ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ
РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ
(ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)»

И.Е. ФЕДОТОВ

НЕКОТОРЫЕ ПРИЕМЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

УЧЕБНОЕ ПОСОБИЕ
по специальности 230401 –
«Прикладная Математика»

МОСКВА 2008

ББК 22.18я7+32.81я7
Ф34
УДК 004.42(075)+519.712(075)

Рецензенты: д.ф.-м.н. А.С. Крюковский, д.ф.-м.н. А.В. Сетуха.

Ф34 Федотов И.Е. Некоторые приемы параллельного программирования: Учебное пособие / Государственное образовательное учреждение высшего профессионального образования «Московский государственный институт радиотехники, электроники и автоматики (технический университет)» – М., 2008. – 188 с.

Учебное пособие посвящено рассмотрению некоторых из основных существующих подходов к построению параллельных программ. Рассмотрение сопровождается приведением примеров на популярном языке программирования с использованием популярных современных средств и программных интерфейсов. Рассмотрены вопросы абстрактной программной реализации описанных подходов без привязки к конкретным задачам, а также приведены примеры решения с использованием такой реализации некоторых конкретных задач.

Пособие предназначено для студентов старших курсов и аспирантов, специализирующихся в области прикладной математики и численного моделирования.

Ил. 34, Библиогр.: 41 назв.

Печатается по решению редакционно-издательского совета университета.

ISBN 978-5-7339-0724-6

Введение

О целях издания

Настоящее пособие посвящено рассмотрению некоторых существующих подходов в написании параллельных программ. Поскольку сам по себе факт наличия параллелизма среды выполнения и вытекающая отсюда необходимость распараллеливания программ в общем случае не привязаны к конкретной реализации параллельной среды, здесь не приводится классификация многопроцессорных систем и не делается акцента на конкретную категорию вычислительной системы. Для ознакомления с классификацией можно обратиться к другой литературе [1, 3, 6].

Также здесь не рассматриваются специализированные языки параллельного программирования, такие как, например, Occam, или языки с естественной поддержкой параллельного программирования, наподобие Erlang или Oz. Здесь, скорее, 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 будет наиболее удобен язык C++, хотя вообще выбор языка непринципиален.

Наконец, поскольку настоящие пособие также не предназначено для первоначального ознакомления с основами программирования, оно не содержит обзора используемого в примерах языка. Напротив, для понимания изложенного материала необходимо хорошее владение используемым 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.

Приводимые программы, как правило, не являются как таковыми полноценными программами для компиляции, а являются лишь фрагментами кода, иллюстрирующими 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).

Реализация как такового механизма распараллеливания для большинства программ приводится с использованием нескольких различных программных интерфейсов, при этом делается попытка максимально сохранить предоставляемый вызывающему коду программный интерфейс и, соответственно, избежать изменений в нем при переходе от одного способа реализации 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 архитектуры параллельной программы, наиболее полно соответствующий стоящей задаче.

Поскольку полностью уйти от вопросов реализации нельзя, в первой главе приводится краткий обзор двух популярных программных интерфейсов распараллеливания с примерами программ. Однако за более подробным описанием этих и других программных интерфейсов следует обращаться к спецификациям [36, 38] и специально посвященным им изданиям [2, 3, 31].

В остальных главах рассматриваются несколько моделей параллельного программирования, существующих сегодня и зачастую выдвигаемых их приверженцами в качестве универсального средства построения параллельных программ. Возможные подходы не ограничиваются рассмотренными вариантами. К примеру, в силу довольно специфичных областей применения здесь не рассмотрены нейронные сети и клеточные автоматы.

Во всех случаях делается попытка отделить универсальные механизмы построения соответствующих выбранной модели конструкций и параллельного выполнения ветвей от специфики конкретной задачи. С этой целью строится универсальный набор классов, который может быть использован для решения широкого круга задач, и в том числе используется в приводимых примерах. Реализация конкретных задач сводится к написанию кода, использующего эти классы. Осуществление же параллельного выполнения отдельных ветвей инкапсулируется внутри них, что позволяет упростить написание и тем самым повысить надежность вызывающего кода.

О проблеме параллельного программирования

Большинство сегодняшних параллельных вычислений реализуется в виде параллельно-последовательных программ. При этом в существующем последовательном алгоритме выделяются независимые последовательные ветви, которые могут быть выполнены параллельно, и с этим учетом пишется параллельно-последовательная программа. Зачастую изначально берется существующая последовательная программа и путем добавления неких конструкций распараллеливания преобразуется в параллельно-последовательную. Чем сложнее исходная последовательная программа, чем запутаннее зависимости между отдельными ее ветвями, тем сложнее оказывается выполнить ее распараллеливание.

Основная проблема сегодняшнего параллельного программирования заключается в 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. Сейчас, когда полупроводниковые компьютеры подошли к пределу своей производительности и начинают расти «вширь», а не «в высоту», становится все более актуальным программирование параллельных вычислительных структур. Большинство авторов, работающих в области параллельного программирования, сходятся в следующем: развитой дисциплины параллельного программирования на сегодняшний момент нет.

Существует немало изданий, посвященных интерфейсам и технологиям параллельного программирования, наподобие рассматриваемых ниже MPI и OpenMP. Однако это лишь инструмент, а программисту, как правило, одного прекрасного знания инструмента недостаточно для того, чтобы легко и эффективно реализовать конкретную задачу. Иначе говоря, помимо знаний о том, что нужно использовать и как это нужно использовать, необходимы знания о том, для чего это нужно использовать, какие задачи при этом выполнять и, самое главное, как выполнять эти самые задачи.

