поскольку позволяет лучше понять, с какой целью выполняются те или иные преобразования и вследствие чего получен нужный результат.

Приведем код, реализующий описанный алгоритм с использованием построенного ранее симулятора. Прежде всего, требуется описание схемы, реализующей исследуемую функцию (рис. 6.18):

```
// схема классической однобитной функции (1 из 4 вариантов)
// |x> |0>  —>  |x> |f(x)>
class func_type: public circuit_type
{
public:
 enum type {
  CONSTANT00,
  BALANCED01,
  BALANCED10,
  CONSTANT11
 };
 func_type(type t):
  circuit_type(2)
 {
  if (t == BALANCED01 || t == BALANCED10)
   add_binary(gate_cnot_type(), 1, 0);
  if (t == BALANCED10 || t == CONSTANT11)
   add_unary(gate_not_type(), 1);
 }
};
```

Приведенный класс реализует одну из четырех однобитных функций в зависимости от переданного параметра (рис. 6.18). На базе такой схемы реализуется схема обращения знака (рис. 6.19):

```
// схема изменения знака по значению функции
// |x> |0>  —>  (-1)^f(x) |x> |0>
class flipsign_type: public circuit_type
{
public:
 flipsign_type(func_type::type t):
  circuit_type(2)
 {
  add_unary(gate_not_type(), 1);
  add_unary(gate_hadamard_type(), 1);
  add_binary(func_type(t), 0, 1);
  add_unary(gate_hadamard_type(), 1);
  add_unary(gate_not_type(), 1);
 }
};
```

Наконец, для выполнения решения задачи строится двухкубитный компьютер, формируется схема алгоритма Дойча и производится вычисление:

```
// исследуемая функция
func_type::type functype = func_type::/*...*/;
```

```
// двухкубитный квантовый компьютер
quantum_machine_type qm(2);
// схема решения задачи Дойча
qm.add_unary(gate_hadamard_type(), 0);
qm.add_binary(flipsign_type(functype), 0, 1);
qm.add_unary(gate_hadamard_type(), 0);

// подготовка начального состояния компьютера
qm.prepare(0);
// выполнение схемы
qm.run();
// измерение результата
int result = qm.measure();
// интерпретация результата
bool isbalanced = (result != 0);
```

Квантовый компьютер содержит два кубита, поскольку включает в том числе временный кубит, необходимый для работы вложенной схемы. Оба кубита первоначально задаются равными нулю. В измеренном результате также содержатся оба бита, но, поскольку временный уже зачищен, представляет интерес лишь один из них.

## 6.4. Полная реализация алгоритма Шора

В качестве полноценного примера алгоритма решения практически значимой задачи на квантовом компьютере мы рассмотрим алгоритм факторизации, т.е. разложения составного числа на простые множители, предложенный Питером Шором (Peter Williston Shor) [77]. Это далеко не единственный алгоритм, разработанный специально для квантового компьютера, однако именно он зачастую оказывается наиболее удобным для демонстрации потенциальной мощи квантовых компьютеров, поскольку сложность решаемой этим алгоритмом задачи известна сегодня достаточно широко и является основой криптостойкости одного из наиболее популярных алгоритмов асимметричного шифрования RSA.

Лучшие существующие классические алгоритмы решают задачу факторизации за экспоненциальное время относительно длины факторизуемого числа, тогда как временная сложность алгоритма Шора на квантовом компьютере является полиномиальной.

### 6.4.1. Общая схема и описание

Задача разложения некоторого составного числа $N$ на простые множители может быть сведена к подзадаче поиска любого нетривиального множителя, т.е. такого, который не равен $N$ или единице. Для поиска такого множителя по алгоритму Шора производится выбор некоторого произвольного числа $C$, взаимно простого с $N$, и вычисление его порядка по модулю $N$, т.е. наименьшего целого числа $r$, такого, что:

$$C^r \equiv 1 \pmod{N}.$$

Значение $r$ является периодом функции $f(x) = C^x \bmod N$. Если найденное число $r$ оказалось четным, приведенное сравнение может быть записано следующим образом:

$$(C^{\frac{r}{2}} - 1)(C^{\frac{r}{2}} + 1) \equiv 0 \pmod{N}.$$

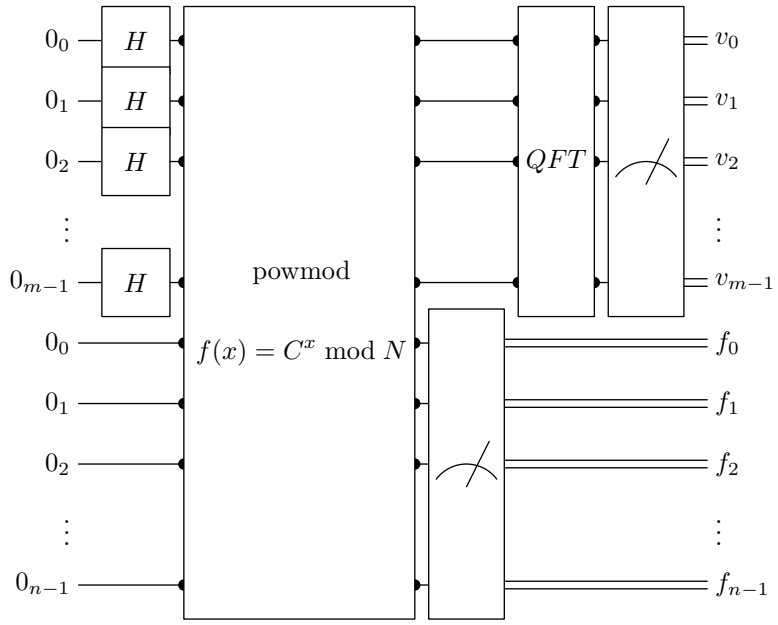



Рис. 6.20. Квантовая схема, используемая в алгоритме Шора

Выражение в левой части произведения не кратно $N$, поскольку иначе порядком являлось бы значение $r/2$. Если же и выражение в правой его части оказалось не кратно $N$, каждый из двух множителей кратен некоторому нетривиальному делителю $N$, поскольку числу $N$ кратно все произведение. В таком случае эти делители могут быть получены путем вычисления наибольшего общего делителя (НОД) чисел $(C^{\frac{r}{2}} \pm 1)$ и $N$. Нам необходим лишь один из них, поскольку второй мы можем получить путем деления на него числа $N$. Достаточно вычислить НОД пары чисел $(C^{\frac{r}{2}} - 1)$ и $N$, проверить результат на равенство с единицей и, если оказался больше, он и будет являться искомым нетривиальным делителем.

В случае, если какие-либо допущенные условия не выполнились (значение порядка $r$ нечетно или числа $C^{\frac{r}{2}} - 1$ и $N$ взаимно просты), производится повторная попытка вычислений с новым произвольно выбранным числом $C$ [77].

Видно, что описанный алгоритм базируется на сведении задачи факторизации к задаче поиска порядка некоторого числа $C$ по модулю $N$. Эта задача является вычислительно не менее сложной, и именно она решается с помощью квантового компьютера. Для поиска порядка некоторого числа используется схема, изображенная на рис. 6.20.

Схема использует два регистра (помимо временных, не изображенных на рис. 6.20). Первый предназначен для задания аргументов функции $f(x) = C^x \bmod N$ (первые $m$ кубитов), второй — для помещения ее вычисленных значений (следующие $n$ кубитов). Длина второго регистра выбирается ровно такой, чтобы уместить все возможные $N$ значений функции, т.е. $n = \lceil \log_2 N \rceil$. Длина регистра аргументов $m$ выбирается исходя из соотношения $N^2 < 2^m < 2N^2$ и оказывается ровно или приблизительно вдвое больше $n$ (а именно $2n$ или $2n - 1$). Строго говоря, в исходном описании фигурировало соотношение $N^2 \leqslant 2^m < 2N^2$ [77], но там же сказано о корректности использования алгоритма лишь с нечетными $N$, не являющимися степенями простого числа, а значит $N^2$ не может быть степенью двойки. Двойная ширина аргумента нужна для обеспечения возможности использования механизма цепных дробей при извлечении значения порядка из результата

измерений [77]. Подробнее об этом будет сказано ниже.

Система начинает работать из состояния $|0\rangle\,|0\rangle$, т.е. когда кубиты обоих регистров обнулены. Первым делом выполняется преобразование Уолша-Адамара над первым регистром (операция сложности $O(m)$), в результате которого система переходит в суперпозицию аргументов $x$:

$$|0\rangle\,|0\rangle \quad \mapsto \quad \frac{1}{\sqrt{2^m}} \sum_{x=0}^{2^m-1} |x\rangle\,|0\rangle\,.$$

На данный момент система пребывает в суперпозиции состояний, охватывающей все возможные аргументы из заданного диапазона, при этом для каждого значения аргумента значение функции пока равно нулю. Эта ситуация меняется после выполнения следующей схемы, подключенной к обоим регистрам. Она соответствует классической обратимой схеме, выполняющей вычисление необратимой функции от аргумента, заданного в первом регистре (рис. 6.2). В нашем случае схема вычисляет значение функции $f(x)$ для каждого значения аргумента $x$, в результате чего система меняет состояние следующим образом:

$$\frac{1}{\sqrt{2^m}} \sum_{x=0}^{2^m-1} |x\rangle\,|0\rangle \quad \mapsto \quad \frac{1}{\sqrt{2^m}} \sum_{x=0}^{2^m-1} |x\rangle\,|C^x \bmod N\rangle\,.$$

Схема $f(x)$ в процессе выполнения использует еще два временных регистра длиной $n$ каждый. Они необходимы для хранения промежуточных данных в процессе умножения и сложения, о чем подробнее будет сказано ниже.

Для наглядности, рассмотрим случай, когда длина вектора состояния первого регистра кратна искомому периоду, т.е. $2^m/r$ — целое число. Поскольку $f(x)$ — периодическая функция с периодом $r$, каждое свое значение она принимает для всего множества аргументов $x = a+br, b = 0, \ldots, 2^m/r-1$ при $0 \leqslant a < r$. Тогда полученное на текущий момент состояние системы можно представить как суперпозицию разложимых состояний, сгруппированных по значениям функции:

$$\frac{1}{\sqrt{2^m}} \sum_{x=0}^{2^m-1} |x\rangle\,|C^x \bmod N\rangle = \frac{1}{\sqrt{2^m}} \sum_{a=0}^{r-1} \left( \sum_{b=0}^{2^m/r-1} |a+br\rangle \right) |C^a \bmod N\rangle\,. \tag{6.8}$$

Если в этот момент произвести измерение второго регистра (рис. 6.20), получим какое-либо конкретное значение функции $f(x)$. Первый регистр при этом перейдет в суперпозицию состояний, определяемую аргументами $x$, соответствующими этому значению функции:

$$\frac{1}{\sqrt{2^m}} \sum_{x=0}^{2^m-1} |x\rangle\,|C^x \bmod N\rangle \quad \mapsto \quad \sqrt{\frac{r}{2^m}} \sum_{b=0}^{2^m/r-1} |a+br\rangle\,|C^a \bmod N\rangle\,.$$

Полученная суперпозиция определяется соответствующим числом $a$, любое конкретное значение которого из диапазона $[0; r-1]$ будет задействовано равновероятно. Для вычисления периода на основе полученной суперпозиции используется квантовое преобразование Фурье (КПФ). КПФ является унитарным преобразованием и производит нормированное дискретное преобразование Фурье (ДПФ) над вектором состояния, т.е. над последовательностью амплитуд базисных состояний (рис. 6.21).

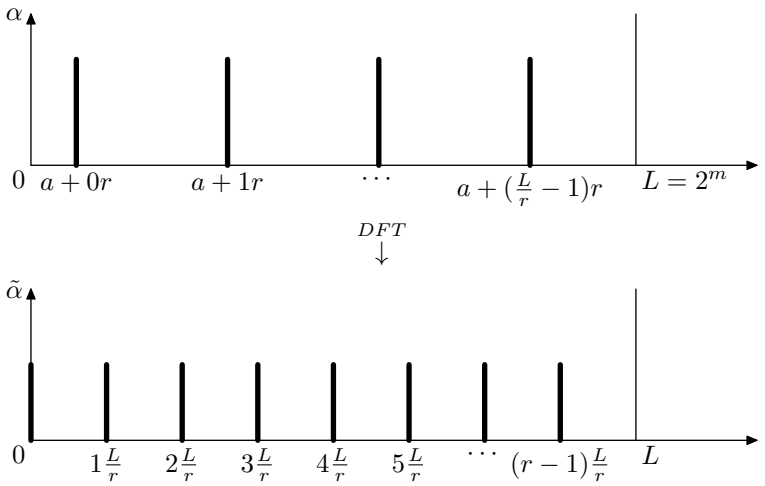



Рис. 6.21. Преобразование периодической последовательности с помощью ДПФ

Как известно, свойства ДПФ таковы [30], что для периодической последовательности длиной $2^m$ и периодом $r$ оно формирует преобразованную последовательность с ненулевыми элементами лишь в позициях, кратных $2^m/r$:

$$\sqrt{\frac{r}{2^m}} \sum_{b=0}^{2^m/r-1} |a+br\rangle\,|C^a \bmod N\rangle \quad \mapsto \quad \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{i\gamma_k} \left|k\frac{2^m}{r}\right\rangle |C^a \bmod N\rangle\,.$$

Измерив теперь первый регистр, получаем число, равное $k2^m/r$ для некоторого целого $k = 0, \ldots, r-1$:

$$\frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{i\gamma_k} \left|k\frac{2^m}{r}\right\rangle |C^a \bmod N\rangle \quad \mapsto \quad \left|k\frac{2^m}{r}\right\rangle |C^a \bmod N\rangle\,.$$

Для возможных состояний при различных значениях $a$ спектры будут отличаться лишь сдвигами фаз $\gamma_k$, что на результате измерения не отразится.

В случае, когда $2^m$ не кратно $r$, получаемые выражения более громоздки, однако суть остается прежней. Скачки в спектре преобразованной последовательности будут в некоторой степени «размытыми», поскольку $2^m/r$ не является целым числом. При измерении мы с большой вероятностью получаем целое число, близкое к некоторому дробному числу $k2^m/r$. Позже мы опишем механизм извлечения из такого результата измерения искомой величины периода.

В соответствии с принципом отложенного измерения, промежуточный замер второго регистра может быть опущен. Поскольку для последующих вычислений результат замера не нужен, для этого не требуется даже модификация схемы. Это может показаться интуитивно противоречивым, поскольку только что мы опирались именно на факт замера. Однако здесь следует иметь в виду следующее. Схема вычисления $f(x)$ таким образом запутывает состояния обоих регистров (6.8), что все состояния системы, отвечающие некоторому значению функции, находятся в своем отдельном подпространстве, ортогональном подпространствам, соответствующим другим ее значениям. Как следствие, КПФ, выполняемое над первым регистром, производится, в силу квантового параллелизма, одновременно во всех подпространствах, но, тем не менее, в каждом подпространстве оно выполняется

отдельно, не воздействуя на остальные подпространства. Измерение в конце вычислений даст тот же результат, что и с промежуточным измерением: второй регистр выдаст произвольное значение функции, а первый — некоторый скачок из спектра, полученного в соответствующем подпространстве. В то же время, можно произвести измерение лишь первого регистра, поскольку с точностью до фаз спектры во всех случаях одинаковы. Вероятность каждого скачка будет в этом случае определяться сразу всеми подпространствами.

Таким образом, на основе построенного нами симулятора квантового компьютера поиск нетривиального делителя с помощью алгоритма Шора представляется следующей функцией `factorize`:

```
// возведение числа a в степень b по модулю mod
int powmod(int a, int b, int mod)
{
 assert(a >= 0 && b >= 0 && mod > 0);
 a %= mod;
 // начальное значение, коэффициент умножения и номер бита
 int p = (a != 0) ? 1 : 0, q = a, i = 0;
 // пока в степени есть необработанные ненулевые биты
 while ((b & ~((1 << i) - 1)) != 0)
 {
  // если текущий бит в степени установлен
  if ((b & (1 << i)) != 0)
   // домножить на текущий коэффициент
   p = p * q % mod;
  // возводим текущий коэффициент в квадрат
  q = q * q % mod;
  // переходим к следующему биту
  ++i;
 };
 return p;
}

// поиск одного из множителей числа (алгоритм Шора)
int factorize(int number)
{
 assert(number > 2);

 int numlength, arglength, base, order, factor;
 numlength = int(ceil(log(1.0 * number) / log(2.0)));
 arglength = int(ceil(2.0 * log(1.0 * number) / log(2.0)));

 for (;;)
 {
  quantum_machine_type qm(arglength + 3 * numlength);

  // случайный выбор основания из диапазона [2, number)
  base = int(random<double>() * (number - 2)) + 2;

  // если числа не взаимно просты, возвращаем НОД
  factor = gcd(base, number);
  if (factor > 1)
   break;

  // составление квантовой схемы
  qm.add_circuit(
```

```
    circuit_hadamard_type(arglength),
    qm.range(0, arglength));
  qm.add_circuit(
    circuit_powmod_type(arglength, numlength, base, number),
    qm.range(0, qm.qubit_number()));
  qm.add_circuit(
    circuit_fourier_type(arglength),
    qm.range(0, arglength));

  // выполнение и измерение
  qm.prepare(0);
  qm.run();
  int result = qm.measure() & ((1 << arglength) - 1);

  // извлечение периода из результата измерения
  order = extract(result, 1 << arglength, number);

  // если, действительно, получен период
  // и если он четный
  if (order > 0 &&
    powmod(base, order, number) == 1 &&
    (order & 1) == 0)
  {
    // вычисление делителя исходного числа
    factor = gcd(
      powmod(base, order / 2, number) + number - 1,
      number);
    // если нашли нетривиальный, заканчиваем поиск
    if (factor > 1)
      break;
  };
 };
 return factor;
}
```

Квантовый компьютер содержит $m+3n$ кубитов, поскольку помимо двух описанных выше регистров для аргументов и значений $f(x)$ нужны еще два регистра для промежуточных значений в процессе ее вычисления. Внутри цикла выполняется случайный выбор произвольного основания $C$. Для проведения вычислений необходимо, чтобы оно было взаимно простым с раскладываемым числом $N$. Если оно таковым не является, функция возвращает их общий делитель. В противном случае выполняется построение схемы в соответствии с рис. 6.20 (измерение второго регистра отложено), ее выполнение и замер результата.

Извлечение значения порядка из результата измерений производится с помощью функции `extract`, которая детально будет рассмотрена ниже. Если найдено удачное значение порядка, функция вычисляет и возвращает нетривиальный делитель. Функция `powmod` используется лишь для проверки результата поиска периода, при проведении же квантовой части вычислений она не задействуется.

Далее опишем подробно реализацию использованных здесь квантовых схем и функций.