Многие авторы публикаций по параллельным вычислениям сходятся в том, что для описания параллельных программ необходимо уйти от императивных языков программирования. Императивные языки (от лат. imperativus – повелительный) описывают, какие действия и в каком порядке надо выполнить, чтобы получить результат. В данном же случае требуется использование декларативных (от лат. declaratio — заявление, объявление) языков, т.е. таких, которые описывают, чем является результат, который должен быть получен, оставив выполнение действий на долю компилятора или интерпретатора. Примерами декларативных языков являются язык логического программирования Prolog и язык функционального программирования Lisp.

Однако многие существующие языки декларативного программирования, несмотря на свою мощь, не обеспечивают полноценной базовой платформы для описания параллельных алгоритмов. Причина в том, что спецификации этих языков создавались тогда, когда вопрос параллельного программирования не стоял так остро, как сегодня, и, видимо, поэтому они зачастую также содержат ограничения, сводящие вычисления к последовательным.

Таким образом, для полноценного удобного описания параллельных программ, так или иначе, потребуется новый язык. Сегодня известно немало попыток создания подобных языков. Автор воздержится от их перечисления, дабы избежать оглашения личных предпочтений. Результаты этих попыток пока не обрели широкой популярности, и можно лишь надеяться, что они обретут ее в будущем. В противном случае можно также надеяться, что появятся другие полноценные параллельные языки высокого уровня. Пока же мы будем отталкиваться от факта отсутствия последних, в связи с чем будем рассматривать популярные модели параллельных вычислений с примерами на основе популярного императивного языка.

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

Разделяют понятия логического и физического параллелизма. В случае логического параллелизма задача может быть разделена на независимые подзадачи, которые вследствие своей независимости могут решаться параллельно. При этом физически выполнение может производиться как угодно, в том числе последовательно. В случае физического параллелизма подразумевают, что задача физически должна выполняться параллельно в некоторой существующей параллельной вычислительной системе. При этом задача может не обладать ярко выраженным логическим параллелизмом, вследствие чего распараллеливание может являться в некотором роде неестественным и сопровождаться сложностью программной реализации. Очевидно, наилучшие возможности по физическому распараллеливанию предоставляют алгоритмы, обладающие логическим параллелизмом, поэтому здесь рассматриваются возможные пути представления задач с использованием логического параллелизма.

Разумеется, ни один из представленных ниже подходов к организации параллельного выполнения задач не является панацеей и не дает простого и наглядного 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 основ всех рассмотренных моделей и парадигм в комплексе может существенно упростить разработку параллельных программ с той точки зрения, что даст возможность смотреть шире и сгладит отсутствие у человека «параллельного» мышления.

По сути, в рамках всех описанных моделей рассматриваемые задачи представляются декларативно (сетевым графиком работ, диаграммой состояний автомата, схемой автоматной сети, схемой сети Петри), выполнение же возлагается на реализацию, которая может быть как последовательной, так и параллельной с использованием различных подходов к физическому параллелизму. Иначе говоря, параллельное выполнение декларативно описанной программы зависит от внутренней реализации конкретного механизма обработки декларативных данных, а не возлагается на плечи программиста.

Об используемой терминологии

Следует оговорить некоторые используемые в дальнейшем термины, поскольку в существующей литературе они зачастую расходятся. К примеру, в одних источниках термин «процессор» используются для указания отдельного последовательного вычислительного элемента, 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 многоядерного 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 следующее расхождение между терминами, хотя оно не столь принципиально. Так сложилось, что отдельные параллельные ветви выполняющегося процесса (threads, light-weight processes) многие называют потоками (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 русскоязычных терминах 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 в силу исторически сложившихся обстоятельств этот термин оказывается более привычным для большинства программистов.

Поскольку изложение зачастую не будет привязано к реализации физического параллелизма, помимо прочего нам потребуется универсальный термин, которым мы будем обозначать некий ресурс, в рамках которого выполняется последовательная ветвь программы. Будем называть такую сущность параллельным ресурсом. В зависимости от контекста, этот термин может означать поток многопоточного процесса или последовательный процесс, выполняющийся на узле распределенной системы.

Наконец, следует оговорить понятие «вызывающий код», которое часто встречается в изложении. Выше говорилось, что код большинства приводимых программ разделен на общие универсальные классы и некий код, реализующий конкретную задачу и использующий эти классы. Под вызывающим кодом в изложении подразумевается именно код, реализующий конкретную задачу, поскольку он явно или неявно вызывает код универсальных классов.

Глава 1. Интерфейсы и технологии параллельного программирования

Здесь 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 параллелизма для выполнения в параллельной среде.

Общий подход при использовании всех подобных технологий заключается в разработке в два этапа. Первый этап – написание и полноценная отладка последовательной версии программы. Второй этап – распараллеливание с использованием выбранного средства и последующая отладка. Написание сразу параллельной версии, разумеется, физически возможно, но сопряжено с существенным усложнением отладки, связанным со сложностью поиска источника ошибки, поскольку он может крыться как в ошибочном алгоритме, так и в некорректном распараллеливании.

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

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 изначально последовательной программы, оставляя за кадром тонкости создания потоков и управления ими.

Здесь будет приведено поверхностное описание предоставляемого интерфейса с целью создания общего представления о принципах работы с ним. Для более подробного ознакомления с форматом описания директив и вызова функций следует обращаться к спецификации [38].

Весь предоставляемый OpenMP интерфейс 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 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 использования директив OpenMP. Использование же runtime-функций OpenMP не столь удобно, о чем подробнее будет сказано при описании этих функций.

1.1.1 Первая программа – ряд Лейбница

Одним из довольно популярных простейших примеров вычислительных задач, используемых для демонстрации распараллеливания программ, является программа вычисления числа π. Обычно при этом приближенно вычисляют 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 выглядит следующим образом:

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

Одним из свойств этого ряда является очень медленная сходимость, которая и объясняет удобство его рассмотрения для демонстрации распараллеливания вычислений. Здесь мы, однако, не оговариваем вопросы оптимального порядка вычисления этого ряда с целью повышения точности, поскольку этот вопрос лежит вне обсуждаемой темы. Последовательная программа вычисления суммы первых n членов ряда выглядит достаточно просто:

```
#define NUM_ITERATIONS  1000000

int main(int argc, char *argv[])
{
 double sum = 0.0;
 for (int i = 0; i < NUM_ITERATIONS; i++)
  sum += ((i & 1) ? - 1.0 : 1.0) / ((i << 1) | 1);
 std::cout << std::setprecision(20) << sum * 4.0 << std::endl;
 return 0;
}
```

Она состоит из одного цикла, в котором текущее значение суммы ряда последовательно пополняется очередными его членами. В конце полученная сумма увеличивается в четыре раза, чтобы получить число π. При вычислении каждого члена ряда не используются рекуррентные соотношения, и все итерации являются независимыми друг от друга, что позволяет нам свободно выполнять распараллеливание итераций.

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). Каждый поток должен вычислить локальную сумму своей части ряда, после чего эти суммы из всех потоков должны быть просуммированы.

Thread 0

$$s_0 = \sum_{i=0}^{n/N-1} \frac{(-1)^i}{2i+1}$$

. . .

Thread N-1

$$s_{N-1} = \sum_{i=0}^{n/N-1} \frac{(-1)^{(N-1)n/N+i}}{2((N-1)n/N+i)+1}$$

$$\sum_{i=0}^{N-1} s_i \approx \frac{\pi}{4}$$

Рис. 1

Для низкоуровневой организации многопоточного распараллеливания существуют различные программные интерфейсы. К примеру, можно воспользоваться интерфейсом Win32 API, предоставляемым операционными системами семейства Microsoft Windows. В этом случае полученный после распараллеливания код может выглядеть следующим образом:

```
#define NUM_ITERATIONS  1000000
#define NUM_THREADS  4

struct thr_param
{
 int begin;
 int end;
 double result;
};

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[])
{
 // объявление структур с параметрами для каждого потока
 thr_param param[NUM_THREADS];
 for (int i = 0; i < NUM_THREADS; i++)
 {
  param[i].begin = i * (NUM_ITERATIONS / NUM_THREADS);
  param[i].end = (i + 1) * (NUM_ITERATIONS / 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 их работы
 ::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;
 std::cout << std::setprecision(20) << sum * 4.0 << std::endl;
 return 0;
}
```

Программа осуществляет создание NUM_THREADS потоков, после чего ожидает их завершения, освобождает ресурсы, суммирует промежуточные суммы и выводит полученный результат. 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 текущем потоке.

Как таковое суммирование ряда 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.

Во многих UNIX-системах для организации многопоточной работы приложениям предоставляется программный интерфейс POSIX Threads (pthreads). Распараллеленный с использованием такого интерфейса код довольно похож на вариант распараллеливания с помощью функций Win32 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 их работы 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, будем осуществлять создание меньшего на единицу количества дополнительных потоков, при этом главный поток перед выполнением ожидания должен выполнить свою часть работы. Построенная таким образом программа с использованием интерфейса POSIX Threads может выглядеть следующим образом:

```
#define NUM_ITERATIONS  1000000
#define NUM_THREADS  4

struct thr_param
{
 int begin;
 int end;
 double result;
};

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[])
{
 // объявление структур с параметрами для каждого потока
 thr_param param[NUM_THREADS];
 for (int i = 0; i < NUM_THREADS; i++)
 {
  param[i].begin = i * (NUM_ITERATIONS / NUM_THREADS);
  param[i].end = (i + 1) * (NUM_ITERATIONS / 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 их работы 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;
 std::cout << std::setprecision(20) << sum * 4.0 << std::endl;
 return 0;
}
```

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

Теперь перейдем к главному моменту текущего раздела. Последней приведенной программе по функциональности полностью эквивалентен следующий код:

```
#define NUM_ITERATIONS  1000000
#define 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_ITERATIONS; i++)
  sum += ((i & 1) ? - 1.0 : 1.0) / ((i << 1) | 1);
 std::cout << std::setprecision(20) << sum * 4.0 << std::endl;
 return 0;
}
```

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

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. 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, мы видим, что весь функционал, который мы только что выполняли вручную с помощью низкоуровневых интерфейсов, может быть выполнен автоматически и практически без нашего участия. Более того, этот эффект может быть достигнут лишь при использовании специального флага компилятора. В противном случае программа будет скомпилирована 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 чуть более детально.

1.1.2 Основные конструкции параллельного выполнения

Использование директив OpenMP заключается во вставке строк с директивами препроцессора перед существующими участками кода. Строка директивы OpenMP 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:

```
#pragma omp parallel
{
 //...
}
```

Объявленный такой директивой параллельный регион ограничивается оператором, следующим сразу за ним. Это может быть также составной оператор, ограниченный фигурными скобками.

В начале параллельного региона создается некоторое количество потоков, после чего все они выполняют участок кода, заключенный в параллельный регион. В конце региона выполняется барьерная синхронизация всех созданных в начале 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 одним потоком, остальные уничтожаются за ненадобностью (вопрос возможности кэширования потоков здесь не рассматривается, поскольку является вопросом внутренней реализации).