### 6.4.2. Модульное возведение в степень

Схема модульного возведения в степень выполняет классическое обратимое вычисление и потому может быть построен на основе лишь классических обратимых вентилей: NOT, CNOT и Toffoli. Нами будет использован простой механизм модульного возведения

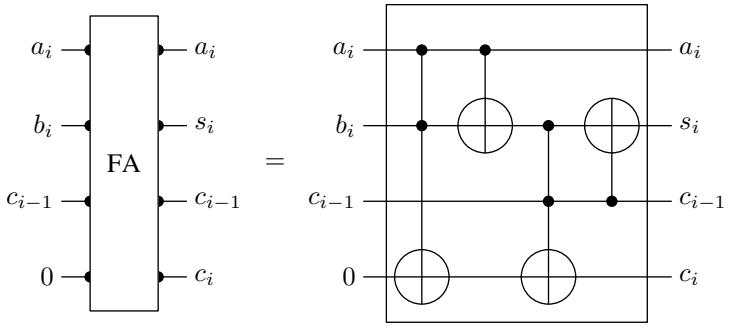



Рис. 6.22. Обратимый полный сумматор (схема full adder)

константы в степень, кратко описанный автором алгоритма в оригинальной работе [77]. Результат при такой методике получается путем многократного условного модульного умножения единицы на заданные на этапе построения схемы другие константы. В свою очередь, модульное умножение выполняется путем многократного модульного суммирования. Наконец, модульное суммирование будет нами реализовано через обыкновенное суммирование с использованием операции сравнения.

Покажем, поочередно усложняя схемы, как могут быть реализованы все эти шаги. Многие схемы будут нами реализованы не столько эффективно, сколько наглядно. К примеру, при выполнении модульного сложения реализованной нами схемой выполняется пять операций обыкновенного сложения: одна основная, выполняющая целевую задачу, и по две в каждой из двух сопутствующих операций сравнения. Это, разумеется, не слишком эффективный подход, однако в нашем случае он обеспечивает достаточно высокую степень наглядности. Общее количество вентилей, необходимых для модульного возведения в степень, сохраняет при этом порядок $O(n^3)$.

### Сложение двух чисел

Рассмотрим сначала общую обратимую схему сложения двух чисел, заданных наборами битов, после чего упростим ее до схемы сложения одного числа с константой. Каждое слагаемое задано набором из $n$ битов, т.е. оба представляют собой неотрицательные числа от 0 до $2^n - 1$.

Подобная схема суммирования строится на основе сумматоров, а именно $n - 1$ полных сумматоров и одного полусумматора [62].

Полный сумматор (full adder) в классической схемотехнике — схема с тремя входами и двумя выходами, осуществляющая сложение единиц на трех входах и выдающее двубитный результат. На вход подаются по одному очередному биту от обоих слагаемых и бит переноса с предыдущего сумматора. На выходе — очередной бит значения суммы и бит переноса текущего шага. В таком виде схема полного сумматора не является обратимой, в связи с чем для квантовых вычислений используется адаптированный вариант [62] (рис. 6.22).

Выходное значение бита суммы заменяет собой значение бита одного из входных чисел. Для знака переноса резервируется еще один бит, который на входе должен быть установлен в 0. Из таблицы истинности мы видим, что в таком виде схема является обратимой (табл. 6.1).

Схема сложения двух чисел строится путем каскадного присоединения набора полных сумматоров к каждой очередной паре входных битов. Для самой первой пары входных битов (нулевой) предыдущий бит переноса не определен, в связи с чем вместо полного сум-

Таблица 6.1. Таблица истинности полного сумматора

| 0 | $c_{i-1}$ | $b_i$ | $a_i$ | $0 \oplus c_i$ | $c_{i-1}$ | $s_i$ | $a_i$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

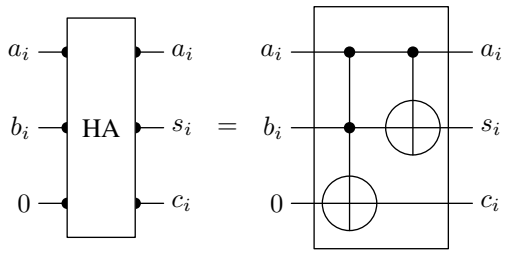



Рис. 6.23. Обратимый полусумматор (схема half adder)

матора используется полусумматор (half adder) (рис. 6.23), который отличается от полного сумматора отсутствием одного входа (табл. 6.2).

Для промежуточного хранения переносов требуются дополнительные биты, изначально установленные в 0. Соединенные каскадно один полусумматор и $n-1$ полных сумматоров позволяют построить преобразование $|a\rangle\,|b\rangle\,|0\rangle \mapsto |a\rangle\,|(a+b) \bmod 2^n\rangle\,|c\rangle$. Т.е. вместо числа $b$ получаем значение суммы $a$ и $b$ по модулю $2^n$, чего и добивались, а дополнительно выделенные под флаги переносов нулевые биты остаются «грязными». Для очистки битов переноса, выставленных полными сумматорами, может быть использован мажоритарный вентиль с тремя входами [62]. Мажоритарный вентиль (majority gate) реализует логическую функцию, на выходе которой оказывается то значение, которых на входе большинство (табл. 6.3). Иначе говоря, обратимый мажоритарный вентиль (рис. 6.24) меняет значение последнего бита, если на входе две или три единицы, и оставляет неизменным в противном случае.

Использование мажоритарного вентиля над возможными выходными значениями полного сумматора (табл. 6.1) дает необходимый результат, если временно инвертировать значение бита суммы. Таким образом, схему очистки бита переноса после полного сумматора (full-adder carry reverser) получаем в виде мажоритарного вентиля, заключенного между двух инверторов (рис. 6.25). Последовательность выходных значений после каждого из этих трех вентилей отражена в табл. 6.4.

Наконец, рассмотрим схему очистки нулевого бита переноса. Полусумматор выставляет бит переноса лишь в одном случае — когда на обоих входах была единица. В этом случае значение суммы будет равно нулю. Таким образом, схема очистки бита переноса после полусумматора (half-adder carry reverser) должна обращать этот бит лишь в случае, когда первый ее вход равен единице, а второй — нулю. Такое поведение задается вентилем Тоффоли с одним инвертированным входом (рис. 6.26). В табл. 6.5 отражена последовательность выполняемых преобразований над выходными значениями полусумматора.

Таблица 6.2. Таблица истинности полусумматора

| $0$ | $b_0$ | $a_0$ | $0 \oplus c_0$ | $s_0$ | $a_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 |

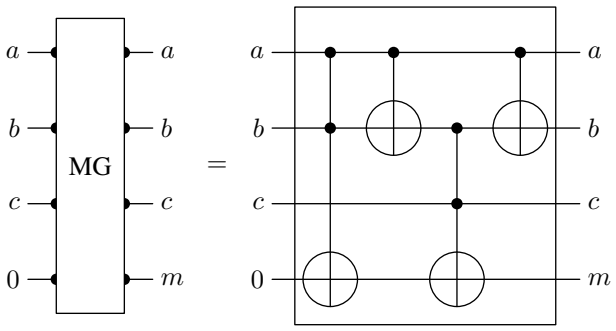



Рис. 6.24. Обратимый мажоритарный вентиль с тремя входными значениями (majority gate)

Теперь мы имеем достаточно базовых элементов, чтобы построить полную схему сложения двух $n$-битных чисел (рис. 6.27). В процессе работы она использует $n$ вспомогательных битов, которые для правильной работы должны быть изначально установлены в 0. Вспомогательные биты каскадно заполняются значениями переносов по мере вычисления суммы двух чисел. После вычислений схема снова сбрасывает значения этих битов в 0 путем каскадного выполнения схем очистки в обратном порядке.

Стоит отметить, что последний вспомогательный бит (с номером $n-1$) в данной схеме функциональной нагрузки не несет, поскольку его результат нигде не учитывается и лишь требует последующей очистки. Он присутствует в схеме по той причине, что иначе нижний полный сумматор «висел бы в воздухе», вообще же для сложения по модулю $2^n$ достаточно всего $n-1$ вспомогательных битов. Однако в некоторых случаях в программе бывает нужно определить факт переполнения, т.е. превышения полученной суммой значения $2^n-1$, для чего требуется сохранить значение последнего бита переноса. В этом случае необходимо наличие всех $n$ вспомогательных битов, а самая нижняя схема очистки на рис. 6.27 не нужна.

### Сложение числа с константой

Реализуем на основе рассмотренной только что схемы сложения двух произвольных чисел (рис. 6.27) упрощенный вариант схемы, осуществляющей сложение числа, заданного с помощью $n$ входных битов, с некой $n$-битной константой $C$. Вместо преобразования $|a\rangle\,|b\rangle\,|0\rangle \mapsto |a\rangle\,|(a+b) \bmod 2^n\rangle\,|0\rangle$ в этом случае мы должны построить $|a\rangle\,|0\rangle \mapsto |(a+C) \bmod 2^n\rangle\,|0\rangle$. Это требует исключения из схемы на рис. 6.27 первых $n$ битов, а также модификации сумматоров и очистителей переносов для учета соответствующих битов константы $C$. Внесем также возможность исключения из схемы последнего бита переноса, поскольку для выполнения как такового сложения он нам не нужен. В результате получаем

Таблица 6.3. Таблица истинности мажоритарного вентиля

| 0 | $c$ | $b$ | $a$ | $0 \oplus m$ | $c$ | $b$ | $a$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

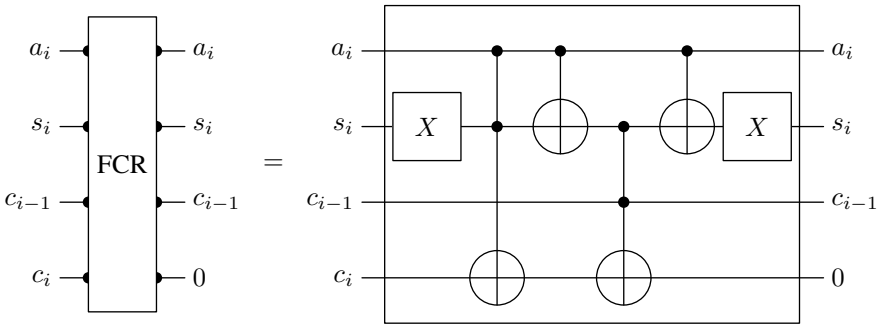



Рис. 6.25. Схема очистки бита переноса после полного сумматора (схема full-adder carry reverser)

схему, изображенную на рис. 6.28.

Содержимое каждого $i$-го сумматора на схеме зависит от $i$-го бита константы $C$, с которой производится сложение (рис. 6.29). Первый сумматор подключен к двум битам, поскольку построен на базе полусумматора, при этом из него исключен один вход. Все остальные сумматоры, кроме последнего, подключаются к трем битам, поскольку являются полными сумматорами также без одного входа. Наконец, последний сумматор строится путем исключения из полного сумматора еще и выходного бита переноса.

В полном сумматоре и полусумматоре (рис. 6.22 и 6.23) нулевым битом контролировались вентили Тоффоли и CNOT. В случае сложения с константой аналогичный результат достигается путем использования вместо них соответственно вентилей CNOT и NOT, при этом битом константы управляется само по себе их наличие. Пунктиром на рис. 6.29 обозначены вентили, помещаемые в схему лишь в случае, когда соответствующий бит константы установлен в единицу.

Аналогично реализуются схемы очистки битов переноса (рис. 6.30). Они также построены на базе приведенных ранее (рис. 6.25 и 6.26), при этом контроль с первого бита перенесен на бит константы.

Приведенная на рис. 6.28 схема для построенного ранее симулятора реализуется следующим классом:

```
// схема ADD сложения n-кубитного числа и константы num
// размещается на 2n-1 или 2n кубитах (параметр storecarry)
// в первом случае кубиты разделены на группы (n) + (n-1):
// |a> |0>   -->   |a + num> |0>
```

Таблица 6.4. Порядок очистки бита переноса после полного сумматора

| FA output | | | | NOT | | | | MAJORITY | | | | NOT | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $c_i$ | $c_{i-1}$ | $s_i$ | $a_i$ | $c_i$ | $c_{i-1}$ | $\bar{s_i}$ | $a_i$ | $c_i \oplus m$ | $c_{i-1}$ | $\bar{s_i}$ | $a_i$ | 0 | $c_{i-1}$ | $s_i$ | $a_i$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

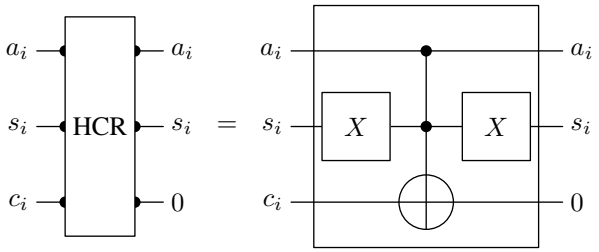



Рис. 6.26. Схема очистки бита переноса после полусумматора (схема half-adder carry reverser)

```
// первая часть - a, становится (a + num) % 2^n
// вторая часть - 0, используется в процессе и возвращается в 0
// во втором случае добавляется еще один кубит:
// |a> |0> |0>   -->   |a + num> |0> |c>
// третья часть - кубит переноса, инвертируется при переполнении
class circuit_add_type: public circuit_type
{
public:
 circuit_add_type(int n, int num, bool storecarry = false):
  circuit_type(storecarry ? (2 * n) : (2 * n - 1))
 {
  assert(num >= 0 && num < (1 << n));
  // вносим сумматоры, переносы сохраняются
  for (int i = 0; i < n; ++i)
  {
   bool ibitset = (num & (1 << i)) != 0;
   // полусумматор, управляемый константой
   if (i < n - 1 || storecarry)
    if (ibitset)
     add_binary(gate_cnot_type(), n + i, i);
   if (ibitset)
    add_unary(gate_not_type(), i);
   // остальная часть полного сумматора
   if (i > 0)
   {
    if (i < n - 1 || storecarry)
```

Таблица 6.5. Порядок очистки бита переноса после полусумматора

| HA output | | | NOT | | | Toffoli | | | NOT | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $c_0$ | $s_0$ | $a_0$ | $c_0$ | $\bar{s_0}$ | $a_0$ | $c_0 \oplus (\bar{s_0} \wedge a_0)$ | $\bar{s_0}$ | $a_0$ | 0 | $s_0$ | $a_0$ |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

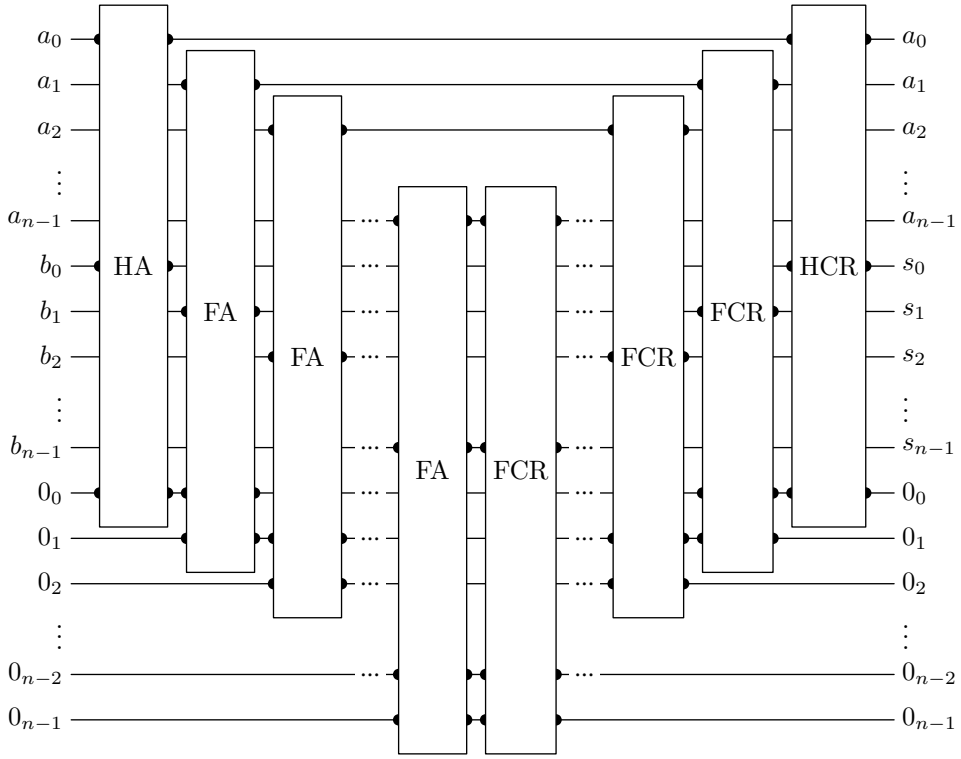



Рис. 6.27. Схема сложения двух $n$-битных чисел

```
    add_ternary(gate_toffoli_type(), n + i, n + i - 1, i);
   add_binary(gate_cnot_type(), i, n + i - 1);
  };
};
// чистим сохраненные переносы, кроме последнего
for (int i = n - 2; i >= 0; --i)
{
 bool ibitset = (num & (1 << i)) != 0;
 if (i > 0)
 {
  // мажоритарный вентиль с одним входом - константой
  // и одним инвертированным входом
  add_unary(gate_not_type(), i);
  if (ibitset)
   add_binary(gate_cnot_type(), n + i, i);
  if (ibitset)
```

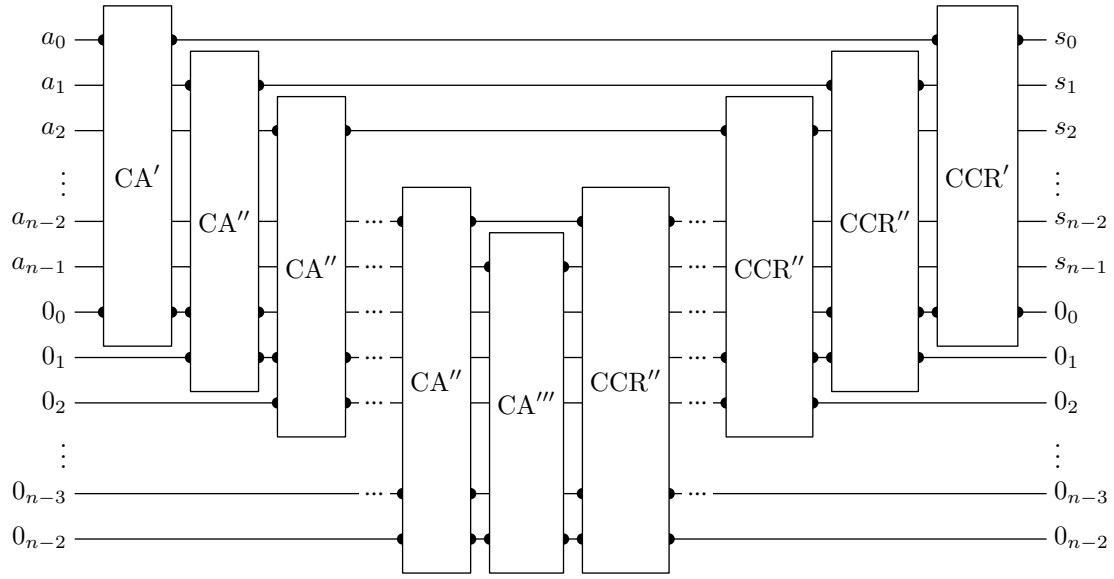
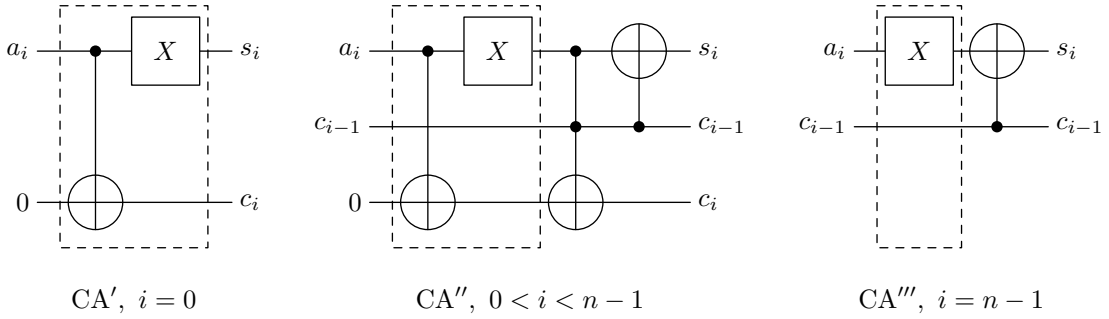