С помощью директивы parallel объявляется участок кода, который должен быть выполнен параллельно несколько раз. К примеру, следующий фрагмент кода на многопроцессорной машине выведет несколько строк приветствия:

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

Строго говоря, такой код может вывести несколько строк отнюдь не последовательно, а вперемешку, поскольку здесь все потоки осуществляют одновременный доступ к общему ресурсу – консоли. Однако здесь и далее в текущем разделе мы будем считать, что вывод строки в консоль является атомарной операцией для всех потоков параллельного региона. Для случаев, когда это не так, может быть выполнено явное разграничение доступа к общему ресурсу с помощью описанных ниже директив.

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

```
#pragma omp parallel num_threads(5)
printf("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 также и вообще фактом осуществления распараллеливания текущего региона. К примеру, если нам требуется выполнять регион параллельно лишь в случае выполнения каких-либо условий, мы можем воспользоваться параметром if:

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

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

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

Внутри параллельного региона может быть выполнено разделение работы между потоками. Как правило, параллельный регион содержит какие-либо участки кода, которые должны быть выполнены параллельно, но каждый из них не должен быть выполнен многократно. Тогда требуется распределить выполнение параллельного региона частями между параллельными потоками. Для такого распределения предусмотрены директивы for, sections и single.

Директива sections позволяет распределить между потоками разные независимые участки кода. К примеру, если некоторый участок кода выполняет 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
 printf("Hello, World 1!\n");
 #pragma omp section
```

```
 printf("Hello, World 2!\n");
 #pragma omp section
 printf("Hello, World 3!\n");
}
```

Каждый независимый фрагмент участка sections обрамляется в отдельный участок section. В приведенном фрагменте 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, в них не должно быть зависимости от порядка выполнения. К примеру, рекуррентные итерации являются зависимыми, поэтому их распараллеливать 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++)
 printf("Hello, World %d!\n", i);
```

При этом аргументы цикла должны быть в так называемой канонической форме. Говоря коротко, аргументы должны содержать инициализацию некой переменной, сравнение ее значения с заданным значением границы диапазона и приращение значения переменной, которое может содержать увеличение или уменьшение на фиксированную величину. Подробнее об этом можно прочитать в спецификации [38].

Итерации цикла, объявленного в контексте директивы for, распределяются между потоками объемлющего региона parallel. Для явного указания способа распределения итераций между потоками (статическое, динамическое, др.) используется параметр 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
 printf("Stage 1\n");
 // ... какая-либо параллельная работа
 #pragma omp single
 printf("Stage 2\n");
 // ... другая параллельная работа
}
```

В конце регионов 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 независимы друг от друга, а третий зависит от них обоих:

```
#pragma omp parallel
{
 #pragma omp for nowait
 for (int i = 0; i < 200; i++)
 {
  // ...
 }
 #pragma omp for
 for (int j = 0; j < 300; j++)
 {
  // ...
 }

 #pragma omp for
 for (int k = 0; k < 100; k++)
 {
  // ...
 }
}
```

Вследствие указания параметра nowait перед первым циклом, после него не выполняется синхронизация потоков, вследствие 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 вследствие того, что во второй директиве for параметр nowait не указан, а значит, послед 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, вследствие чего 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 область внутри параллельного региона, которая должна быть выполнена только главным потоком. 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 соответствующей области.

Другой директивой, также отчасти близкой к single, является директива critical. Она объявляет область внутри параллельного региона, которая в любой момент времени может выполняться лишь одним потоком. 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 < 100; i++)
{
 int item;
 // ... вычисление элемента суммы
 #pragma omp critical
 sum += item;
}
```

В приведенном фрагменте осуществляется изменение общей переменной sum. Поскольку 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 директива critical предназначена для более сложных ситуаций, нежели приведенная, к примеру, для взаимоисключающего доступа к файлу. Приведенный же пример грамотнее реализовать с использованием параметра reduction директивы for.

В параллельном регионе могут быть объявлены области critical, требующие взаимоисключающего выполнения не со всеми, а лишь с некоторыми объявленными в том же регионе областями critical. В таких случаях бывает удобным задавать в качестве параметра директивы critical имя соответствующей критической области. В этом случае поток по достижении начала критической области 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 критической области с тем же именем. Критические области, имя для которых не указано, считаются одноименными.

Директива atomic является более упрощенным вариантом critical. В то время как область конструкции critical может быть какой угодно, область конструкции atomic ограничивается одним оператором присваивания с модификацией, таким как в приведенном ранее примере с директивой critical, или же выражением инкремента-декремента. При этом критической областью является левая часть оператора, а именно изменение какой-либо переменной. Правая часть не попадает в критическую область, поэтому если в правой части стоит, к примеру, вызов длительной вычислительной функции, это не вызовет задержки всех остальных потоков.

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 содержать при этом параметр ordered.

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. Такая интерпретация может быть изменена с помощью параметра default(none) директивы parallel.

Директива threadprivate объявляет разделенный запятыми список глобальных либо статических переменных, которые следует сделать частными. В момент создания потоков в начале параллельного региона для такой переменной в каждом потоке создается своя копия. До момента первого обращения каждая копия инициализируются 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 переменных 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++)
 printf("Local copy 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. Наличие copyin заставляет компилятор в начале параллельного региона присвоить копиям значение из главного потока. Иначе говоря, при отсутствии copyin значения частных переменных определяются статически во время компиляции, при наличии – динамически, во время выполнения.

Для объявления частными либо общими переменных, не являющихся глобальными либо статическими, предусмотрены параметры 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, копируются в исходную переменную с помощью оператора присваивания.

Параметр shared позволяет в явном виде объявить перечисленные в скобках переменные общими.

Наконец, параметр reduction предназначен для обозначения общих переменных, в которые попадает результат некой множественной операции, выполняемой внутри параллельного региона. К примеру, это может быть сумма, произведение или битовое сложение по модулю два большого количества элементов. 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 создается копия в каждом потоке и инициализируется «пустым» значением. Для суммы это ноль, для произведения – единица, для логического «ИЛИ» – «ЛОЖЬ», и т.д. После выполнения параллельного региона значения всех копий объединяются с помощью той же операции, и результат попадает в исходную переменную.

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

```
int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < 100; i++)
{
 int item;
 // ... вычисление элемента суммы
 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, проще 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 сложности 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 использование функций-заглушек, эмулирующих работу runtime-функций в последовательной среде выполнения. Такие функции могут быть реализованы и приложены к программе, которая вызывает runtime-функции OpenMP и 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 [38]. 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, созданных 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;
- 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 динамически управляет количеством создаваемых потоков в последующих параллельных регионах;
- 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, о чем подробнее описано в спецификации [38].