Рис. 6.28. Схема добавления к $n$-битному числу константы (схема add)




CA′, $i = 0$ CA″, $0 < i < n - 1$ CA‴, $i = n - 1$

Рис. 6.29. Схемы отдельных сумматоров для добавления константы (схемы constant adder)

```
    add_unary(gate_not_type(), i);
   add_ternary(gate_toffoli_type(), n + i, n + i - 1, i);
   if (ibitset)
    add_unary(gate_not_type(), i);
   add_unary(gate_not_type(), i);
  }
  else
  {
   // сброс самого первого переноса
   if (ibitset)
   {
    add_unary(gate_not_type(), i);
    add_binary(gate_cnot_type(), n + i, i);
    add_unary(gate_not_type(), i);
   };
  };
 };
}
```

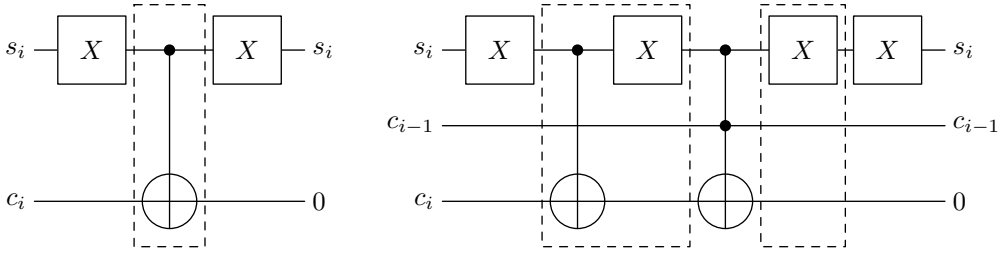



Рис. 6.30. Схемы очистки переносов после добавления константы (схемы constant-adder carry reverser)

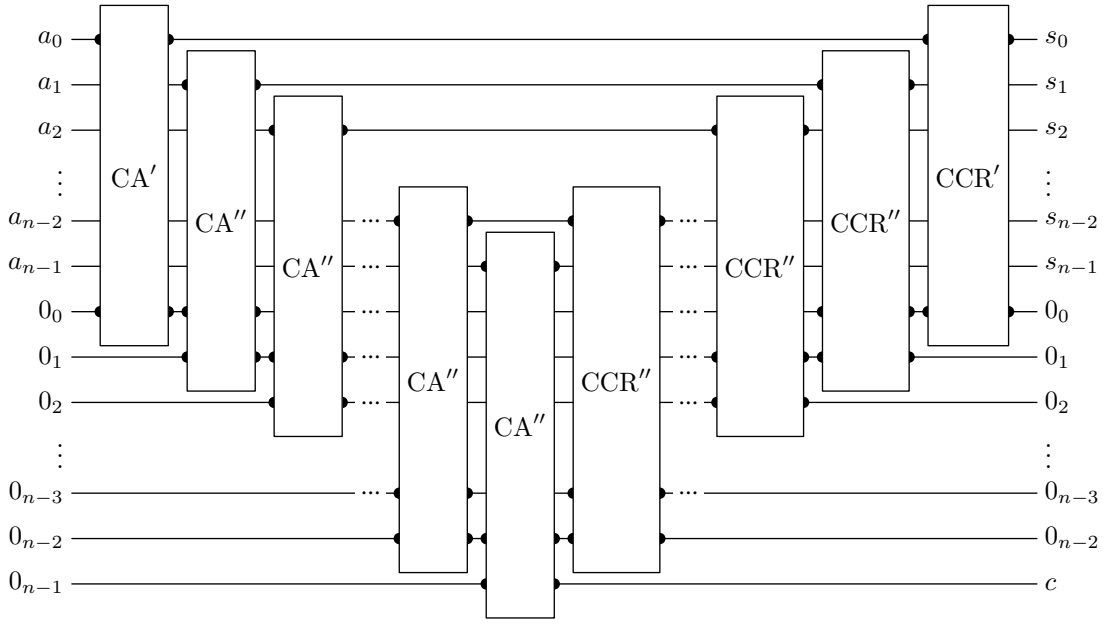



Рис. 6.31. Схема добавления к числу константы с сохранением последнего переноса (схема addc)

```
};
```

Конструктор схемы принимает три параметра, из которых обязательных два: разрядность входного числа и константа, с которой осуществляется сложение. Третий параметр отвечает за сохранение последнего бита переноса, он является необязательным и по умолчанию сброшен. Если флаг сохранения переноса сброшен, схема размещается на $2n - 1$ битах и соответствует изображенной на рис. 6.28. Когда же флаг установлен, схема размещается на $2n$ битах, причем после выполнения сложения последний бит инвертирует свое первоначальное значение в случае, если произошло переполнение (рис. 6.31).

Возможность сохранения последнего бита переноса будет использована нами в дальнейшем при реализации операции сравнения.

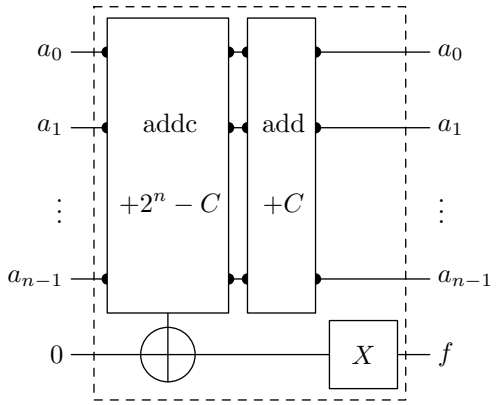



Рис. 6.32. Схема сравнения числа с константой (схема less)

### Сравнение числа с константой

Для того, чтобы на базе описанного только что сложения числа с константой реализовать модульное сложение с произвольным основанием, нам потребуется операция сравнения. Для наших целей будет удобно реализовать операцию «меньше», т.е. схему, инвертирующую значение некоторого бита в случае, если число на входе меньше, чем заданная константа: $|a\rangle\,|0\rangle \mapsto |a\rangle\,|a < C\rangle$.

Реализацию схемы сравнения $a < C$ будем выполнять на базе описанной схемы сложения с учетом сохранения последнего флага переноса (рис. 6.31). При этом будем исходить из следующих соображений. Очевидно, что для $C = 0$ результат операции $a < C$ между неотрицательными числами не будет истинным никогда. Для остальных случаев воспользуемся следующим соотношением:

$$a < C \quad \Leftrightarrow \quad a + (2^n - C) < 2^n.$$

При $C > 0$ оба складываемых числа оказываются меньше $2^n$, что позволяет использовать схему сложения $n$-битного числа с $n$-битной константой. Работа операции сравнения базируется на том факте, что при $a < C$ в процессе сложения числа $a$ и константы $2^n - C$ по модулю $2^n$ переполнения не возникнет, т.е. последний бит переноса установлен не будет. Если инвертировать этот бит, получим требуемый на выходе результат. Поскольку во время сложения входное число меняется, его требуется вернуть обратно, для чего снова должно быть выполнено сложение с числом $C$, но уже без сохранения бита переноса (рис. 6.32). Пунктиром обозначена зависимость от значения константы: весь охваченный им фрагмент включается в схему лишь при $C > 0$.

Описанная схема сравнения реализуется следующим классом:

```
// схема LESS сравнения n–кубитного числа и константы num
// размещается на 2n кубитах
// кубиты разделены на группы (n) + (n–1) + (1):
// |a> |0> |0>   —>   |a> |0> |(a < num)>
// первая часть – a, остается неизменной
// вторая часть – 0, используется в процессе и возвращается в 0
// третья часть – флаг less, инвертируется при a < num
class circuit_less_type: public circuit_type
{
public:
```

```
 circuit_less_type(int n, int num):
  circuit_type(2 * n)
 {
  assert(num >= 0 && num < (1 << n));
  if (num > 0)
  {
   pinlist_type pin = range(0, qubit_number());
   // сложим входное число и константу 2^n - num
   // и сохраним последний кубит переноса
   add_circuit(circuit_add_type(n, (1 << n) - num, true), pin);
   // вернем число на место, кубит переноса не затрагиваем
   pin.pop_back();
   add_circuit(circuit_add_type(n, num), pin);
   // инвертируем выход
   add_unary(gate_not_type(), qubit_number() - 1);
  };
 }
};
```

Поскольку используемые схемы сложения требуют $2n$ и $2n-1$ битов соответственно, вся схема сравнения подключается к $2n$ битам, хотя это не отражено на рис. 6.32. Все содержимое схемы подключается лишь при условии, что константа больше нуля, поскольку лишь в этом случае число, добавляемое первой схемой сложения, укладывается в нужный диапазон.

### Модульное сложение числа с константой

При наличии схемы сравнения можем реализовать модульное сложение с произвольным основанием $N < 2^n$: $|a\rangle \mapsto |(a + C) \bmod N\rangle$. Для наших целей будет достаточно реализовать схему модульного сложения числа с константой лишь для случаев, когда оба аргумента попадают в диапазон $[0, N)$. В этой ситуации результат модульного сложения определяется следующим выражением:

$$(a + C) \bmod N = \begin{cases} a + C, & a + C < N \\ a + C - N = a - (N - C), & a + C \geqslant N. \end{cases}$$

Таким образом, необходимо реализовать две ветви выполнения для разного результата сравнения $a+C < N$ или, что то же самое, $a < N-C$ (рис. 6.33). На выходе схема сравнения в случае положительного результата взводит входной бит. Этим битом управляется схема сложения $a + C$. Другая ветвь должна отработать в случае, если управляющий бит не установлен, поэтому он инвертируется.

В двоичной арифметике по модулю $2^n$ вычитание из числа $a$ некоторого ненулевого числа $b$ может быть выполнено путем суммирования $a$ и $2^n - b$. В связи с этим во второй схеме сложения для достижения требуемого результата добавляем константу $2^n - (N - C)$.

Наконец, на выходе всей схемы модульного сложения управляющий бит должен быть снова сброшен в ноль. Для этого вносим еще одну схему сравнения. На момент ее выполнения бит проверки взведен лишь в случае, если первое сравнение выполнилось с отрицательным результатом и, соответственно, отработала вторая схема сложения. Нетрудно заметить, что в этом случае результат сложения всегда будет меньше исходной константы $C$, поскольку равен $C + 2^n - (N - a)$, т.е. разности $C$ и некоторого положительного числа. Таким образом, добавленная в конце схема сравнения сбрасывает установленный в этом случае бит в ноль.

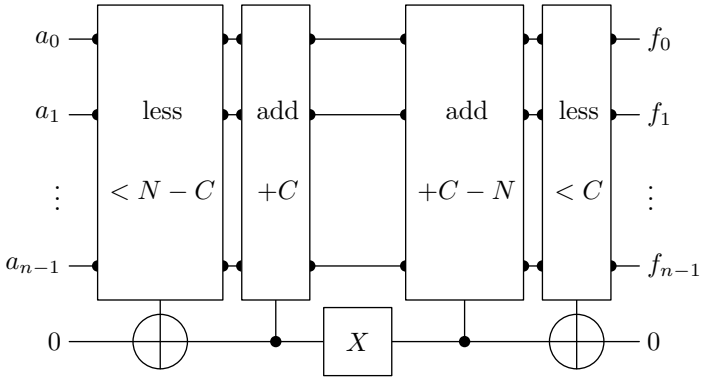



Рис. 6.33. Схема модульного сложения числа с константой (схема addmod)

Следующий класс реализует описанную схему модульного сложения:

```
// схема ADDMOD сложения n-кубитного числа и константы num
// по модулю mod, размещается на 2n кубитах
// кубиты разделены на группы (n) + (n):
// |a> |0>  -->  |(a + num) % mod> |0>
// первая часть - a, становится (a + num) % mod
// вторая часть - 0, используется в процессе и возвращается в 0
class circuit_addmod_type: public circuit_type
{
public:
 circuit_addmod_type(int n, int num, int mod):
  circuit_type(2 * n)
 {
  assert(mod > 0 && mod < (1 << n));
  assert(num >= 0 && num < mod);

  pinlist_type pin = range(0, qubit_number());

  // проверим, меньше ли число константы mod - num
  add_circuit(circuit_less_type(n, mod - num), pin);

  // если меньше, прибавим num
  circuit_add_type add1(n, num);
  add_circuit(circuit_controlled_type(add1), pin);
  // иначе
  add_unary(gate_not_type(), pin.back());
  // прибавим num - mod
  circuit_add_type add0(n, (1 << n) - mod + num);
  add_circuit(circuit_controlled_type(add0), pin);

  // вернем в ноль кубит проверки
  add_circuit(circuit_less_type(n, num), pin);
 }
};
```

Конструктор принимает три параметра: разрядность входного числа, добавляемую константу и величину модуля. Вся схема, как и схема сравнения, подключается к $2n$ битам.

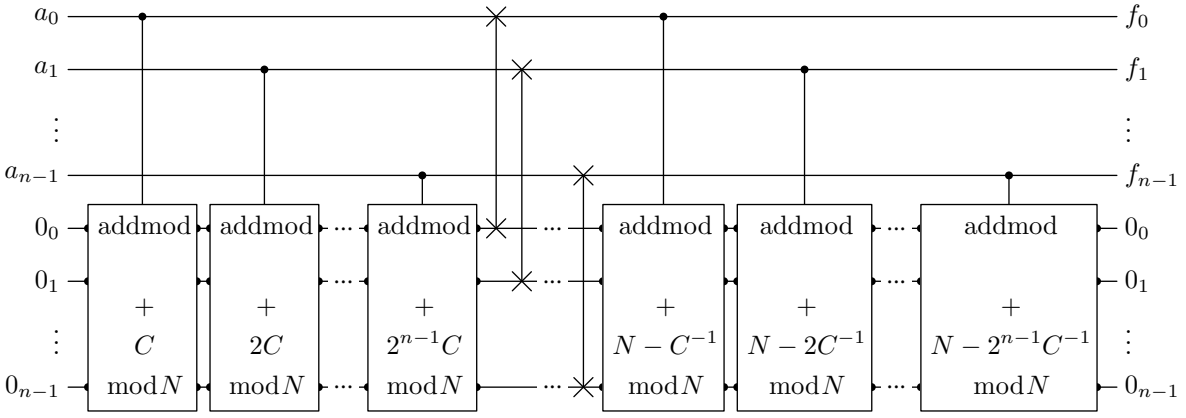



Рис. 6.34. Схема модульного умножения числа на константу (схема mulmod)

**Умножение числа на константу**

Опишем построение схемы модульного умножения на константу, т.е. $|a\rangle \mapsto |(a \cdot C) \bmod N\rangle$. Обратимая схема, выполняющая такое преобразование, возможна при условии, что $C$ и $N$ взаимно просты. Общий принцип построения такой схемы, предложенный в оригинальной работе [77], базируется на использовании набора схем модульного сложения, управляемых битами входного числа. Входное $n$-битное число $a$ может быть выражено через значения составляющих его битов $a_i, i = 0, \ldots, n-1$:

$$a = a_0 \cdot 1 + a_1 \cdot 2 + a_2 \cdot 4 + \cdots + a_{n-1} \cdot 2^{n-1} = \sum_{i=0}^{n-1} a_i 2^i.$$

Таким образом, результат выполнения требуемой операции может быть представлен следующим образом:

$$a \cdot C \bmod N = \sum_{i=0}^{n-1} (a_i 2^i C \bmod N) \bmod N.$$

На основе полученного выражения строится первая часть схемы умножения (рис. 6.34). Набор управляемых схем модульного сложения с основанием $N$ поочередно добавляет к нулевому значению предварительно вычисленные константы $2^i C \bmod N$. После их выполнения вспомогательные $n$ битов содержат требуемое значение. Последующий набор вентилей SWAP производит обмен значениями обоих регистров так, чтобы требуемое значение оказалось в основных $n$ битах. Теперь во вспомогательном регистре содержится значение входного числа $a$, от которого следует избавиться, сбросив снова все биты в 0.

Для решения этой задачи предлагается умножить полученный на текущий момент результат (а именно $aC \bmod N$) на величину $C^{-1}$, обратную константе $C$ по модулю $N$, и вычесть полученное число из вспомогательного регистра [77]. Величиной, обратной к $C$ по модулю $N$, называют такое число $C^{-1}$, для которого удовлетворяется сравнение $C \cdot C^{-1} \equiv 1 \pmod N$. Таким образом, умножив $aC \bmod N$ на $C^{-1}$, мы снова получаем $a$, а вычитая этот результат из вспомогательного регистра, обнуляем последний. В соответствии с правилами модульной арифметики, чтобы выполнить модульное вычитание некоторого числа $b$, достаточно произвести модульное сложение с числом $N - b$, что и отражено в правой части

полученной схемы (рис. 6.34).

В результате при реализации схемы модульного умножения получаем следующий класс:

```
// схема MULMOD умножения n-кубитного числа и константы num
// по модулю mod, размещается на 3n кубитах
// кубиты разделены на группы (n) + (2n):
// |a> |0>  -->  |(a * num) % mod> |0>
// первая часть - a, становится (a * num) % mod
// вторая часть - 0, используется в процессе и возвращается в 0
class circuit_mulmod_type: public circuit_type
{
public:
 circuit_mulmod_type(int n, int num, int mod):
  circuit_type(3 * n)
 {
  assert(mod > 0 && mod < (1 << n));
  assert(num >= 0 && num < mod);
  assert(gcd(num, mod) == 1);

  // номера кубитов, к которым привязываются схемы сложения
  pinlist_type pin = range(n, qubit_number());
  // создание и привязка управляемых схем сложения
  for (int i = 0; i < n; ++i)
  {
   int term = (((1 << i) % mod) * num) % mod;
   circuit_addmod_type addmod(n, term, mod);
   pin.push_back(i);
   add_circuit(circuit_controlled_type(addmod), pin);
   pin.pop_back();
  };
  // замена местами результата умножения и исходного значения
  for (int i = 0; i < n; ++i)
   add_binary(gate_swap_type(), i, n + i);
  // обращение множителя по модулю mod
  num = invmod(num, mod);
  // обнуление исходного значения
  for (int i = 0; i < n; ++i)
  {
   int term = (mod - (((1 << i) % mod) * num) % mod) % mod;
   circuit_addmod_type addmod(n, term, mod);
   pin.push_back(i);
   add_circuit(circuit_controlled_type(addmod), pin);
   pin.pop_back();
  };
 }
};
```

Помимо $n$ входных битов, для схем модульного сложения используется $2n$ вспомогательных (на рис. 6.34 отражены не все из них). В связи с этим вся схема модульного умножения подключается к $3n$ битам. Проверка на равенство НОД единице выполняется, чтобы убедиться, что основание $N$ и константа $C$ взаимно просты, поскольку в противном случае невозможно получение модульной обратной величины [8]. Сама функция вычисления модульной обратной величины требует особого разговора, к ней мы вернемся чуть позже.

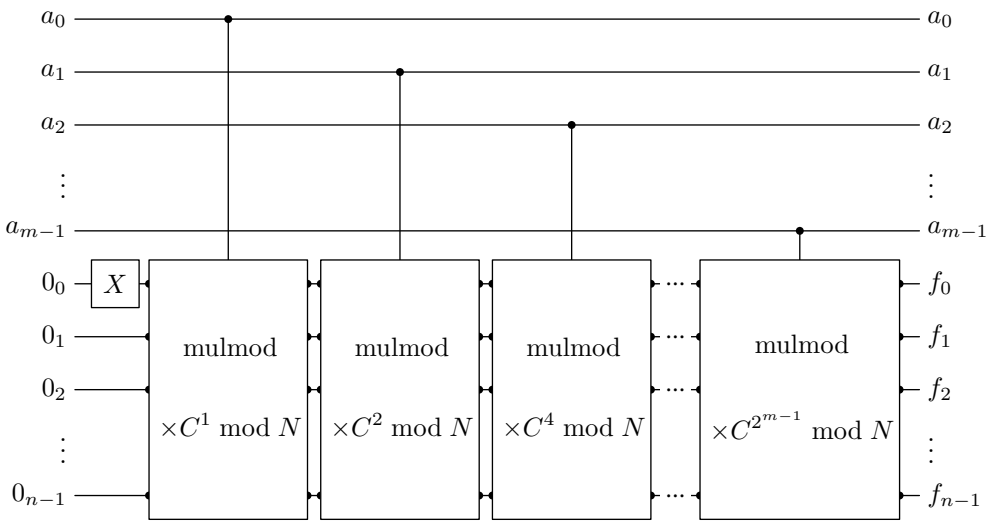



Рис. 6.35. Схема модульного возведения константы в степень (схема powmod)

### Возведение константы в степень

Наконец, построим полную схему, выполняющую возведение в степень константы $C$ по модулю $N$. На вход схемы поступает $m$-битный регистр с числом, выражающим степень, и обнуленный $n$-битный регистр для результата. Схема выполняет преобразование $|a\rangle\,|0\rangle \mapsto |a\rangle\,|C^a \bmod N\rangle$. Как и в предыдущем случае, такое преобразование обратимо только в случае, когда $C$ и $N$ взаимно просты.

Входное число $a$ снова представляется в виде набора битов $a_i, i = 0, \ldots, m-1$, после чего требуемая операция выражается следующим образом:

$$C^a \bmod N = C^{\sum_{i=0}^{m-1} a_i 2^i} \bmod N = \prod_{i=0}^{m-1} (C^{a_i 2^i} \bmod N) \bmod N.$$

Схема строится в виде набора управляемых схем модульного умножения, последовательно домножающих текущее значение произведения на константы $C^{2^i} \bmod N$ (рис. 6.35). В самом начале текущее значение произведения выставляется равным единице, для чего к соответствующему биту подключается вентиль NOT.

Описанная схема реализуется следующим классом:

```
// схема POWMOD возведения константы num в степень
// m-кубитного числа по n-кубитному модулю mod
// размещается на m+3n кубитах
// кубиты разделены на группы (m) + (n) + (2n):
// |a> |0> |0>  -->  |a> |(num ^ a) % mod> |0>
// первая часть - a, остается неизменной
// вторая часть - 0, становится (num ^ a) % mod
// третья часть - 0, используется в процессе и возвращается в 0
class circuit_powmod_type: public circuit_type
{
public:
 circuit_powmod_type(int m, int n, int num, int mod):
  circuit_type(m + 3 * n)
```

```
 {
  assert(m > 0 && n > 0);
  assert(mod > 0 && mod < (1 << n));
  assert(num >= 0 && num < mod);
  assert(gcd(num, mod) == 1);

  // сформируем на выходе единицу
  add_unary(gate_not_type(), m);

  // номера кубитов, к которым привязываются схемы умножения
  pinlist_type pin = range(m, qubit_number());
  // первый множитель
  int factor = num;
  // создание и привязка управляемых схем умножения
  for (int i = 0; i < m; ++i)
  {
   // управляемая схема умножения
   circuit_mulmod_type mulmod(n, factor, mod);
   // добавление управляющего кубита и присоединение схемы
   pin.push_back(i);
   add_circuit(circuit_controlled_type(mulmod), pin);
   pin.pop_back();
   // вычисление последующего множителя
   factor = factor * factor % mod;
  };
 }
};
```

Схема подключается к $m + 3n$ битам, из которых $m + n$ изображены на рис. 6.35, а остальные используются для промежуточных результатов. Вычисление очередных множителей выполняется путем последовательного модульного возведения текущего множителя в квадрат.

### Вычисление НОД и модульной обратной величины

Опишем кратко механизмы поиска НОД и модульной обратной величины, поскольку они в нашем случае требуют эффективной классической реализации.

Поиск НОД (GCD, greatest common divisor) традиционно выполняется с помощью широко известного алгоритма Евклида [8]. Суть алгоритма для случая поиска НОД чисел $a$ и $b$ заключается в рекуррентном вычислении последовательности остатков от попарного деления до тех пор, пока не будет получен ноль. Значением НОД в этом случае является последний ненулевой член последовательности:

$$r_{-2} = a, \quad r_{-1} = b,$$
$$r_i = r_{i-2} \bmod r_{i-1} = r_{i-2} - r_{i-1} \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor;$$
$$\gcd(a, b) = r_{n-1} \neq 0, \quad r_n = 0.$$

Здесь $\lfloor \cdot \rfloor$ — взятие ближайшего снизу целого (операция «пол», «floor»). Программно этот алгоритм представляется следующей функцией:

```
// вычисление наибольшего общего делителя (алгоритм Евклида)
int gcd(int a, int b)
{
```

```
 while (b != 0)
 {
  int r = a % b;
  a = b;
  b = r;
 };
 return (a < 0) ? -a : a;
}
```

Поиск модульной обратной величины (modular multiplicative inverse) осуществляется путем решения сравнения первой степени и может быть выполнен на основе разложения в цепную дробь [8]. Сделаем небольшое отступление и опишем кратко основные понятия цепных дробей.

Любая рациональная дробь $a/b$ может быть представлена в виде конечной цепной дроби:

$$\frac{a}{b} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cdots \cfrac{1}{q_{n-1} + \cfrac{1}{q_n}}}}} = [q_0; q_1, q_2, q_3, \ldots q_{n-1}, q_n].$$

Вычисление положительных целых коэффициентов $q_i$ выполняется по следующим формулам:

$$\begin{aligned}
q_0 &= \left\lfloor \frac{a}{b} \right\rfloor, \quad f_0 = \frac{a}{b} - q_0; \\
q_1 &= \left\lfloor \frac{1}{f_0} \right\rfloor, \quad f_1 = \frac{1}{f_0} - q_1; \\
q_2 &= \left\lfloor \frac{1}{f_1} \right\rfloor, \quad f_2 = \frac{1}{f_1} - q_2; \\
&\ldots \\
q_n &= \left\lfloor \frac{1}{f_{n-1}} \right\rfloor, \quad f_n = \frac{1}{f_{n-1}} - q_n = 0.
\end{aligned}$$

Здесь $q_i$ — целые, а $f_i$ — дробные части соответствующих дробей. Описанные рекуррентные соотношения можно переписать, используя вместо дробных частей остатки от деления:

$$\begin{aligned}
q_0 &= \left\lfloor \frac{a}{b} \right\rfloor, \quad r_0 = a - bq_0; \\
q_1 &= \left\lfloor \frac{b}{r_0} \right\rfloor, \quad r_1 = b - r_0 q_1; \\
q_2 &= \left\lfloor \frac{r_0}{r_1} \right\rfloor, \quad r_2 = r_0 - r_1 q_2; \\
&\ldots \\
q_n &= \left\lfloor \frac{r_{n-2}}{r_{n-1}} \right\rfloor, \quad r_n = r_{n-2} - r_{n-1} q_n = 0.
\end{aligned}$$

В этом случае мы видим, что разложение в цепную дробь тесно связано с алгоритмом Евклида. По сути, в обоих случаях выполняются одни и те же действия, но при поиске

НОД нас интересует последовательность остатков от деления $r_i$, а при построении цепной дроби — последовательность соответствующих частных $q_i$.

При разложении некоторого числа в цепную дробь возникает такое понятие, как $i$-я подходящая дробь. Подходящей дробью $P_i/Q_i$ для некоторой цепной дроби $[q_0; q_1, \dots q_n]$ называют число, выражаемое цепной дробью с теми же элементами до $i$ включительно:

$$\frac{a}{b} = [q_0; q_1, q_2, q_3, \dots q_{n-1}, q_n];$$
$$\frac{P_i}{Q_i} = [q_0; q_1, q_2, \dots q_{i-1}, q_i], \quad i = 0, \dots, n.$$

Для вычисления числителей и знаменателей подходящих дробей на основе коэффициентов цепной дроби справедливы следующие рекуррентные соотношения:

$$P_{-1} = 1, \quad P_0 = q_0, \quad P_i = q_i P_{i-1} + P_{i-2};$$
$$Q_{-1} = 0, \quad Q_0 = 1, \quad Q_i = q_i Q_{i-1} + Q_{i-2}.$$

Наконец, вернемся к теме поиска модульной обратной величины к константе $C$ по модулю $N$. Число $N/C$ может быть разложено в конечную цепную дробь, причем последняя подходящая дробь $P_n/Q_n = N/C$. Модульная обратная величина $C^{-1}$ в этом случае определяется следующим выражением [8]:

$$C^{-1} = (-1)^n P_{n-1}. \tag{6.9}$$

Здесь $P_{n-1}$ — числитель предпоследней подходящей дроби, а $n$ — номер последнего элемента в цепной дроби $N/C$.

Собирая все воедино, получаем следующее. Для поиска модульной обратной величины нам требуется построить последовательность числителей подходящих дробей для числа $N/C$, при этом для вычисления элементов цепной дроби $q_i$ можно использовать алгоритм, близкий к алгоритму Евклида. На фоне этих вычислений требуется хранить счетчик элементов последовательности, поскольку от его четности зависит получаемое в конце значение.

В результате функция поиска модульной обратной величины принимает следующий вид:

```
// вычисление модульной обратной величины
int invmod(int a, int mod)
{
 assert(a > 0 && mod > 0);
 a %= mod;
 assert(gcd(a, mod) == 1);

 // начальные значения для рекуррентных соотношений
 int ppre = 1, pcur = mod / a;
 // раскладываем число в цепную дробь
 int rpre = a, rcur = mod % a, n = 0;
 while (rcur != 0)
 {
  // номер очередной подходящей дроби
  ++n;
  // числитель очередной подходящей дроби
  int pnxt = (rpre / rcur) * pcur + ppre;
```

```
  ppre = pcur;
  pcur = pnxt;
  // очередной остаток (алгоритм Евклида)
  int rnxt = rpre % rcur;
  rpre = rcur;
  rcur = rnxt;
 };
 // для нечетного n инвертируем знак
 return ((n & 1) == 0) ? ppre : (mod - ppre);
}
```

При анализе счетчика элементов в конце функции в случае его четности возвращается число $P_{n-1}$, в противном случае — $N - P_{n-1}$.

### 6.4.3. Квантовое преобразование Фурье

Схема квантового преобразования Фурье, как и схема Уолша-Адамара, реализует квантовое унитарное преобразование, не реализуемое классически. Такое преобразование переводит состояние $m$-кубитной системы, представляемое вектором из $2^m$ элементов, в новое состояние, амплитуды которого определяются результатом ДПФ над последовательностью амплитуд исходного состояния.

Нормированное ДПФ для некоторой последовательности $\{\alpha_k\}$ из $2^m$ элементов определяется следующей формулой:

$$\tilde{\alpha}_l = \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} \alpha_k e^{i\frac{2\pi}{2^m}lk}, \quad l = 0, \ldots, 2^m - 1. \tag{6.10}$$

Такое преобразование переводит произвольное состояние $|\psi\rangle$ в некоторое состояние $\left|\tilde{\psi}\right\rangle$ следующим образом:

$$\sum_{k=0}^{2^m-1} \alpha_k |k\rangle \quad \mapsto \quad \sum_{l=0}^{2^m-1} \tilde{\alpha}_l |l\rangle \equiv \frac{1}{\sqrt{2^m}} \sum_{l=0}^{2^m-1} \sum_{k=0}^{2^m-1} \alpha_k e^{i\frac{2\pi}{2^m}lk} |l\rangle .$$

Отсюда следует, что каждое состояние вычислительного базиса преобразуется в соответствии со следующим выражением:

$$|k\rangle \quad \mapsto \quad \frac{1}{\sqrt{2^m}} \sum_{l=0}^{2^m-1} e^{i\frac{2\pi}{2^m}lk} |l\rangle , \quad k = 0, \ldots, 2^m - 1. \tag{6.11}$$

Эффективная реализация КПФ над состоянием $m$-кубитной системы требует $O(m^2)$ вентилей. Опуская множество подробностей, которые заинтересованный читатель без труда сможет найти в [59, 53, 69, 6], скажем, что суперпозиция (6.11), в которую КПФ переводит каждое базисное состояние $|k\rangle$, является разложимой по состояниям отдельных кубитов. На основе этого разложения воздействие на каждый кубит осуществляется последовательностью из преобразования Адамара и некоторого сдвига фазы, причем сдвиг зависит от конкретных значений битов исходного базисного состояния (рис. 6.36). Полученная путем такого разложения схема порождает нужный результат с инвертированным порядком следования битов, в связи с чем для его обращения схема завершается набором вентилей SWAP. На рис. 6.36 управляемые вентили выполняют сдвиг фазы единичного базисного вектора на угол $\theta_d = \pi/2^d$, т.е. реализуются управляемым вентилем $\hat{R}(2\pi/2^{d+1})$.

Описанная схема в соответствии с рис. 6.36 реализуется следующим классом:

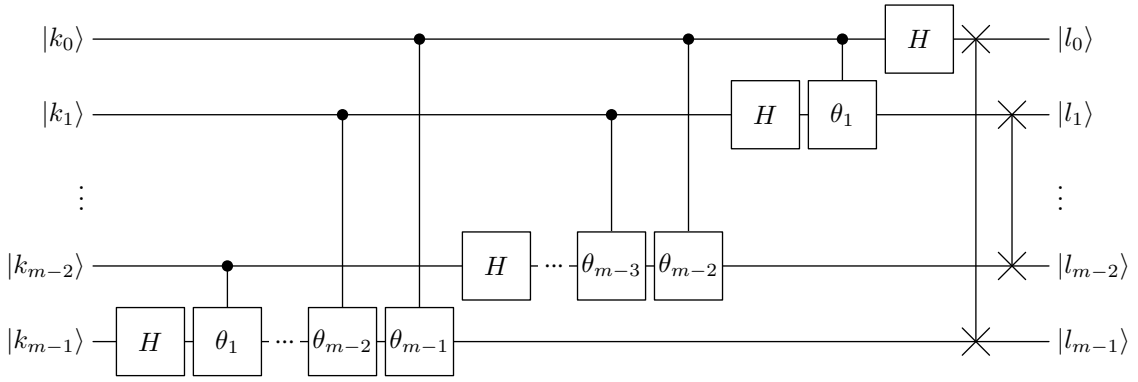



Рис. 6.36. Схема квантового преобразования Фурье (QFT)

```
// n-кубитная схема QFT
class circuit_fourier_type: public circuit_type
{
public:
 circuit_fourier_type(int n):
  circuit_type(n)
 {
  for (int i = qubit_number() - 1; i >= 0; --i)
  {
   // вентиль Адамара
   add_unary(gate_hadamard_type(), i);
   for (int j = i - 1; j >= 0; --j)
   {
    // управляемый сдвиг фазы
    gate_shift_type shift(1.0 / (1 << (i - j + 1)));
    add_binary(circuit_controlled_type(shift), i, j);
   };
  };
  // обращение порядка номеров кубитов
  for (int i = 0; i < qubit_number() / 2; ++i)
   add_binary(gate_swap_type(), i, qubit_number() - i - 1);
 }
};
```

### 6.4.4. Извлечение порядка из результата измерения

Допустим, после выполнения КПФ над первым регистром путем его измерения мы получили некоторое значение $v$. Для извлечения из полученного числа искомого значения порядка $r$ снова используется аппарат цепных дробей. В [77] показано, что с большой вероятностью полученное значение удовлетворяет следующему неравенству:

$$\left| v - k\frac{2^m}{r} \right| \leqslant \frac{1}{2}. \tag{6.12}$$

Здесь $2^m$ — количество различных значений аргумента, представляемых первым регистром, $k$ — произвольный целочисленный множитель. Далее будем отталкиваться от пред-

положения, что неравенство (6.12) выполняется. Деля на $2^m$ и учитывая, что $2^m > N^2$, можно записать следующее:

$$\left|\frac{v}{2^m} - \frac{k}{r}\right| < \frac{1}{2N^2}. \tag{6.13}$$

Поскольку $N \geqslant r$, имеем:

$$\left|\frac{v}{2^m} - \frac{k}{r}\right| < \frac{1}{2r^2}. \tag{6.14}$$

Выражение (6.14) дает возможность воспользоваться теоремой из теории цепных дробей, гласящей, что, если несократимая дробь $k/r$ удовлетворяет неравенству $|\alpha - k/r| < 1/2r^2$, она является подходящей дробью для числа $\alpha$ [33].

Более того, в нашем случае приемлемых вариантов такой подходящей дроби будет не более одного. Как известно, две разные дроби $k/r$ и $k'/r'$ со знаменателями, меньшими $N$, отличаются на величину, большую $1/N^2$ [76]:

$$\left|\frac{k}{r} - \frac{k'}{r'}\right| > \frac{1}{N^2}, \quad \frac{k}{r} \neq \frac{k'}{r'}, r < N, r' < N.$$

Это является следствием того, что в дроби $\left|\frac{kr'-k'r}{rr'}\right|$ числитель не меньше единицы (иначе дроби были бы равны), а знаменатель заведомо меньше $N^2$. Следовательно, более одной подходящей дроби $k/r$ со знаменателем, меньшим $N$, в диапазон, задаваемый неравенством (6.13), попадать не может. Однако нужно иметь в виду, что может попадать менее одной дроби, т.е. мы можем не найти ни одного значения, удовлетворяющего заданным критериям. Так произойдет в случае, если первая же подходящая дробь, удовлетворяющая неравенству (6.13), будет иметь знаменатель $r \geqslant N$.

Таким образом, нам требуется найти знаменатель одной из подходящих дробей в разложении числа $v/2^m$, удовлетворяющей критерию (6.12) или, что удобнее для оценки в целых числах, $2|vr - k2^m| \leqslant r$, а также критерию $r < N$.