Помимо функций среды выполнения, интерфейс OpenMP предлагает функции блокировки. Эти функции обеспечивают возможность более гибкого построения конструкций, по смыслу близких к critical. Для использования функций блокировки программа должна объявить переменную блокировки. Каждая переменная блокировки 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.

Наконец, среди runtime-функций OpenMP предлагает функции оценки времени выполнения. Среди них две:

- 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 соседними

изменениями значения таймера; бывает нужно для оценки погрешности при замере времени выполнения.

За подробным описанием runtime-функций OpenMP следует обратиться к спецификации [38].

1.1.6 Вычисление определенного интеграла

Для завершения знакомства с возможностями интерфейса OpenMP рассмотрим небольшой пример распараллеливания некой вычислительной процедуры. Покажем на ее примере 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 одной переменной методом средних прямоугольников:

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

double h, x, f;
int i;

do
{
 sumpre = sum;
 h = (b - a) / n;
 sum = 0.0;
 for (i = 0; i < n; i++)
 {
  x = a + h * (i + 0.5);
  f = /* интегрируемая функция */;
  sum += f * h;
 };
 n <<= 1;
} while (iter++ < 1 || std::abs(sum - sumpre) > eps);

printf("Calculated value is %.16f, %d iterations\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 точности. Для осуществления такой проверки должно быть выполнено минимум две итерации.

На каждой итерации осуществляется 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 нельзя 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++)
{
 x = a + h * (i + 0.5);
 f = /* интегрируемая функция */;
 sum += f * h;
};
```

Мы видим, что даже в такой достаточно простой вычислительной процедуре потребовалось явное объявление частных переменных. Если бы по каким-либо причинам эти переменные были сделаны 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 следует 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:

```
int n = 10;
int iter = 0;
double sum, sumpre;

do
{
 sumpre = sum;
 double h = (b - a) / n;
 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 = /* интегрируемая функция */;
  sum += f * h;
 };
 n <<= 1;
} while (iter++ < 1 || std::abs(sum - sumpre) > eps);
```

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.

1.2 Интерфейс передачи сообщений MPI

Описанию программного интерфейса MPI, помимо спецификации [36], посвящено немало изданий, в том числе русскоязычных [1, 2, 3, 31], 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.

Поскольку мы не ставим себе целью охватить все наиболее прогрессивные на сегодняшний момент подходы к параллельному программированию, мы не будем стремиться рассмотреть все тонкости и моменты, существующие в различных реализациях MPI. На текущий момент 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 Краткое описание предоставляемых функций

Здесь мы перечислим кратко основные предоставляемые программным интерфейсом MPI функции. Мы не рассмотрим эти функции детально, также мы не охватим все множество предоставляемых функций. Для этого есть немало уже написанной литературы, в частности [1, 2]. Мы лишь упомянем поверхностно задачи, выполняемые функциями предоставляемого интерфейса, с тем, чтобы была понятна его структура. Детали выполняемых функциями действий станут ясны при разборе описанных ниже примеров программ.

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 обмена, такие как формирование произвольных типов данных или произвольных групп процессов.

Прежде всего, MPI предоставляет функции парного обмена данными между процессами, т.е. функции передачи данных один-к-одному:

- Синхронные операции отправки/приема:
  MPI_Send с ее вариантами, MPI_Probe, MPI_Recv;
- асинхронные операции отправки/приема:

MPI_Isend с ее вариантами, MPI_Iprobe, MPI_Irecv;

- проверка завершения одной или более асинхронных операций отправки/приема:

  MPI_Test, MPI_Testall, MPI_Testany, MPI_Testsome;
- ожидание завершения одной или более асинхронных операций отправки/приема:

  MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome;
- отложенные отправка/прием:

  MPI_Send_init с вариантами, MPI_Recv_init, MPI_Start, MPI_Startall, MPI_Request_free;
- комбинированные операции отправки/приема:

  MPI_Sendrecv, MPI_Sendrecv_replace.

Помимо парного обмена, MPI содержит функции коллективного обмена данными:

- пересылка один-ко-многим:

  MPI_Bcast, MPI_Scatter, MPI_Scatterv;
- пересылка много-к-одному:

  MPI_Gather, MPI_Gatherv;
- пересылка много-ко-многим:

  MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Alltoallv;
- глобальная редукция, выполнение некоторой ассоциативной и, возможно, коммутативной операции над распределенными данными:

  MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter, MPI_Scan;
- барьерная синхронизация:

  MPI_Barrier, относится к коллективным коммуникациям, хотя не является операцией передачи данных в явном виде.

Все коммуникационные операции осуществляются в контексте некоторой группы, связанной с указанным при вызове операции коммуникатором. Для манипуляций группами и коммуникаторами существуют следующие функции:

- получение информации о группах процессов:

  MPI_Group_size, MPI_Group_rank, MPI_Group_translate_ranks и др.;
- создание и уничтожение групп:

  MPI_Comm_group, MPI_Group_union,

MPI_Group_intersection, MPI_Group_difference,
MPI_Group_incl, MPI_Group_excl, MPI_Group_free;

- получение информации о коммуникаторах:
  MPI_Comm_size, MPI_Comm_rank, MPI_Comm_compare;
- создание и уничтожение коммуникаторов:
  MPI_Comm_create, MPI_Comm_dup, MPI_Comm_split,
  MPI_Comm_free;
- функции работы с так называемыми интер-коммуникаторами:
  MPI_Intercomm_create, MPI_Intercomm_merge,
  MPI_Comm_test_inter, MPI_Comm_remote_group,
  MPI_Comm_remote_size.

Дополнительно к описанным функциям работы с коммуникаторами, MPI также содержит функции для работы с виртуальными топологиями, которые также представляются коммуникаторами и предназначены, прежде всего, для более удобной адресации процессов в программе:

- функции работы с многомерной решеткой в декартовых координатах:
  MPI_Cart_create, MPI_Cart_coords, MPI_Cart_rank,
  MPI_Cart_shift, MPI_Cart_sub и др.;
- функции работы с произвольным графом:
  MPI_Graph_create, MPI_Graph_neighbors_count,
  MPI_Graph_neighbors и др.

В рамках коммуникаций процессов между собой могут передаваться данные как предопределенных типов, так и более сложных определенных пользователем типов данных. Для работы со сложными типами данных MPI предоставляет следующие функции:

- создание и освобождение составных типов данных:
  MPI_Type_contiguous, MPI_Type_vector, MPI_Type_indexed,
  MPI_Type_struct, MPI_Type_commit, MPI_Type_free;
- формирование из произвольных данных с произвольным размещением непрерывной области данных для передачи между узлами, а также извлечение их на приемной стороне:
  MPI_Pack_size, MPI_Pack, MPI_Unpack.

Наконец, программный интерфейс содержит функции работы со средой выполнения MPI, обработчиками ошибок, а также механизм профилирования программ, т.е. оценки производительности различных частей программы с точки зрения работы вызываемых функций MPI.

Разумеется, здесь перечислены далеко не все функции, предоставляемые MPI. Для ознакомления с остальными функциями, также как и для получения более детальной информации о перечисленных, рекомендуется обращаться к спецификации [36], либо к специально посвященной этой теме литературе [1, 2, 3, 31].

1.2.2 Снова ряд Лейбница

Приведем простой пример использования библиотеки MPI на примере уже рассмотренного нами ранее частичного вычисления ряда Лейбница. Последовательное вычисление суммы первых n членов ряда выглядит следующим образом:

```
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) делится нацело на количество вычисляющих процессов N. Тогда мы можем распределить нагрузку между процессами равномерно. В иных случаях требуется неравномерное распределение нагрузки, некоторые возможные варианты которой будут рассмотрены позже.

Будем осуществлять распараллеливание по той же схеме, что и ранее – одинаковыми интервалами по n/N итераций (рис. 1). Отличие заключается лишь в том, что сейчас итерации распределяются не по потокам, а по процессам. Такую распределенную схему вычисления с использованием функций MPI иллюстрирует следующий код:

```
int main(int argc, char *argv[])
{
 int n;
 MPI_Init(&argc, &argv);
 if (argc == 2 && (n = atoi(argv[1])) > 0)
 {
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (n % size == 0)
  {
   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;
   if (rank == 0)
    cout << setprecision(20) << "calc = " << sum << endl;
  };
 };
 MPI_Finalize();
 return 0;
}
```

Здесь функция main приведена практически полностью, включая все необходимые вызовы функций MPI общего назначения (исключен лишь вывод сообщений об ошибках). В дальнейшем эти вызовы будут опускаться, вплоть до вызовов MPI_Comm_size и MPI_Comm_rank, будут приводиться лишь фрагменты кода, иллюстрирующие непосредственно описываемый механизм взаимодействия с интерфейсом MPI.

Первым делом в представленном фрагменте осуществляется инициализация интерфейса MPI с передачей аргументов функции main. Далее задается количество вычисляемых членов ряда. Оно в общем случае может быть прописано в программе константой, может задаваться вводом пользователя, параметром командной строки, из файла конфигурации или каким-либо еще способом. После этого с помощью функции MPI_Comm_size выполняется получение количества процессов в группе коммуникатора MPI_COMM_WORLD, а также 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 проверяем остаток от соответствующего деления на равенство нулю.

Наконец, мы вычисляем размер nloc интервала итераций для текущего процесса. В данном 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 суммы ряда все полученные результаты должны быть просуммированы. Для этого используется операция глобальной редукции с параметром MPI_SUM. Во время вызова функции MPI_Allreduce значения переменных locsum из всех процессов коммуникатора MPI_COMM_WORLD суммируются, после чего полученное значение рассылается снова во все процессы и помещается в переменную sum. Значение этой переменной увеличивается в четыре раза, в результате чего все процессы хранят в переменной sum значение, являющееся аппроксимацией числа π, которое может быть использовано ими в дальнейшем.

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

Как мы видим, приведенная программа получилась гораздо более громоздкой, нежели в случае распараллеливания с помощью OpenMP. Однако здесь следует учитывать, что задача распараллеливания выполнения между процессами гораздо более трудоемка, чем задача распараллеливания между потоками одного процесса, и в случае, если бы мы и здесь попытались привести аналогичный по функциональности код с использованием средств более низкоуровневого межпроцессного взаимодействия, он бы получился гораздо более объемным.