Этот способ позволяет найти дробь $k/r$ в младших членах, т.е. дает значение порядка лишь в случае, если $k$ оказалось взаимно простым с $r$. В противном случае будет получен лишь один из множителей порядка. В такой ситуации удобно сохранить значение этого множителя, чтобы на следующей итерации вычислить наименьшее общее кратное (НОК) из обоих результатов, и именно его проверять на соответствие порядку.

Механизм вычисления подходящих дробей был кратко описан нами выше, используем его и в данном случае. Разложение будем производить до тех пор, пока знаменатель очередной подходящей дроби не удовлетворит заданному критерию или не превысит $N$:

```
// извлечение периода из результата измерения
// если не найдено подходящее число, возвращается 0
int extract(int num, int denom, int limit)
{
 assert(num >= 0 && denom > 0 && limit > 0);

 int rc = 0;
 // начальные значения для рекуррентных соотношений
 int ppre = 1, pcur = num / denom;
 int qpre = 0, qcur = 1;
 // раскладываем число в цепную дробь
 int rpre = denom, rcur = num % denom;
```

```
 while (rcur != 0)
 {
  // числитель очередной подходящей дроби
  int pnxt = (rpre / rcur) * pcur + ppre;
  ppre = pcur;
  pcur = pnxt;
  // знаменатель очередной подходящей дроби
  int qnxt = (rpre / rcur) * qcur + qpre;
  qpre = qcur;
  qcur = qnxt;

  // проверяем выход за границы
  if (qcur >= limit)
   break;
  else
  {
   // проверяем удовлетворение условию
   if (2 * abs(num * qcur - denom * pcur) <= qcur)
    rc = qcur;
   // очередной остаток (алгоритм Евклида)
   int rnxt = rpre % rcur;
   rpre = rcur;
   rcur = rnxt;
  };
 };
 return rc;
}
```

На вход функции `extract` подаются числитель и знаменатель раскладываемой дроби, а также предельное значение искомого знаменателя подходящей дроби, перед которым поиск должен быть прекращен. В цикле строится последовательность подходящих дробей до тех пор, пока не будет достигнуто полное совпадение или же превышена допустимая величина знаменателя. Функция возвращает знаменатель подходящей дроби, удовлетворяющей соотношению (6.12), если таковая была найдена. Поскольку соответствующая подходящая дробь может быть не найдена, клиентский код должен проверять возвращаемое функцией значение на равенство нулю, что и выполняется в приведенной выше функции `factorize`.

# Приложение А.

# Шаблоны классов матрицы и вектора

Ниже приводится пример реализации шаблона класса матрицы, предоставляющего возможность выполнения основных математических операций, а также унаследованный от него шаблон вектора, являющийся матрицей шириной в один столбец. Приведенный шаблон `vector_type` по названию напоминает стандартный `std::vector`, однако не следует их путать — стандартный шаблон имеет мало общего с математическим понятием вектора, поскольку не обеспечивает необходимый функционал, а именно поддержку математических операций.

В примере не реализуются конструктор копирования и оператор присваивания, поскольку элементы хранятся не в динамическом массиве, а в контейнере `std::vector`. В этом случае конструкторы копирования и операторы присваивания, предоставляемые компилятором для классов `matrix_type` и `vector_type` по умолчанию, вполне удовлетворяют необходимым требованиям, поскольку за корректное копирование данных в этом случае отвечает уже реализация внутреннего контейнера.

```
// матрица
template <typename e_t>
class matrix_type
{
public:
 typedef e_t element_type;

private:
 int m_vsize, m_hsize;
 std::vector<element_type> m_data;

public:
 // конструктор (инициализация значениями element_type())
 matrix_type(int vsize, int hsize):
  m_vsize(vsize), m_hsize(hsize)
 {
  assert(m_vsize > 0 && m_hsize > 0);
  m_data.resize(m_vsize * m_hsize, element_type());
 }

 // размеры по вертикали и горизонтали
 int vsize(void) const
 {
  return m_vsize;
 }
```

```
int hsize(void) const
{
 return m_hsize;
}

// обращение к элементам по индексам (нумерация с нуля)
element_type & operator ()(int i, int j)
{
 assert(i >= 0 && j >= 0 && i < vsize() && j < hsize());
 return m_data[i * hsize() + j];
}
const element_type & operator ()(int i, int j) const
{
 return const_cast<matrix_type *>(this)->operator ()(i, j);
}

// прибавление матрицы
matrix_type & operator +=(const matrix_type &src)
{
 assert(hsize() == src.hsize() && vsize() == src.vsize());
 #pragma omp parallel for
 for (int i = 0; i < vsize(); ++i)
  for (int j = 0; j < hsize(); ++j)
   (*this)(i, j) += src(i, j);
 return *this;
}
// вычитание матрицы
matrix_type & operator -=(const matrix_type &src)
{
 assert(hsize() == src.hsize() && vsize() == src.vsize());
 #pragma omp parallel for
 for (int i = 0; i < vsize(); ++i)
  for (int j = 0; j < hsize(); ++j)
   (*this)(i, j) -= src(i, j);
 return *this;
}

// сумма двух матриц
friend
matrix_type operator +(
 const matrix_type &src1,
 const matrix_type &src2)
{
 assert(src1.hsize() == src2.hsize());
 assert(src1.vsize() == src2.vsize());
 return (matrix_type(src1) += src2);
}
// разность двух матриц
friend
matrix_type operator -(
 const matrix_type &src1,
 const matrix_type &src2)
{
 assert(src1.hsize() == src2.hsize());
 assert(src1.vsize() == src2.vsize());
```

```
  return (matrix_type(src1) -= src2);
 }
 // перемножение двух матриц
 friend
 matrix_type operator *(
  const matrix_type &src1,
  const matrix_type &src2)
 {
  assert(src1.hsize() == src2.vsize());
  matrix_type mtx(src1.vsize(), src2.hsize());
  #pragma omp parallel for
  for (int i = 0; i < mtx.vsize(); ++i)
  {
   for (int j = 0; j < mtx.hsize(); ++j)
   {
    element_type sum(0);
    for (int k = 0; k < src1.hsize(); ++k)
     sum += src1(i, k) * src2(k, j);
    mtx(i, j) = sum;
   };
  };
  return mtx;
 }
};

// вектор - матрица в один столбец
template <typename e_t>
class vector_type: public matrix_type<e_t>
{
public:
 typedef matrix_type<e_t> base_type;
 typedef typename base_type::element_type element_type;

public:
 // конструктор (инициализация значениями element_type())
 vector_type(int vsize):
  base_type(vsize, 1)
 {}
 // конструктор - преобразование типа
 vector_type(const base_type &src):
  base_type(src.vsize(), 1)
 {
  assert(src.hsize() == 1);
  this->operator =(src);
 }

 // обращение к элементам по индексу (нумерация с нуля)
 const element_type & operator ()(int i) const
 {
  return base_type::operator ()(i, 0);
 }
 element_type & operator ()(int i)
 {
  return base_type::operator ()(i, 0);
 }
```

```
 // присваивание матрицы в один столбец
 vector_type & operator =(const base_type &src)
 {
  assert(src.hsize() == 1);
  return static_cast<vector_type &>(base_type::operator =(src));
 }
};
```

# Приложение Б.

# Классы для выполнения комплексов работ

Приводятся классы, используемые в главе 2 и реализующие построение и выполнение комплекса работ на основе переданного списка работ и зависимостей между ними. Текущий вариант представляет собой последовательную версию, распараллеленную с использованием интерфейса OpenMP.

```
// абстрактный интерфейс работы
class job_abstract_type
{
public:
 // выполнение работы
 // получение исходных данных и вывод результата
 // выполняются внутри через разделяемые ресурсы
 virtual
 void run(void) = 0;
};

// комплекс работ
class jobcomplex_type: public job_abstract_type
{
private:
 // список работ в порядке добавления
 typedef std::vector<job_abstract_type *> joblist_type;
 // набор номеров работ
 typedef std::vector<int> jobnums_type;
 // таблица зависимостей работ по номерам
 typedef std::vector<jobnums_type> deplist_type;
 // номера работ в ярусно-параллельной форме
 typedef std::vector<jobnums_type> multilevel_type;

 // ========================================
 // описание содержимого комплекса работ
 // ========================================

 // множество работ с отображением на их номера
 typedef std::map<job_abstract_type *, int> jobs_type;
 jobs_type m_jobs;
 // множество зависимостей работ
 typedef std::set<std::pair<int, int> > deps_type;
```

```
 deps_type m_deps;

private:
 // функции получения списков работ и зависимостей
 joblist_type get_joblist(void) const
 {
  // заполняем список работ
  joblist_type joblist(m_jobs.size());
  jobs_type::const_iterator it;
  for (it = m_jobs.begin(); it != m_jobs.end(); ++it)
   joblist[it->second] = it->first;
  return joblist;
 }
 deplist_type get_deplist(void) const
 {
  // строим таблицу зависимостей работ
  deplist_type deplist(m_jobs.size());
  deps_type::const_iterator it;
  for (it = m_deps.begin(); it != m_deps.end(); ++it)
   deplist[it->first].push_back(it->second);
  return deplist;
 }

 // построение ярусно-параллельной структуры
 // на основе зависимостей работ между собой
 static
 multilevel_type build(const deplist_type &deplist)
 {
  enum { NOLEVEL = -1 };
  // номера ярусов всех работ, индексированные по их номерам
  typedef std::vector<int> levnums_type;
  levnums_type levnums(deplist.size(), NOLEVEL);

  // набор номеров работ, ярусы которых еще не определены
  jobnums_type nondetermined;
  levnums_type::iterator lt;
  for (lt = levnums.begin(); lt != levnums.end(); ++lt)
   nondetermined.push_back(std::distance(levnums.begin(), lt));

  // цикл определения ярусов работ
  while (!nondetermined.empty())
  {
   jobnums_type determined;
   // пройдемся по всем неопределенным работам
   jobnums_type::iterator it;
   for (it = nondetermined.begin(); it != nondetermined.end(); ++it)
   {
    // если зависимостей нет - нулевой ярус
    int lev = 0;
    // пройдемся по всем зависимостям, если есть
    deplist_type::value_type::const_iterator jt;
    for (jt = deplist[*it].begin(); jt != deplist[*it].end(); ++jt)
    {
     // если ярус зависимости еще не определен
     if (levnums[*jt] == NOLEVEL)
```

```
      {
       // текущая работа также остается неопределенной
       lev = NOLEVEL;
       break;
      }
      else
       // иначе ярус на единицу больше максимального
       lev = std::max(lev, levnums[*jt] + 1);
     };
     // если ярус определили, добавляем в определенные
     if (lev != NOLEVEL)
     {
      determined.push_back(*it);
      levnums[*it] = lev;
     };
    };

    // если ничего не удалось определить,
    // видимо, у нас циклические зависимости
    assert(!determined.empty());

    // выкинем из неопределенных те, что определили
    jobnums_type diff;
    std::set_difference(
     nondetermined.begin(), nondetermined.end(),
     determined.begin(), determined.end(),
     std::back_inserter(diff));
    nondetermined.swap(diff);
   };

   // получим высоту ярусно-параллельной структуры
   int height = *std::max_element(levnums.begin(), levnums.end()) + 1;

   // построим ярусно-параллельную структуру
   // из всех номеров работ на основе вычисленных ярусов
   multilevel_type multilevel(height);
   for (lt = levnums.begin(); lt != levnums.end(); ++lt)
    multilevel[*lt].push_back(std::distance(levnums.begin(), lt));

   return multilevel;
  }

public:

  // добавление работы в комплекс
  void add_job(job_abstract_type &job)
  {
   // добавляемой работы не должно быть в списке
   assert(m_jobs.find(&job) == m_jobs.end());
   int idx = m_jobs.size();
   m_jobs.insert(jobs_type::value_type(&job, idx));
  }

  // добавление зависимости одной работы от другой
  void add_dependence(
```

```
  job_abstract_type &dst,
  job_abstract_type &src)
 {
  // работы должны быть различными
  assert(&src != &dst);
  // обе работы уже должны быть в списке
  assert(m_jobs.find(&dst) != m_jobs.end());
  assert(m_jobs.find(&src) != m_jobs.end());
  m_deps.insert(deps_type::value_type(m_jobs[&dst], m_jobs[&src]));
 }

 // выполнение всего комплекса работ
 void run(void)
 {
  // получим список подлежащих выполнению работ
  joblist_type joblist = get_joblist();
  // построим ярусно-параллельную структуру номеров работ
  multilevel_type multilevel = build(get_deplist());

  multilevel_type::iterator it;
  // выполним по очереди каждый ярус работ
  for (it = multilevel.begin(); it != multilevel.end(); ++it)
  {
   int width = it->size();
   #pragma omp parallel for
   for (int i = 0; i < width; ++i)
    joblist[(*it)[i]]->run();
  };
 }
};
```

# Приложение В.

# Классы для выполнения сетей конечных автоматов

Приводится последовательная версия классов, используемых в главе 3 для построения и выполнения автоматных сетей, содержащая директивы OpenMP для распараллеливания работы автоматов на каждом такте.

```
// абстрактный тип автомата
class fsm_abstract_type
{
public:
 // тип состояния автомата
 typedef int state_type;
 // тип входных и выходных данных
 typedef int signal_type;
 // полный набор входных или выходных сигналов
 typedef std::vector<signal_type> signals_type;

 // количество входов
 virtual
 int number_input(void) const = 0;
 // количество выходов
 virtual
 int number_output(void) const = 0;
 // передать автомату вектор входных данных
 virtual
 void put_input(const signals_type &input) = 0;
 // выполнить такт работы автомата
 virtual
 void do_work(void) = 0;
 // получить от автомата вектор выходных данных
 virtual
 signals_type get_output(void) const = 0;
 // проверка, находится ли автомат в начальном состоянии
 virtual
 bool is_off(void) const = 0;
};

// конечный автомат
class fsm_type: public fsm_abstract_type
{
```

```
protected:
 // обработчик, вызываемый из do_work в конкретном состоянии
 typedef state_type (fsm_type::*handler_type)(state_type state);
 // начальное/конечное состояние
 enum { STATE_OFF = -1 };

private:
 // текущее состояние
 state_type m_state;
 // таблица обработчиков
 typedef std::map<state_type, handler_type> handlertable_type;
 handlertable_type m_handlertable;

protected:
 // входные и выходные сигналы автомата на текущем такте
 signals_type m_input, m_output;

 void add_handler(state_type state, handler_type handler)
 {
  m_handlertable[state] = handler;
 }

public:
 fsm_type(int inputnum, int outputnum):
  m_state(STATE_OFF),
  m_input(inputnum, 0), m_output(outputnum, 0)
 {}

 int number_input(void) const
 {
  return m_input.size();
 }

 int number_output(void) const
 {
  return m_output.size();
 }

 void put_input(const signals_type &input)
 {
  assert(input.size() == m_input.size());
  m_input.assign(input.begin(), input.end());
 }

 void do_work(void)
 {
  signals_type::size_type insize = m_input.size();
  signals_type::size_type outsize = m_output.size();

  // найти обработчик текущего состояния и вызвать его
  handlertable_type::iterator it = m_handlertable.find(m_state);
  assert(it != m_handlertable.end());
  m_state = (this->*(it->second))(m_state);

  assert(insize == m_input.size());
```

```
  assert(outsize == m_output.size());

  // если автомат выключился, обнулить выходы
  if (m_state == STATE_OFF)
   m_output.assign(m_output.size(), 0);
 }

 signals_type get_output(void) const
 {
  return m_output;
 }

 bool is_off(void) const
 {
  // проверка завершения работы
  return m_state == STATE_OFF;
 }
};

// сеть конечных автоматов
class fsmnet_type: public fsm_abstract_type
{
private:
 // список связей автоматов по входам и выходам
 // в форме (№ автомата, № выхода) -> (№ автомата, № входа)
 typedef std::list<
  std::pair<std::pair<int, int>, std::pair<int, int> >
 > linklist_type;
 // два псевдономера для обозначения входа и выхода сети
 enum { PSEUDOFSM_NETINPUT = -1, PSEUDOFSM_NETOUTPUT = -2 };

 // класс для работы с общими данными
 class shared_area_type
 {
 private:
  enum { NOPOS = -1 };
  // общие данные (выходы всех автоматов и входы сети)
  signals_type m_data;
  // позиции начал и размеры областей выходов автоматов и входов сети
  std::vector<int> m_outpos, m_outsize;
  // списки позиций входов автоматов и выходов сети
  std::vector<std::vector<int> > m_allinpos;

 public:
  // в конструкторе на основе входящего списка связей формируем
  // область общих данных, список позиций выходных областей
  // автоматов, а также списки позиций конкретных входов автоматов
  shared_area_type(const linklist_type &linklist)
  {
   linklist_type::const_iterator it;
   // полное количество автоматов = максимальный указанный + 1
   int fsmnum = -1;
   for (it = linklist.begin(); it != linklist.end(); ++it)
   {
    fsmnum = std::max(fsmnum, it->first.first);
```

```
 fsmnum = std::max(fsmnum, it->second.first);
};
++fsmnum;

// количества выходов автоматов (сначала) и входов сети (последний)
std::vector<int> outsize(fsmnum + 1);
// количества входов автоматов (сначала) и выходов сети (последний)
std::vector<int> insize(fsmnum + 1);
// ищем максимальные номера входов и выходов для всех
for (it = linklist.begin(); it != linklist.end(); ++it)
{
 int idxout = it->first.first;
 int idxin = it->second.first;
 idxout = (idxout != PSEUDOFSM_NETINPUT) ? idxout : fsmnum;
 idxin = (idxin != PSEUDOFSM_NETOUTPUT) ? idxin : fsmnum;

 assert(it->first.first != PSEUDOFSM_NETOUTPUT);
 assert(it->second.first != PSEUDOFSM_NETINPUT);
 outsize[idxout] = std::max(outsize[idxout], it->first.second);
 insize[idxin] = std::max(insize[idxin], it->second.second);
};
// корректируем до размеров (добавляем по единице)
std::transform(
 outsize.begin(), outsize.end(),
 outsize.begin(), std::bind2nd(std::plus<int>(), 1));
std::transform(
 insize.begin(), insize.end(),
 insize.begin(), std::bind2nd(std::plus<int>(), 1));

// формируем список позиций начал областей выходов
// первая область по нулевому смещению
m_outpos.assign(1, 0);
// смещения остальных - частичные суммы размеров областей
std::partial_sum(
 outsize.begin(), outsize.end(),
 std::back_inserter(m_outpos));
// последний (лишний) элемент - полное количество всех выходов
m_data.assign(m_outpos.back(), 0);
m_outpos.pop_back();
// размеры выходных областей в массиве общих данных
m_outsize.swap(outsize);

// создаем списки позиций входов
m_allinpos.resize(fsmnum + 1);
for (int i = 0; i < fsmnum + 1; ++i)
 m_allinpos[i].resize(insize[i], NOPOS);
// заполняем списки позиций входов
for (it = linklist.begin(); it != linklist.end(); ++it)
{
 int idxout = it->first.first;
 int idxin = it->second.first;
 idxout = (idxout != PSEUDOFSM_NETINPUT) ? idxout : fsmnum;
 idxin = (idxin != PSEUDOFSM_NETOUTPUT) ? idxin : fsmnum;

 // каждый вход может быть подключен лишь к одному выходу
```

```
    assert(m_allinpos[idxin][it->second.second] == NOPOS);
    m_allinpos[idxin][it->second.second] =
     m_outpos[idxout] + it->first.second;
   };
  }
  // сохранение выходных данных конкретного автомата
  void put_output(int i, const signals_type &output)
  {
   assert(size_t(m_outsize[i]) == output.size());
   std::copy(
    output.begin(), output.end(),
    m_data.begin() + m_outpos[i]);
  }
  // сохранение входных данных сети
  void put_input(const signals_type &input)
  {
   assert(size_t(m_outsize.back()) == input.size());
   std::copy(
    input.begin(), input.end(),
    m_data.begin() + m_outpos.back());
  }
  // восстановление входных данных конкретного автомата
  signals_type get_input(int i) const
  {
   int signum = m_allinpos[i].size();
   signals_type input(signum);
   for (int j = 0; j < signum; ++j)
   {
    assert(m_allinpos[i][j] != NOPOS);
    input[j] = m_data[m_allinpos[i][j]];
   };
   return input;
  }
  // восстановление выходных данных сети
  signals_type get_output(void) const
  {
   int signum = m_allinpos.back().size();
   signals_type output(signum);
   for (int j = 0; j < signum; ++j)
   {
    assert(m_allinpos.back()[j] != NOPOS);
    output[j] = m_data[m_allinpos.back()[j]];
   };
   return output;
  }
 };

public:
 // абстрактный тип фабрики автоматной сети
 class factory_abstract_type
 {
 public:
  // хранилище связей (для упрощения их добавления)
  class links_type
  {
```

```
  private:
   linklist_type m_linklist;
  public:
   void fsm_to_fsm(int srcfsm, int srcnum, int dstfsm, int dstnum)
   {
    m_linklist.push_back(linklist_type::value_type(
     linklist_type::value_type::first_type(srcfsm, srcnum),
     linklist_type::value_type::second_type(dstfsm, dstnum)));
   }
   void input_to_fsm(int srcnum, int dstfsm, int dstnum)
   {
    fsm_to_fsm(PSEUDOFSM_NETINPUT, srcnum, dstfsm, dstnum);
   }
   void fsm_to_output(int srcfsm, int srcnum, int dstnum)
   {
    fsm_to_fsm(srcfsm, srcnum, PSEUDOFSM_NETOUTPUT, dstnum);
   }
   linklist_type get(void) const
   {
    return m_linklist;
   }
  };

  // количество автоматов в сети
  virtual
  int number_fsm(void) const = 0;
  // количество входов сети
  virtual
  int number_input(void) const = 0;
  // количество выходов сети
  virtual
  int number_output(void) const = 0;
  // получение списка всех связей
  virtual
  links_type links(void) const = 0;
  // создание очередного автомата в сети
  virtual
  fsm_abstract_type *create_fsm(int id) = 0;
  // уничтожение его же
  virtual
  void destroy_fsm(int id, fsm_abstract_type *pfsm) = 0;
 };

private:

 // список активных автоматов
 std::vector<fsm_abstract_type *> m_pfsm;
 // фабрика сети
 factory_abstract_type &m_factory;
 // общая область памяти
 shared_area_type m_shared;

public:
 fsmnet_type(factory_abstract_type &factory):
  m_factory(factory), m_shared(m_factory.links().get())
```

```
{
 // создание автоматов
 m_pfsm.resize(m_factory.number_fsm());
 #pragma omp parallel for
 for (int i = 0; i < m_factory.number_fsm(); ++i)
  m_pfsm[i] = m_factory.create_fsm(i);
}

~fsmnet_type(void)
{
 // уничтожение автоматов
 #pragma omp parallel for
 for (int i = m_factory.number_fsm() - 1; i >= 0; --i)
  m_factory.destroy_fsm(i, m_pfsm[i]);
}

int number_input(void) const
{
 return m_factory.number_input();
}

int number_output(void) const
{
 return m_factory.number_output();
}

void put_input(const signals_type &input)
{
 m_shared.put_input(input);
}

void do_work(void)
{
 // прочитать входные данные, выполнить действия
 #pragma omp parallel
 {
  #pragma omp for
  for (int i = 0; i < m_factory.number_fsm(); ++i)
  {
   m_pfsm[i]->put_input(m_shared.get_input(i));
   m_pfsm[i]->do_work();
  };
  // записать выходные (запись должна быть отделена от чтения)
  #pragma omp for
  for (int i = 0; i < m_factory.number_fsm(); ++i)
   m_shared.put_output(i, m_pfsm[i]->get_output());
 };
}

signals_type get_output(void) const
{
 return m_shared.get_output();
}

bool is_off(void) const
```

```
 {
  // сеть в выключенном состоянии, когда все автоматы выключены
  bool rc = true;
  #pragma omp parallel for reduction(&&: rc)
  for (int i = 0; i < m_factory.number_fsm(); ++i)
   rc = (rc && m_pfsm[i]->is_off());
  return rc;
 }
};
```