Приведенная программа выполняет, как уже было сказано, равномерное распределение нагрузки между процессами. Далеко не всегда такой подход удобен, поэтому далее мы опишем возможные варианты неравномерной нагрузки.

1.2.3 Неравномерное распределение вычислений

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

Рассмотрим, к примеру, возможные варианты распределения приблизительно одинаковых по длительности независимых итераций цикла. Существует два основных подхода к распределению нагрузки в подобных ситуациях – блочное и циклическое распределение [1]. Также существует блочно-циклическое распределение, которое является комбинацией обоих подходов, однако мы его здесь не будем рассматривать. При блочном распределении итерации распределяются на каждый процесс последовательными интервалами параметра цикла:

```
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 все итерации распределяются по процессам последовательным чередованием, или прореживанием по номеру итерации с шагом, равным количеству процессов:

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

К примеру, у нас есть задача, состоящая из десяти независимых подзадач 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 на однородных системах в тех случаях, когда количество подзадач не кратно количеству процессов. Иначе говоря, достигается наименьшая разница между занятостью отдельных процессов. Именно поэтому во избежание существенной неравномерности загрузки процессов зачастую прибегают к циклическому распределению подзадач.

Однако по различным причинам блочное распределение зачастую бывает предпочтительнее циклического, как с точки зрения программирования, так и с точки зрения производительности. Такая ситуация зачастую может возникнуть в силу специфики конкретной задачи. К примеру, если 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. Также при решении некоторых конкретных задач скорость вычислений 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:

$$n = aN + b, \quad 0 \le b < N. \tag{2}$$

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

$$n_i = \begin{cases} a+1, & i < b \\ a, & i \ge b \end{cases} \tag{3}$$

Левая граница каждого интервала $n^l{}_i, i = 0, \ldots, N-1$ может быть вычислена на основе тех же соотношений:

$$n^l{}_i = \sum_{k=0}^{i} n_k - n_i = ai + \begin{cases} i, & i < b \\ b, & i \ge b \end{cases} \tag{4}$$

При этом на каждый процесс распределяется интервал итераций $\left[n^l{}_i; n^l{}_i + n_i\right)$. Описанную схему иллюстрирует следующий код:

```
// вычисление nloc и левой границы интервала индексов
int nloc, nleft;
nloc = n / size + (rank < (n % size) ? 1 : 0);
nleft = (n / size) * rank + (rank < (n % size) ? rank : n % size);
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 $\{\{0,1,2\},\{3,4,5\},\{6,7\},\{8,9\}\}$.

В некоторых случаях нам 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 распределение нагрузки на узлы вычислительной системы снова ложится на плечи программиста. В общем случае это задача очень непростая, однако в некоторых частных случаях, которых весьма немало, можно приблизительно оценить производительность каждого узла и распределить нагрузку в соответствии с выполненными оценками. Такая оценка может быть сделана перед началом вычислений на основе замера времени выполнения всеми узлами некоторой небольшой типичной для всей задачи подзадачи-образца, многократное выполнение которой занимает подавляющее время в ходе вычислений.

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 у нас стоит задача умножения матрицы на вектор, нужная оценка может быть выполнена на основе замера времени вычисления каждым узлом скалярного произведения.

После того, как распределяющий процесс получил результаты замеров со всех узлов, он может выполнить распределение большого количества подзадач по узлам в соответствии с полученными оценками производительности.

Пусть каждый процесс выполнил свою подзадачу-образец за время $T_i, i = 0,\dots N-1$. Тогда скорость выполнения таких задач каждым процессом в единицу времени составляет $1/T_i$. Суммарная скорость выполнения задач всеми процессами одновременно равна $\sum_{i=0}^{N-1} 1/T_i$. Тогда, если за весь период работы совокупности процессов должно быть выполнено n задач, распределенное на конкретный процесс количество должно быть равно приблизительно:

$$n_i \approx \frac{1/T_i}{\sum_{k=0}^{N-1} 1/T_k} n. \tag{5}$$

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

Правая $n^r{}_i$ и левая $n^l{}_i$ границы каждого интервала могут быть вычислены из полученных значений n_i в виде частичной суммы:

$$n^r{}_i = \sum_{k=0}^{i} n_k, \quad n^l{}_i = n^r{}_i - n_i, \quad i = 0,\dots,N-1. \tag{6}$$