# Приложение Г.

# Классы для выполнения сетей Петри

Ниже приводится набор классов, предоставляющий возможность построения и выполнения рассмотренных в главе 4 сетей Петри и строго иерархических сетей. Там же рассматриваются вопросы распараллеливания программ, построенных на основе таких классов.

```
// пустой тип для обозначения позиции
class place_type {};

// абстрактный тип перехода
class transition_abstract_type
{
public:
 // список разрешенных переходов
 typedef std::vector<transition_abstract_type *> enabledlist_type;
 // список помеченных позиций
 typedef std::map<place_type *, int> markedlist_type;

 // активация перехода
 virtual
 void activate(void) = 0;
 // получение информации, активен ли переход
 virtual
 bool is_active(void) const = 0;
 // получение списка внутренних разрешенных переходов
 // (только если текущий переход активен)
 virtual
 enabledlist_type get_enabled(void) const = 0;
 // получение списка внутренних помеченных позиций
 // (только если текущий переход активен)
 virtual
 markedlist_type get_marked(void) const = 0;
 // срабатывание одного из разрешенных внутренних переходов
 // (только если текущий переход активен)
 virtual
 void fire(int number) = 0;

 // обработчики событий
 virtual
 void on_activate(void) {}
 virtual
 void on_passivate(void) {}
};
```

```
// простой (атомарный) переход
class transition_simple_type: public transition_abstract_type
{
public:
 void activate(void)
 {}
 bool is_active(void) const
 { return false; }
 enabledlist_type get_enabled(void) const
 { return enabledlist_type(); }
 markedlist_type get_marked(void) const
 { return markedlist_type(); }
 void fire(int)
 { assert(false); }
};

// составной переход (вложенная сеть Петри)
class transition_compound_type: public transition_abstract_type
{
private:
 typedef std::vector<place_type *> placelist_type;
 typedef std::vector<transition_abstract_type *> transitionlist_type;
 typedef std::vector<int> marking_type;
 typedef std::vector<marking_type> arcmatrix_type;

public:
 // содержимое сети Петри
 class content_type
 {
 private:
  typedef std::map<place_type *, int> plmap_type;
  typedef std::map<transition_abstract_type *, int> trmap_type;
  typedef std::map<std::pair<int, int>, int> arcmap_type;
  typedef std::map<int, int> tokmap_type;

  // отображение позиций на их номера
  plmap_type m_plmap;
  // отображение переходов на их номера
  trmap_type m_trmap;
  // отображения входных и выходных дуг на их веса
  arcmap_type m_inmap, m_outmap;
  // отображение номеров позиций на количество фишек
  tokmap_type m_tokmap;

 private:
  // добавление дуги некоторой кратности
  void add_arc(
   place_type &pl,
   transition_abstract_type &tr,
   int weight,
   arcmap_type &arcmap)
  {
   assert(m_plmap.find(&pl) != m_plmap.end());
   assert(m_trmap.find(&tr) != m_trmap.end());
```

```
  assert(weight > 0);
  arcmap_type::key_type arc(m_trmap[&tr], m_plmap[&pl]);
  arcmap[arc] += weight;
 }
 // построение матрицы кратности дуг
 arcmatrix_type build_matrix(const arcmap_type &arcmap) const
 {
  arcmatrix_type mtx(m_trmap.size());
  arcmatrix_type::iterator it;
  for (it = mtx.begin(); it != mtx.end(); ++it)
  {
   arcmatrix_type::value_type vec(m_plmap.size());
   arcmatrix_type::value_type::iterator jt;
   for (jt = vec.begin(); jt != vec.end(); ++jt)
   {
    arcmap_type::const_iterator ft;
    ft = arcmap.find(arcmap_type::key_type(
     std::distance(mtx.begin(), it),
     std::distance(vec.begin(), jt)));
    *jt = (ft != arcmap.end()) ? ft->second : 0;
   };
   *it = vec;
  };
  return mtx;
 }

public:

 // ——————— функции, используемые при построении сети ———————

 // добавление позиции
 void add_place(place_type &pl)
 {
  assert(m_plmap.find(&pl) == m_plmap.end());
  m_plmap.insert(plmap_type::value_type(&pl, m_plmap.size()));
 }

 // добавление перехода
 void add_transition(transition_abstract_type &tr)
 {
  assert(m_trmap.find(&tr) == m_trmap.end());
  m_trmap.insert(trmap_type::value_type(&tr, m_trmap.size()));
 }

 // добавление входной дуги
 void add_arc(
  place_type &in,
  transition_abstract_type &tr,
  int weight = 1)
 {
  add_arc(in, tr, weight, m_inmap);
 }
 // добавление выходной дуги
 void add_arc(
  transition_abstract_type &tr,
```

```
 place_type &out,
 int weight = 1)
{
 add_arc(out, tr, weight, m_outmap);
}

// добавление некоторого количества фишек к позиции
void add_token(place_type &pl, int tokens = 1)
{
 assert(tokens > 0);
 plmap_type::iterator it = m_plmap.find(&pl);
 assert(it != m_plmap.end());
 m_tokmap[it->second] += tokens;
}

// ------- функции, используемые сетью -------

// получение списка позиций
placelist_type get_pllist(void) const
{
 placelist_type pllist(m_plmap.size());
 plmap_type::const_iterator it;
 for (it = m_plmap.begin(); it != m_plmap.end(); ++it)
  pllist[it->second] = it->first;
 return pllist;
}

// получение списка переходов
transitionlist_type get_trlist(void) const
{
 transitionlist_type trlist(m_trmap.size());
 trmap_type::const_iterator it;
 for (it = m_trmap.begin(); it != m_trmap.end(); ++it)
  trlist[it->second] = it->first;
 return trlist;
}

// получение матриц входных и выходных дуг
arcmatrix_type get_inmatrix(void) const
{
 return build_matrix(m_inmap);
}
arcmatrix_type get_outmatrix(void) const
{
 return build_matrix(m_outmap);
}

// получение начальной разметки
marking_type get_marking(void) const
{
 marking_type marking(m_plmap.size(), 0);
 tokmap_type::const_iterator it;
 for (it = m_tokmap.begin(); it != m_tokmap.end(); ++it)
  marking[it->first] = it->second;
 return marking;
```

```
  }
 };

private:
 // список позиций
 placelist_type m_pllist;
 // список переходов
 transitionlist_type m_trlist;
 // матрицы входных и выходных дуг
 arcmatrix_type m_mtxin;
 arcmatrix_type m_mtxout;
 // начальная разметка
 marking_type m_marking_init;

 // текущая разметка
 marking_type m_marking;
 // полный список помеченных позиций, включая внутренние
 markedlist_type m_marked;
 // полный список разрешенных переходов, включая внутренние
 enabledlist_type m_enabled;
 // отображение элементов m_enabled на локальные переходы
 std::vector<int> m_location;
 // смещение в m_enabled области каждого локального перехода
 std::vector<int> m_offset;

 // количество локальных позиций
 int pl_num(void) const
 {
  return m_pllist.size();
 }
 // количество локальных переходов
 int tr_num(void) const
 {
  return m_trlist.size();
 }

 // обновление списков разрешенных переходов и помеченных позиций
 void refresh(void)
 {
  // соберем все разрешенные переходы и их размещение
  m_enabled.clear();
  m_location.clear();
  m_offset.resize(tr_num());
  for (int i = 0; i < tr_num(); ++i)
  {
   // формирование в m_enabled области i-го локального перехода
   m_offset[i] = m_enabled.size();

   // если переход активен
   if (m_trlist[i]->is_active())
   {
    // получим внутренние разрешенные переходы
    enabledlist_type inner = m_trlist[i]->get_enabled();
    // добавим их в конец текущего списка
    m_enabled.insert(
```

```
      m_enabled.end(), inner.begin(), inner.end());
     // все они размещаются в i-м локальном переходе
     m_location.insert(m_location.end(), inner.size(), i);
    }
    else
    {
     // если не активен, выясним, разрешен ли
     bool isenabled = true;
     for (int j = 0; isenabled && j < pl_num(); ++j)
      isenabled = (m_marking[j] >= m_mtxin[i][j]);
     // и добавим его, если разрешен
     if (isenabled)
     {
      m_enabled.push_back(m_trlist[i]);
      m_location.push_back(i);
     };
    };
   };

   // соберем все помеченные позиции
   m_marked.clear();
   // локальные позиции
   for (int j = 0; j < pl_num(); ++j)
    if (m_marking[j] > 0)
     m_marked[m_pllist[j]] = m_marking[j];
   // внутренние позиции локальных активных переходов
   for (int i = 0; i < tr_num(); ++i)
   {
    if (m_trlist[i]->is_active())
    {
     markedlist_type inner = m_trlist[i]->get_marked();
     m_marked.insert(inner.begin(), inner.end());
    };
   };
  }

public:
 explicit
 transition_compound_type(const content_type &content):
  m_pllist(content.get_pllist()),
  m_trlist(content.get_trlist()),
  m_mtxin(content.get_inmatrix()),
  m_mtxout(content.get_outmatrix()),
  m_marking_init(content.get_marking())
 {}

 void activate(void)
 {
  m_marking = m_marking_init;
  refresh();
 }
 bool is_active(void) const
 {
  return !m_enabled.empty();
 }
```

```
 enabledlist_type get_enabled(void) const
 {
  return m_enabled;
 }
 markedlist_type get_marked(void) const
 {
  return m_marked;
 }
 void fire(int number)
 {
  assert(number >= 0 && size_t(number) < m_enabled.size());

  // найдем размещение срабатывающего перехода в локальных
  int local = m_location[number];
  // и внутренний номер (если найденный локальный активен)
  int lower = number - m_offset[local];

  // если переход активен, выполним внутренний
  if (m_trlist[local]->is_active())
   m_trlist[local]->fire(lower);
  else
  {
   // иначе изымем входные фишки
   for (int j = 0; j < pl_num(); ++j)
    m_marking[j] -= m_mtxin[local][j];
   // активируем переход
   m_trlist[local]->on_activate();
   m_trlist[local]->activate();
  };
  // если переход перестал быть активным
  if (!m_trlist[local]->is_active())
  {
   m_trlist[local]->on_passivate();
   // выложим выходные фишки
   for (int j = 0; j < pl_num(); ++j)
    m_marking[j] += m_mtxout[local][j];
  };

  // обновим состояние
  refresh();
 }
};

// сеть Петри верхнего уровня
class petrinet_type: public transition_compound_type
{
public:
 // абстрактный тип среды, в которой "живет" сеть Петри
 class environment_abstract_type
 {
 public:
  // ожидание срабатывания одного из переходов
  // передаются списки разрешенных переходов и помеченных позиций
  virtual
  int wait(
```

```
    const enabledlist_type &enabled,
    const markedlist_type &marked) = 0;
 };

public:
 petrinet_type(const content_type &content):
  transition_compound_type(content)
 {}
 // полный жизненный цикл сети Петри
 void live(environment_abstract_type &env)
 {
  activate();
  while (is_active())
   fire(env.wait(get_enabled(), get_marked()));
 }
};
```

# Приложение Д.

# Классы для выполнения систем актеров

Приводится набор классов, используемых при реализации программ на основе модели актеров. Модель актеров рассматривается в главе 5. Для использования клиентским кодом предназначены классы `actor_type` и `scheduler_type`. Работа системы актеров реализована последовательно в рамках класса `scheduler_type`, выполнено распараллеливание на основе директив OpenMP.

```
// генерация псевдослучайного числа в интервале [0, bound)
inline int random(int bound)
{
 assert(bound > 0);
 return int(rand() / (RAND_MAX + 1.0) * bound);
}

class arbiter_type;
class factory_type;
class scheduler_type;

// почтовый адрес актера
typedef int address_type;

// базовый класс актера
class actor_type
{
private:
 // === внутренние типы и общие функции для всех классов ===
 friend class factory_type;
 friend class arbiter_type;
 friend class scheduler_type;

 // идентификатор поведения актера
 typedef std::string defid_type;
 // идентификатор образца сообщения
 typedef std::string patid_type;
 // параметры создания актера или сообщения,
 // преобразованные к массиву байтов
 typedef std::vector<char> chunk_type;
```

```
// структура поведения актера
struct behaviour_type
{
 // идентификатор поведения
 defid_type defid;
 // параметры создания актера
 chunk_type chunk;
};

// структура сообщения
struct task_type
{
 // адрес назначения
 address_type address;
 // идентификатор образца
 patid_type patid;
 // параметры сообщения
 chunk_type chunk;
};

// преобразование структуры параметров в массив байтов и обратно
template <typename param_type>
static
chunk_type param2chunk(const param_type &p)
{
 typedef chunk_type::value_type data_type;
 assert(sizeof(param_type) % sizeof(data_type) == 0);
 const data_type &c = reinterpret_cast<const data_type &>(p);
 int size = sizeof(param_type) / sizeof(data_type);
 // читаем передаваемый параметр как массив
 return chunk_type(&c, &c + size);
}
template <typename param_type>
static
const param_type & chunk2param(const chunk_type &chunk)
{
 return reinterpret_cast<const param_type &>(chunk.front());
}

// генерация идентификатора поведения по типам
template <typename someactor_type, typename init_type>
static
defid_type def_id(void)
{
 // проверка соответствия параметров конструктора
 assert(1 ? 1 : (delete new someactor_type(init_type()), 0));
 // составной идентификатор конструктор–параметр
 return
  defid_type(typeid(someactor_type).name()) +
  "|" +
  defid_type(typeid(init_type).name());
}

// генерация идентификатора образца сообщения по его типу
template <typename pattern_type>
```

```
 static
 patid_type pat_id(void)
 {
  return patid_type(typeid(pattern_type).name());
 }

public:
 // === типы, используемые клиентским кодом ===

 // "пустой" тип параметров инициализации, используется
 // для краткой записи создания актера без параметров
 class empty_type {};
 // неопределенный тип сообщения, используется для обработки
 // сообщений, не соответствующих ни одному образцу
 class unknown_type
 {
  friend class actor_type;
  task_type task;
 };

private:
 // указатель на функцию-активатор
 typedef void (*activator_type)(actor_type *, const task_type &);
 // таблица активаторов
 typedef std::map<patid_type, activator_type> activatortable_type;
 activatortable_type m_acttable;

 // привязки актера к арбитру и планировщику
 scheduler_type *m_sched;
 arbiter_type *m_arbiter;

 // шаблон активатора обработки конкретного сообщения
 template <
  typename someactor_type,
  typename pattern_type,
  void (someactor_type::*action)(const pattern_type &)>
 static
 void activate(actor_type *thisp, const task_type &task)
 {
  someactor_type *actor = static_cast<someactor_type *>(thisp);
  (actor->*action)(chunk2param<pattern_type>(task.chunk));
 }
 // шаблон активатора обработки "прочих" сообщений
 template <
  typename someactor_type,
  void (someactor_type::*action)(const unknown_type &)>
 static
 void activate(actor_type *thisp, const task_type &task)
 {
  someactor_type *actor = static_cast<someactor_type *>(thisp);
  unknown_type unknown;
  unknown.task = task;
  (actor->*action)(unknown);
 }
```

```
 // функции инстанцирования нужного активатора
 // на основе переданных типов
 template <typename someactor_type, typename pattern_type>
 activator_type get_activator(const someactor_type *, const pattern_type *)
 {
  return &actor_type::template
   activate<someactor_type, pattern_type, &someactor_type::action>;
 }
 template <typename someactor_type>
 activator_type get_activator(const someactor_type *, const unknown_type *)
 {
  return &actor_type::template
   activate<someactor_type, &someactor_type::action>;
 }

 // тела этих функций описаны после арбитра и планировщика
 inline address_type raw_create(const behaviour_type &behaviour) const;
 inline void raw_become(const behaviour_type &behaviour) const;
 inline void raw_send(const task_type &task) const;
 inline address_type raw_self(void) const;

 // === интерфейс для arbiter_type и scheduler_type ===
 // привязка только что созданного актера
 void bind(scheduler_type *sched, arbiter_type *arbiter)
 {
  m_sched = sched;
  m_arbiter = arbiter;
 }
 // проверка наличия активатора по сообщению
 bool match(const task_type &task) const
 {
  return
   m_acttable.find(task.patid) != m_acttable.end() ||
   m_acttable.find(pat_id<unknown_type>()) != m_acttable.end();
 }
 // запуск активатора по сообщению
 void apply(const task_type &task)
 {
  assert(task.address == self());
  activatortable_type::iterator it;
  // найдем обработчик по образцу
  it = m_acttable.find(task.patid);
  // если нет, поищем обработчик по умолчанию
  if (it == m_acttable.end())
   it = m_acttable.find(pat_id<unknown_type>());
  // обработчик должен быть найден, запустим его
  assert(it != m_acttable.end());
  it->second(this, task);
 }

protected:
 // === интерфейс для клиентского кода ===

 // конструктор
 actor_type(void):
```

```
  m_sched(NULL), m_arbiter(NULL)
 {}

 // добавление обработчика сообщения заданного образца
 template <typename someactor_type, typename pattern_type>
 void add_action(void)
 {
  patid_type patid = pat_id<pattern_type>();
  // такого обработчика пока не должно быть
  assert(m_acttable.find(patid) == m_acttable.end());
  // получим обработчик на основе текущих типов
  m_acttable[patid] = get_activator(
   static_cast<const someactor_type *>(NULL),
   static_cast<const pattern_type *>(NULL));
 }

 // выполнение операции become с параметрами
 template <typename someactor_type, typename init_type>
 void become(const init_type &init) const
 {
  assert(m_arbiter != NULL);
  behaviour_type behaviour =
  {
   def_id<someactor_type, init_type>(),
   param2chunk(init)
  };
  raw_become(behaviour);
 }
 // то же без параметров
 template <typename someactor_type>
 void become(void) const
 {
  become<someactor_type>(empty_type());
 }

 // получение адреса текущего актера
 address_type self(void) const
 {
  return raw_self();
 }

public:
 // выполнение операции create с параметрами
 template <typename someactor_type, typename init_type>
 address_type create(const init_type &init) const
 {
  behaviour_type behaviour =
  {
   def_id<someactor_type, init_type>(),
   param2chunk(init)
  };
  return raw_create(behaviour);
 }
 // то же без параметров
 template <typename someactor_type>
```

```
 address_type create(void) const
 {
  return create<someactor_type>(empty_type());
 }

 // выполнение отправки некоторого сообщения
 template <typename pattern_type>
 void send(const address_type &address, const pattern_type &msg) const
 {
  task_type task =
  {
   address,
   pat_id<pattern_type>(),
   param2chunk(msg)
  };
  raw_send(task);
 }
 // перенаправление "прочего" сообщения
 void send(const address_type &address, const unknown_type &msg) const
 {
  task_type task = msg.task;
  task.address = address;
  raw_send(task);
 }
};

// класс фабрики актеров
class factory_type
{
private:
 typedef actor_type::defid_type defid_type;
 typedef actor_type::chunk_type chunk_type;
 typedef actor_type::behaviour_type behaviour_type;

 // тип указателя на функцию создания актера
 typedef actor_type *(*creator_type)(const chunk_type &chunk);
 // тип указателя на функцию уничтожения актера
 typedef void (*destroyer_type)(actor_type *actor);

 typedef std::map<defid_type, creator_type> creatortable_type;
 typedef std::map<defid_type, destroyer_type> destroyertable_type;
 creatortable_type m_ctortable;
 destroyertable_type m_dtortable;

 // функции создания и уничтожения актеров заданного типа
 template <typename someactor_type, typename init_type>
 static
 actor_type *construct(const chunk_type &chunk)
 {
  return new someactor_type(actor_type::chunk2param<init_type>(chunk));
 }
 template <typename someactor_type>
 static
 void destruct(actor_type *actor)
 {
```

```
  delete static_cast<someactor_type *>(actor);
 }

 // === интерфейс для arbiter_type ===
 friend class arbiter_type;
 // создание актера
 actor_type *create_actor(const behaviour_type &behaviour) const
 {
  // ищем функцию создания актера с заданным поведением
  creatortable_type::const_iterator it;
  it = m_ctortable.find(behaviour.defid);
  // убедимся, что поведение было внесено в фабрику
  assert(it != m_ctortable.end());
  // создаем актера с заданными параметрами
  actor_type *actor = (it->second)(behaviour.chunk);
  return actor;
 }
 void destroy_actor(
  const behaviour_type &behaviour,
  actor_type *actor) const
 {
  // ищем по заданному поведению функцию уничтожения
  destroyertable_type::const_iterator it;
  it = m_dtortable.find(behaviour.defid);
  // убедимся, что поведение было внесено в фабрику
  assert(it != m_dtortable.end());
  // и уничтожаем актера
  (it->second)(actor);
 }

public:
 // === интерфейс для клиентского кода ===

 // добавление поведения в фабрику
 template <typename someactor_type, typename init_type>
 void add_definition(void)
 {
  // инстанцируем функции создания и уничтожения актера
  // и запоминаем их в соответствии с поведением
  defid_type defid = actor_type::def_id<someactor_type, init_type>();
  m_ctortable[defid] = &factory_type::template
   construct<someactor_type, init_type>;
  m_dtortable[defid] = &factory_type::template
   destruct<someactor_type>;
 }
};

// арбитр, клиентскому коду интерфейса не предоставляет
class arbiter_type
{
private:
 typedef actor_type::behaviour_type behaviour_type;
 typedef actor_type::task_type task_type;
 typedef std::list<task_type> mailbox_type;
```

```
// фабрика для создания и уничтожения актеров
const factory_type &m_factory;
// личный адрес
address_type m_self;
// планировщик текущего арбитра
scheduler_type *m_sched;
// список замещающих поведений
std::list<behaviour_type> m_behaviour;
// текущий актер
actor_type *m_actor;
// почтовый ящик
mailbox_type m_mailbox;

// === интерфейс для actor_type ===
friend class actor_type;
// задание нового поведения
void new_behaviour(const behaviour_type &behaviour)
{
 // в момент вызова должен быть ровно один элемент
 assert(m_behaviour.size() == 1);
 m_behaviour.push_back(behaviour);
}
// получение адреса почтового ящика
const address_type & address(void) const
{
 return m_self;
}

// === интерфейс для scheduler_type ===
friend class scheduler_type;
// конструктор
arbiter_type(
 const factory_type &factory,
 const address_type &self,
 const behaviour_type &behaviour,
 scheduler_type *sched):
 m_factory(factory), m_self(self), m_sched(sched)
{
 // запомним начальное поведение
 m_behaviour.push_back(behaviour);
 // создадим актера
 m_actor = m_factory.create_actor(m_behaviour.front());
 // привяжем к арбитру и планировщику
 m_actor->bind(m_sched, this);
}
// деструктор
~arbiter_type(void)
{
 m_factory.destroy_actor(m_behaviour.front(), m_actor);
}
// помещение сообщения в почтовый ящик
void deliver(const task_type &task)
{
 m_mailbox.push_back(task);
}
```

```
 // проверка почтового ящика на пустоту
 bool empty(void) const
 {
  return m_mailbox.empty();
 }
 // извлечение очередного сообщения из ящика
 bool retrieve(task_type &task)
 {
  bool rc;
  if ((rc = !m_mailbox.empty()))
  {
   // внесение элемента недетерминированности
   mailbox_type::iterator it = m_mailbox.begin();
   advance(it, random(m_mailbox.size()));
   // проверка соответствия сообщения образцам
   if ((rc = m_actor->match(*it)))
   {
    // извлечение и удаление сообщения из ящика
    task = *it;
    m_mailbox.erase(it);
   };
  };
  return rc;
 }
 // обработка сообщения
 void process(const task_type &task)
 {
  // обработка сообщения
  m_actor->apply(task);
  // если было задано новое поведение
  if (m_behaviour.size() > 1)
  {
   // уничтожаем актера
   m_factory.destroy_actor(m_behaviour.front(), m_actor);
   // удаляем текущее поведение
   while (m_behaviour.size() > 1)
    m_behaviour.pop_front();
   // создаем и привязываем нового актера
   m_actor = m_factory.create_actor(m_behaviour.front());
   m_actor->bind(m_sched, this);
  };
 }
};

// класс планировщика
class scheduler_type
{
private:
 typedef actor_type::behaviour_type behaviour_type;
 typedef actor_type::task_type task_type;

 typedef std::vector<arbiter_type *> arbiterlist_type;
 typedef std::list<
  std::pair<address_type, behaviour_type>
 > actorqueue_type;
```

```
 typedef std::list<task_type> taskqueue_type;

 // псевдо-актер, "точка отсчета" для системы актеров
 actor_type m_origin;
 // очередь запросов на создание актеров
 actorqueue_type m_actorqueue;
 // очередь запросов на доставку сообщений
 taskqueue_type m_taskqueue;
 // список созданных арбитров
 arbiterlist_type m_arbiterlist;

 // === интерфейс для actor_type ===
 friend class actor_type;
 // добавление запроса на создание актера
 address_type new_actor(const behaviour_type &behaviour)
 {
  address_type newaddr;
  #pragma omp critical (creation)
  {
   // формирование нового адреса
   newaddr = m_arbiterlist.size() + m_actorqueue.size();
   // добавление запроса
   m_actorqueue.push_back(
    actorqueue_type::value_type(newaddr, behaviour));
  };
  return newaddr;
 }
 // добавление запроса на доставку сообщения
 void new_task(const task_type &task)
 {
  #pragma omp critical (sending)
  m_taskqueue.push_back(task);
 }

public:
 // === интерфейс для клиентского кода ===

 // конструктор
 scheduler_type(void)
 {
  // псевдо-актер привязан только к планировщику
  // арбитра у него нет, как и почтового адреса
  m_origin.bind(this, NULL);
 }

 // "точка отсчета" системы актеров
 const actor_type & system(void) const
 {
  return m_origin;
 }

 // полный цикл развития системы актеров
 void evolve(const factory_type &factory)
 {
  bool busy;
```

```
  do
  {
   // обработка запросов на создание актеров
   while (!m_actorqueue.empty())
   {
    // изымаем из очереди адрес и соответствующий запрос
    address_type newaddr = m_actorqueue.front().first;
    behaviour_type behaviour = m_actorqueue.front().second;
    m_actorqueue.pop_front();
    // создаем арбитра
    assert(size_t(newaddr) == m_arbiterlist.size());
    m_arbiterlist.push_back(
     new arbiter_type(factory, newaddr, behaviour, this));
   };
   // обработка запросов на доставку сообщений
   while (!m_taskqueue.empty())
   {
    // изымаем сообщение
    task_type task = m_taskqueue.front();
    m_taskqueue.pop_front();
    // помещаем его в соответствующий почтовый ящик
    m_arbiterlist[task.address]->deliver(task);
   };
   // цикл работы системы актеров
   busy = false;
   int allnum = m_arbiterlist.size();
   #pragma omp parallel for schedule(guided) reduction(||: busy)
   for (int addr = 0; addr < allnum; ++addr)
   {
    // пополняем признак наличия сообщений в системе
    busy = busy || !m_arbiterlist[addr]->empty();
    // пытаемся получить и обработать сообщение
    task_type task;
    if (m_arbiterlist[addr]->retrieve(task))
     m_arbiterlist[addr]->process(task);
   };
   // выход, если в системе нет сообщений
  } while (busy);

  // уничтожение арбитров
  int allnum = m_arbiterlist.size();
  for (int addr = 0; addr < allnum; ++addr)
   delete m_arbiterlist[addr];
  m_arbiterlist.clear();
 }
};

// не объявленные ранее функции класса actor_type
address_type actor_type::raw_create(const behaviour_type &behaviour) const
{
 assert(m_sched != NULL);
 return m_sched->new_actor(behaviour);
}
void actor_type::raw_send(const task_type &task) const
{
```

```
 assert(m_sched != NULL);
 m_sched->new_task(task);
}
void actor_type::raw_become(const behaviour_type &behaviour) const
{
 assert(m_arbiter != NULL);
 m_arbiter->new_behaviour(behaviour);
}
address_type actor_type::raw_self(void) const
{
 assert(m_arbiter != NULL);
 return m_arbiter->address();
}
```

# Приложение Е.

# Классы для симуляции квантовых вычислений

Ниже приведен код симулятора квантового компьютера, рассмотренного в главе 6. Работа симулятора реализуется классом `quantum_machine_type`. Поскольку квантовый компьютер содержит некую схему вентилей, этот класс унаследован от класса произвольной составной схемы `circuit_type`. Составные схемы формируются на базе других составных схем, а также управляемых схем и однокубитных вентилей. Для образования произвольных однокубитных вентилей и управляемых схем предоставляются соответственно классы `gate_unary_type` и `circuit_controlled_type`.

Манипуляции с векторами состояний и матрицами преобразований производятся с использованием шаблонов классов, приведенных в приложении А.

```
// генерация случайного действительного числа [0.0, 1.0)
template <class real_type>
real_type random(void)
{
 return rand() / (real_type(RAND_MAX) + 1);
}


// == класс-обертка для помещения различных схем в один контейнер ==
// обеспечивает корректное копирование схем в рамках контейнера
template <typename base_type>
class holder_type
{
private:
 enum op_type { BORN, DIE };
 // шаблон создания и уничтожения собственной копии схемы
 template <typename some_type>
 static
 void borndie(const base_type **pp, op_type op)
 {
  if (op == BORN)
   *pp = new some_type(*static_cast<const some_type *>(*pp));
  else if (op == DIE)
   delete static_cast<const some_type *>(*pp);
 }

 // указатель на статическую функцию создания и уничтожения
```

```
 void (*cdtor)(const base_type **, op_type);
 // указатель на собственную копию схемы
 const base_type *ptr;

public:
 // шаблонный конструктор для произвольной схемы
 template <typename some_type>
 explicit
 holder_type(const some_type &circuit)
 {
  // инстанцируем шаблон для заданного типа
  cdtor = &holder_type::template borndie<some_type>;
  // создаем собственную копию объекта
  ptr = &circuit;
  cdtor(&ptr, BORN);
 }
 // деструктор
 ~holder_type(void)
 {
  // уничтожаем собственную копию объекта
  cdtor(&ptr, DIE);
 }
 // конструктор копирования
 holder_type(const holder_type &src):
  cdtor(src.cdtor), ptr(src.ptr)
 {
  cdtor(&ptr, BORN);
 }
 // оператор присваивания
 holder_type & operator =(const holder_type &src)
 {
  if (&src != this)
  {
   cdtor(&ptr, DIE);
   cdtor = src.cdtor;
   ptr = src.ptr;
   cdtor(&ptr, BORN);
  };
  return *this;
 }
 // ссылка на хранимый объект
 const base_type & ref(void) const
 {
  return *ptr;
 }
};


// === абстрактная квантовая схема ===
class circuit_abstract_type
{
public:
 // действительное число
 typedef double real_type;
 // комплексное число
```

```
 typedef std::complex<real_type> element_type;
 // вектор квантового состояния
 typedef vector_type<element_type> qustate_type;
 // набор номеров кубитов
 typedef std::vector<int> pinlist_type;

 // полный размер вектора состояния
 inline
 int qustate_size(void) const { return 1 << qubit_number(); }
 // количество обрабатываемых кубитов
 virtual
 int qubit_number(void) const = 0;
 // выполнение схемы над конкретным вектором состояния
 virtual
 void execute(qustate_type &qustate) const = 0;
};


// === однокубитный вентиль ===
class gate_unary_type: public circuit_abstract_type
{
private:
 // матрица преобразования 2x2
 matrix_type<element_type> m_program;

protected:
 // функция задает преобразование одного из двух базисных состояний
 // |state> --> a |0> + b |1>
 // в наследующих классах должна быть вызвана дважды - с 0 и 1
 void mapsto(
  int state,
  const element_type &a, const element_type &b)
 {
  assert(state >= 0 && state < qustate_size());
  m_program(0, state) = a;
  m_program(1, state) = b;
 }

public:
 gate_unary_type(void):
  m_program(qustate_size(), qustate_size())
 {}
 int qubit_number(void) const { return 1; }
 void execute(qustate_type &qustate) const
 {
  assert(qustate.vsize() == qustate_size());
  qustate = m_program * qustate;
 }
};


// === управляемая схема ===
// управляющий кубит - последний (с максимальным номером)
class circuit_controlled_type: public circuit_abstract_type
{
```

```
private:
 // схема, подлежащая управлению
 holder_type<circuit_abstract_type> m_circuit;

public:
 // конструктор произвольной управляемой схемы
 template <typename some_type>
 explicit
 circuit_controlled_type(const some_type &circuit):
  m_circuit(circuit)
 {}
 // конструктор копирования
 circuit_controlled_type(const circuit_controlled_type &circuit):
  m_circuit(circuit.m_circuit)
 {}

 int qubit_number(void) const
 {
  // количество кубитов больше на один (управляющий)
  return 1 + m_circuit.ref().qubit_number();
 }
 void execute(qustate_type &qustate) const
 {
  assert(qustate.vsize() == qustate_size());
  // вычленяем вторую половину вектора состояния
  qustate_type substate(qustate.vsize() / 2);
  std::copy(
   &qustate(substate.vsize()),
   &qustate(substate.vsize()) + substate.vsize(),
   &substate(0));
  // преобразуем ее подлежащей управлению схемой
  m_circuit.ref().execute(substate);
  // заливаем полученный результат обратно
  std::copy(
   &substate(0),
   &substate(0) + substate.vsize(),
   &qustate(substate.vsize()));
 }
};


// === произвольная составная квантовая схема ===
class circuit_type: public circuit_abstract_type
{
private:
 // набор битовых масок базисных состояний
 typedef std::vector<int> indexmap_type;
 // список всех внутренних схем вместе с наборами битовых масок
 typedef std::list<
  std::pair<
   holder_type<circuit_abstract_type>,
   std::pair<indexmap_type, indexmap_type>
  > > circuits_type;

 // количество кубитов, к которым подключается схема
```

```
 int m_qubitnum;
 // содержимое схемы (список внутренних схем и вентилей)
 circuits_type m_circuits;


public:
 // конструктор
 circuit_type(int qubitnum):
  m_qubitnum(qubitnum)
 {
  assert(qubitnum > 0 && size_t(qubitnum) < sizeof(int) * 8);
 }

 // === функции добавления и удаления внутренних подсхем ===
 // добавление унарной схемы
 template <typename some_type>
 void add_unary(
  const some_type &circuit,
  int qubit)
 {
  pinlist_type pin;
  pin.push_back(qubit);
  add_circuit(circuit, pin);
 }
 // добавление бинарной схемы
 template <typename some_type>
 void add_binary(
  const some_type &circuit,
  int qubit1, int qubit2)
 {
  pinlist_type pin;
  pin.push_back(qubit1);
  pin.push_back(qubit2);
  add_circuit(circuit, pin);
 }
 // добавление тернарной схемы
 template <typename some_type>
 void add_ternary(
  const some_type &circuit,
  int qubit1, int qubit2, int qubit3)
 {
  pinlist_type pin;
  pin.push_back(qubit1);
  pin.push_back(qubit2);
  pin.push_back(qubit3);
  add_circuit(circuit, pin);
 }
 // добавление схемы на произвольное количество кубитов
 template <typename some_type>
 void add_circuit(
  const some_type &circuit,
  const pinlist_type &pin)
 {
  // проверки входного набора номеров кубитов
  assert(pin.size() == size_t(circuit.qubit_number()));
```

```
 assert(std::set<int>(pin.begin(), pin.end()).size() == pin.size());
 assert(*std::min_element(pin.begin(), pin.end()) >= 0);
 assert(*std::max_element(pin.begin(), pin.end()) < qubit_number());

 // строим список номеров кубитов, которых нет в pin
 pinlist_type rest;
 // соберем все номера из pin в битовую маску
 int pinmask = 0;
 pinlist_type::const_iterator it;
 for (it = pin.begin(); it != pin.end(); ++it)
  pinmask |= (1 << *it);
 // соберем все номера кубитов, которых нет в маске
 for (int i = 0; i < qubit_number(); ++i)
  if ((pinmask & (1 << i)) == 0)
   rest.push_back(i);

 indexmap_type::size_type i;
 pinlist_type::size_type j;
 // строим на основе pin набор битовых масок, отображающий
 // соответствующее базисное состояние кубитов подсхемы
 // в битовую маску базисного состояния всей схемы
 indexmap_type inner(1 << pin.size(), 0);
 for (i = 0; i < inner.size(); ++i)
  for (j = 0; j < pin.size(); ++j)
   inner[i] |= (i & (1 << j)) ? (1 << pin[j]) : 0;
 // аналогично для rest
 indexmap_type outer(1 << rest.size(), 0);
 for (i = 0; i < outer.size(); ++i)
  for (j = 0; j < rest.size(); ++j)
   outer[i] |= (i & (1 << j)) ? (1 << rest[j]) : 0;

 // сохраняем копию схемы и соответствующие наборы
 m_circuits.push_back(
  circuits_type::value_type(
   holder_type<circuit_abstract_type>(circuit),
   make_pair(inner, outer)));
}
// очистка всей составной схемы
void remove_all(void)
{
 m_circuits.clear();
}

// === реализация функций абстрактного класса ===
int qubit_number(void) const
{
 return m_qubitnum;
}
void execute(qustate_type &qustate) const
{
 assert(qustate.vsize() == qustate_size());

 // последовательно выполняем все подсхемы
 circuits_type::const_iterator it;
 for (it = m_circuits.begin(); it != m_circuits.end(); ++it)
```

```
  {
   const circuit_abstract_type &c = it->first.ref();
   const indexmap_type &inner = it->second.first;
   const indexmap_type &outer = it->second.second;

   // пробегаем по всем подсостояниям
   int subnum = qustate_size() / c.qustate_size();
   #pragma omp parallel for
   for (int i = 0; i < subnum; ++i)
   {
    // вычленяем очередное подсостояние
    qustate_type substate(c.qustate_size());
    #pragma omp parallel for
    for (int j = 0; j < substate.vsize(); ++j)
     substate(j) = qustate(outer[i] | inner[j]);

    // применяем к нему схему
    c.execute(substate);

    // раскладываем подсостояние обратно
    #pragma omp parallel for
    for (int j = 0; j < substate.vsize(); ++j)
     qustate(outer[i] | inner[j]) = substate(j);
   };
  };
 }

 // === вспомогательная функция быстрой генерации диапазона ===
 static
 pinlist_type range(int begin, int end)
 {
  int d = (begin < end) ? 1 : -1;
  pinlist_type pin;
  for (int i = begin; i != end; i += d)
   pin.push_back(i);
  return pin;
 }
};


// === квантовый компьютер ===
class quantum_machine_type: public circuit_type
{
private:
 // вектор текущего состояния
 qustate_type m_qustate;

public:
 quantum_machine_type(int qubitnum):
  circuit_type(qubitnum), m_qustate(qustate_size())
 {
  prepare(0);
 }

 // === естественные функции квантового компьютера ===
```

```
 // подготовка (приведение в конкретное базисное состояние)
 int prepare(int state)
 {
  assert(state >= 0 && state < qustate_size());
  m_qustate = qustate_type(qustate_size());
  m_qustate(state) = 1.0;
  return state;
 }
 // выполнение заложенной квантовой схемы
 void run(void)
 {
  execute(m_qustate);
 }
 // измерение квантового состояния
 int measure(void)
 {
  real_type rnd = random<real_type>();
  real_type prob = 0.0;
  int state = -1;
  for (int i = 0; state < 0 && i < m_qustate.vsize(); ++i)
  {
   prob += std::norm(m_qustate(i));
   state = (rnd < prob) ? i : state;
  };
  return prepare(state);
 }

 // === функции, реализуемые лишь симуляторами ===
 // получение вектора текущего состояния
 qustate_type qustate(void) const
 {
  return m_qustate;
 }
 // построение матрицы выполняемого схемой преобразования
 matrix_type<element_type> build_matrix(void) const
 {
  matrix_type<element_type> program(qustate_size(), qustate_size());

  for (int j = 0; j < program.hsize(); ++j)
  {
   // формируем очередной базисный вектор
   qustate_type qustate(qustate_size());
   qustate(j) = 1.0;

   // преобразуем его
   execute(qustate);

   // помещаем значения результата в матрицу
   for (int i = 0; i < program.vsize(); ++i)
    program(i, j) = qustate(i);
  };
  return program;
 }
};
```

# Литература


1. Антонов А.С. Введение в параллельные вычисления. Методическое пособие. — М.: Изд-во МГУ, 2002. — 70 с.

2. Антонов А.С. Параллельное программирование с использованием технологии MPI. Учебное пособие. — М.: Изд-во МГУ, 2004. — 71 с.

3. Антонов А.С. Параллельное программирование с использованием технологии OpenMP. Учебное пособие. — М.: Изд-во МГУ, 2009. — 77 с.

4. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. — Ростов-на-Дону: Изд-во ООО «ЦВВР», 2003. — 208 с.

5. Вавилов К.В. Программирование за... 1 (одну) минуту // Компьютер Price. — 2002. №31. — С. 288-293.

6. Валиев К.А., Кокин А.А. Квантовые компьютеры: надежды и реальность. — Ижевск: РХД, 2001. — 352 с.

7. Вентцель Е.С. Исследование операций. — М.: Сов. радио, 1972. — 552 с.

8. Виноградов И.М. Основы теории чисел. — М.-Л.: Гостехиздат, 1952. — 180 с.

9. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. Серия «Научное издание». — СПб.: БХВ-Петербург, 2002. — 608 с.

10. Глебов А.Н. Параллельное программирование в функциональном стиле. — 2003 [Электронный ресурс] URL: `http://www.softcraft.ru/parallel/ppfs.shtml`.

11. Глушков В.М. Синтез цифровых автоматов. — М.: Физматгиз, 1962. — 476 с.

12. Дехтяренко И.А. Декларативное программирование. — 2003 [Электронный ресурс] URL: `http://www.softcraft.ru/paradigm/dp/`.

13. Дехтярь М.И. Введение в схемы, автоматы и алгоритмы. — 2007 [Электронный ресурс] URL: `http://www.intuit.ru/department/ds/introsaa/`.

14. Захаров Н.Г., Рогов В.Н. Синтез цифровых автоматов: Учебное пособие. — Ульяновск: УлГТУ, 2003. — 136 с.

15. Зюбин В.Е. Многоядерные процессоры и программирование // Открытые системы. — 2005. №7-8. — С.12-19.

16. Илюшкин Б.И. Операционные системы: Процессы и потоки. Учебное пособие. — СПб.: СЗТУ, 2005. — 103 с.

17. Котов В.Е. Введение в теорию схем программ. — Новосибирск: Наука, 1978. — 258 с.

18. Котов В.Е. Сети Петри. — М.: Наука, 1984. — 160 с.

19. Кремер Н.Ш., Путко Б.А., Тришин И.М., Фридман М.Н. Исследование операций в экономике: Учебное пособие. Под. ред. Кремера Н.Ш. — М.: ЮНИТИ, 1997. — 408с.

20. Кузнецов Б.П. Психология автоматного программирования // BYTE-Россия. — 2000. №11. — С. 22-29.

21. Кузнецов С.Д. Блеск и нищета легковесных процессов // Computerworld Россия. — 1996. №31.

22. Левин М.П. Параллельное программирование с использованием OpenMP. — М.: ИНТУИТ, 2008. — 120 с.

23. Лескин А.А., Мальцев П.А., Спиридонов А.М. Сети Петри в моделировании и управлении. — Л.: Наука, 1989. — 133 с.

24. Ли Эдвард А. Проблемы с потоками // IEEE Computer, 2006. — Перевод с англ.: Петров А.В., 2007 [Электронный ресурс] URL: `http://www.softcraft.ru/parallel/pwt/index.shtml`.

25. Любченко В.С. К проблеме создания модели параллельных вычислений // Труды Третьей международной конференции «Параллельные вычисления и задачи управления» (PACO'2006). Москва, 2-4 октября 2006 г. Институт проблем управления им. В.А. Трапезникова РАН. — М.: Институт проблем управления им. В.А. Трапезникова РАН, 2006. — С. 1359-1374.

26. Любченко В.С. К решению проблемы обедающих философов Дейкстры // Высокопроизводительные параллельные вычисления на кластерных системах: Материалы четвертого международного научно-практического семинара. — Самара: СГАУ, 2005. — С. 186-193.

27. Любченко В.С. Конечно-автоматная технология программирования // Труды международной научно-методической конференции «Телематика'2001». СПб.: СПбГИТМО(ТУ), 2001. — С. 127-128.

28. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. Серия «Мастер программ». — СПб.: БХВ-Петербург, 2002. — 400 с.

29. Питерсон Дж. Теория сетей Петри и моделирование систем: Пер. с англ. — М.: Мир, 1984. — 264 с.

30. Рабинер Л., Гоулд Б. Теория и применение цифровой обработки сигналов. — М.: Мир, 1978. — 848 с.

31. Стивенс У.Р. UNIX: разработка сетевых приложений. — СПб.: Питер, 2003. — 1088 с.

32. Трахтенброт Б.А., Барздинь Я. М. Конечные автоматы (поведение и синтез). — М.: Наука, 1970. — 400 с.

33. Хинчин А.Я. Цепные дроби. — М.: ГИФМЛ, 1960. — 112 с.

34. Хоар Ч. Взаимодействующие последовательные процессы: Пер. с англ. — М.: Мир, 1989. — 264 с.

35. Холево А.С. Введение в квантовую теорию информации. — М.: МЦНМО, 2002. — 128 с.

36. Цилюрик О., Горошко Е. QNX/UNIX: анатомия параллелизма. — СПб.: Символ-Плюс, 2006. — 288 с.

37. Чан Т. Системное программирование на C++ для UNIX. — К.: BHV, 1999. — 592 с.

38. Шалобанов С.В. Моделирование систем управления: Методические указания. — Хабаровск: Изд-во Хабар. гос. техн. ун-та, 2003. — 49 с.

39. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. — СПб.: Наука, 1998. — 628 с.

40. Шалыто А.А. Автоматное проектирование программ. Алгоритмизация и программирование задач логического управления // Известия Академии наук. Теория и системы управления. — №6. Ноябрь-Декабрь 2000. — С. 63-81.

41. Шалыто А.А. Использование граф-схем и графов переходов при программной реализации алгоритмов логического управления // Автоматика и телемеханика. — 1996. №6. С.148-158; №7. С.144-169.

42. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному // Мир ПК. — 2002, №2. — С.144-149.

43. Шалыто А.А., Туккель Н.И. Преобразование итеративных алгоритмов в автоматные // Программирование. — 2002. №5. — С. 12-26.

44. Шалыто А.А., Туккель Н.И. Программирование с явным выделением состояний // Мир ПК. — 2001, №8, №9. — С.116-121; С.132-138.

45. Шалыто А.А., Туккель Н.И., Шамгунов Н.Н. Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. — 2002. №5. — С. 72-99.

46. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие. — Мн.: БГУ, 2002. — 323 с.

47. Элементы параллельного программирования / Вальковский В.А., Котов В.Е., Марчук А.Г., Миренков Н.Н. — М.: Радио и связь, 1983. — 240 с.

48. Agha G.A. Actors: A Model of Concurrent Computation in Distributed Systems. — Cambridge, Massachusetts: MIT Press, 1986. — 190p.

49. Agha G., Thati P. An Algebraic Theory of Actors and its Application to a Simple Object-Based Language // Festschrift in Honor of Ole-Johan Dahl. — Lecture Notes in Computer Science, Vol.2635, 2004. — 33p.

50. Armstrong J., Virding R., Wikstrom C., Williams M. Concurrent Programming in Erlang. — Hertfordshire: Prentice Hall, 1996. — 358p.

51. Atkinson R., Hewitt C. Specification and Proof Techniques for Serializers. — Technical memo. — MIT Artificial Intelligence Laboratory, 1977. — 39p.

52. Barenco A., et al. Elementary gates for quantum computation // Physical Review A. — Vol.52, №5, Nov 1995. — P.3457-3467.

53. Barenco A., Ekert A., Suominen K.-A., Torma P. Approximate Quantum Fourier Transform and Decoherence // Physical Review A. — Vol.54, №1, Jul 1996. — P.139-146.

54. Bennett Charles H. Notes on Landauer's principle, Reversible Computation and Maxwell's Demon // Studies in History and Philosophy of Modern Physics. — Vol.34, №3, Sep 2003. — P.501-510.

55. Brun Todd A. Lecture Notes in Quantum Information Processing. [Электронный ресурс] URL: `http://www-bcf.usc.edu/~tbrun/Course/index.html`.

56. Butenhof David R. Programming with POSIX Threads. — Massachusetts: Addison-Wesley, 1997. — 398 p.

57. Chandy K.M., Misra J. The Drinking Philosophers Problem // ACM Transactions on Programming Languages and Systems (TOPLAS). — Vol.6, №4, Oct 1984. — P.632-646.

58. Clinger W.D. Foundations of Actor Semantics. — Doctoral Dissertation. — MIT Artificial Intelligence Laboratory, 1981. — 177p.

59. Coppersmith D. An approximate Fourier transform useful in quantum factoring. IBM Research Report RC 19642. — Dec 1994. — 9 p.

60. Deutsch D. Quantum theory, the Church-Turing principle and the universal quantum computer // Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences. — Vol.400, №1818, Jul 1985. — P.97-117.

61. Fredkin E., Toffoli T. Conservative Logic // International Journal of Theoretical Physics. — Vol.21, №3, Apr 1982. — P.219-253.

62. Gossett P. Quantum Carry-Save Arithmetic. — 1998 [Электронный ресурс] URL: `http://arxiv.org/abs/quant-ph/9808061`.

63. Grover Lov K. A fast quantum mechanical algorithm for database search // Proceedings, 28th Annual ACM Symposium on the Theory of Computing (STOC). — May 1996. — P.212–219.

64. Hewitt C., Baker H. Laws for Communicating Parallel Processes // 1977 IFIP Congress Proceedings. — Toronto: IFIP, 1977. — P.987-992.

65. Hewitt C., Bishop P., Steiger R. A Universal Modular Actor Formalism for Artificial Intelligence // International Joint Conference on Artifcial Intelligence. — Stanford, California. 1973. — P.235-245.

66. Hewitt C. Viewing Control Structures as Patterns of Message Passing // Journal of Artifcial Intelligence. — Vol.8, №3, 1977. — P.323-364.

67. Hewitt C. What is Commitment? Physical, Organizational, and Social // Workshop of the COIN at AAMAS06. — Hakodate: 2006. — 16p.

68. Lee Edward A. The problem with threads // IEEE Computer. — Vol.39, №5, May 2006. — P. 33-42.

69. Meglicki Z. Introduction to Quantum Computing (M743). — Indiana University: 2002. — 264 p.

70. Mitchell J.C. Concepts in Programming Languages. — Cambridge: Cambridge University Press, 2003. — 529p.

71. MPI: A Message-Passing Interface Standard. (Version 1.1: June, 1995) [Электронный ресурс] URL: `http://www.mcs.anl.gov/mpi/standard.html`.

72. MPI-2: Extensions to the Message-Passing Interface. [Электронный ресурс] URL: `http://www.mcs.anl.gov/mpi/standard.html`.

73. OpenMP C and C++ Application Program Interface (Version 1.0: October, 1998) [Электронный ресурс] URL: `http://www.openmp.org/mp-documents/cspec10.pdf`.

74. OpenMP C and C++ Application Program Interface (Version 2.0: March, 2002) [Электронный ресурс] URL: `http://www.openmp.org/mp-documents/cspec20.pdf`.

75. OpenMP Application Program Interface (Version 3.0: May, 2008) [Электронный ресурс] URL: `http://www.openmp.org/mp-documents/spec30.pdf`.

76. Rieffel E., Polak W. An Introduction to Quantum Computing for Non-Physicists // ACM Computing Surveys. — Vol.32, №3, Sep 2000. — P.300-335.

77. Shor Peter W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer // SIAM Journal on Computing. — Vol.26, №5, 1997. — P.1484-1509.

78. Snir M., Otto S., Huss-Lederman S., etc. MPI — The Complete Reference: The MPI Core. — 2-nd edn. — Cambridge: MIT Press, 1998. — 426 p.

79. Snir M., Otto S., Huss-Lederman S., etc. MPI: The complete Reference. — MIT Press, Cambridge, Massachusetts, 1997 [Электронный ресурс] URL: `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`.

80. Wagner F., Schmuki R., Wagner T., Wolstenholme P. Modeling Software with Finite State Machines: A Practical Approach. — Auerbach Publications, 2006. — 392 p.

81. Wagner T. Towsley D. Getting Started With POSIX Threads. — Massachusetts: University of Massachusetts at Amherst, 1995. — 12 p.

82. Williams C.P., Clearwater S.H. Explorations in quantum computing. — Berlin, Germany: TELOS/Springer-Verlag, 1998. — 307 p.