При этом из всех итераций $[0;n)$ на каждый процесс распределяется интервал итераций $\left[n^l{}_i; n^r{}_i\right), i = 0,\dots,N-1$. Описанный механизм иллюстрирует код ниже:

```
double speed, allspeed;
int nloc, allnloc, nright;
// вычисление скорости
speed = MPI_Wtime();
```

```
// ... оценочная итерация
speed = MPI_Wtime() - speed;
speed = 1.0 / speed;
// 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, 0, MPI_COMM_WORLD);
if (rank == 0 && 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 могут возникать. К примеру, может подвести выбор элементарной операции для оценки. Однако даже в такой ситуации это расхождение в неоднородных системах, как правило, будет гораздо ниже, нежели при равномерном по числу подзадач распределении нагрузки. Иные же пути оценки производительности (к примеру, по отношению тактовых частот) могут еще менее соответствовать реальности вследствие различий, к примеру, во внутренней архитектуре процессоров. Оптимальным здесь, вероятно, является определение соотношений вычислительных скоростей опытным путем на реальных задачах с последующим сохранением этих величин в качестве конфигурационных параметров для использования в дальнейших схожих вычислениях. Подобная экспериментальная схема оценки близка к методам моделирования наподобие Монте-Карло, которые также находят широкое применение в практике, несмотря на потенциальную неточность при малом количестве экспериментов.

Все рассмотренные пути распределения нагрузки являлись статическими, т.е. распределение нагрузки в них происходило один раз на весь период выполнения. На практике зачастую также используется динамическое распределение. В такой ситуации распределение очередных подзадач на процессы производится динамически по мере выполнения ими предыдущих, к примеру, неким управляющим процессом. Такое распределение наиболее удобно для достижения одновременного завершения выполнения задачи, особенно в неоднородных системах. Однако динамическое распределение, как правило, требует гораздо больше коммуникаций между процессами и соответствующих потерь времени.

Зачастую некоторому процессу бывает проще выполнить какую-либо работу самому, нежели передавать данные для выполнения этой работы другому процессу и получать 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 выделяемой за один заход подзадачи с соответствующим сокращением коммуникаций.

Однако существуют задачи, в которых 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. Как правило, в таких ситуациях удобно выделить один процесс, который будет заниматься распределением подзадач на остальные процессы по мере выполнения с последующим сбором результатов. Подробнее о вариантах динамической нагрузки можно прочитать в [31].

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 текущего раздела.

1.2.4 Умножение матрицы на вектор

Рассмотрим теперь еще одну задачу, необходимость решения которой, в отличие от вычисления ряда Лейбница, 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, следует отметить, что, когда речь идет о распараллеливании программы в системе с распределенной памятью, бывает необходимо использовать возможность распределения данных задачи между узлами, 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 два наиболее простых из них. Для простоты описания будем предполагать, что размерность матрицы кратна количеству процессов. Величину $m = n/N$ будем далее называть локальной размерностью.

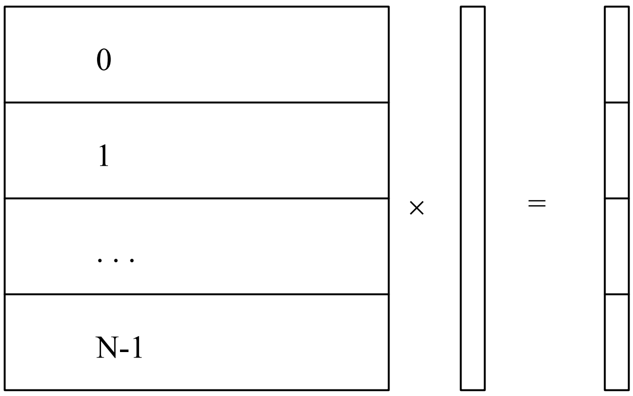

Рис. 2

В первом рассматриваемом нами случае элементы матрицы распределяются по процессам горизонтальными блоками (рис. 2). 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 по правилу умножения блочных матриц 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 должен быть скопирован в память каждого процесса для выполнения дальнейших итераций. Возможность объединить все части вектора и скопировать 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(1), auloc.vsize(), MPI_DOUBLE,
 &au(1), 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 <class e_t>
class matrix_type
{
public:
 typedef e_t element_type;
 // конструктор
 matrix_type(int vsize, int hsize);
 // конструктор копирования
```

```
 matrix_type(const matrix_type &src);
 // деструктор
 ~matrix_type(void);
 // размеры по вертикали и горизонтали
 int vsize(void) const;
 int hsize(void) const;
 // обращение к элементам по индексам (нумерация с единицы)
 const element_type & operator ()(int i, int j) const;
 element_type & operator ()(int i, int j);
 // присваивание матриц одинаковых размеров
 matrix_type & operator =(const matrix_type &src);
 // 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 <class 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
 vector_type(int vsize);
 // конструктор - преобразование типа
 vector_type(const base_type &src);
 // обращение к элементам по индексу (нумерация с единицы)
 const element_type & operator ()(int i) const;
 element_type & operator ()(int i);
 // присваивание
 vector_type & operator =(const base_type &src);
};
```

Основное необходимое требование к реализации – последовательное построчное хранение элементов в памяти. Это требуется для обеспечения возможности обмена данными с использованием функций MPI. По тем же причинам операция обращения к элементу по индексу возвращает ссылку непосредственно на хранимый элемент. Индексация элементов матриц и векторов – с единицы. Это противоречит существующим устоям программирования, нагромождает код излишними приращениями индексов и делает его менее читабельным и удобным. Однако это отвечает математическим устоям, поэтому здесь для соблюдения математической корректности мы будем использовать нумерацию элементов с единицы. Полный текст примера реализации таких шаблонов классов можно найти в приложении 1.

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

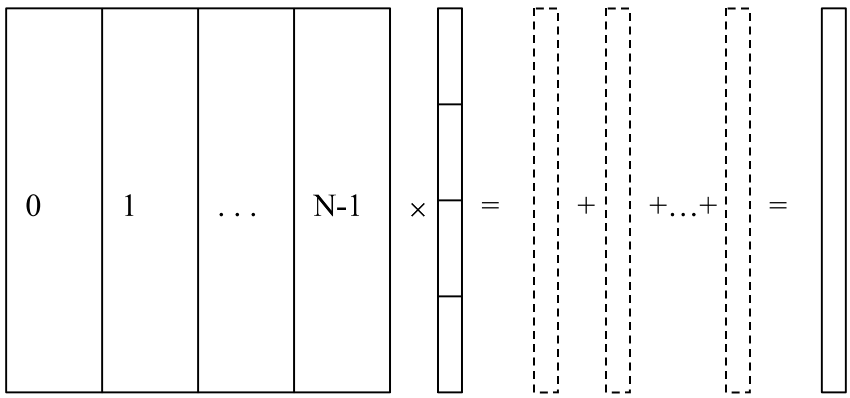

Рис. 3

Во втором рассматриваемом нами случае элементы матрицы распределены по процессам вертикальными блоками (рис. 3). Каждому процессу на каждой итерации требуется наличие одного прямоугольного $n \times m$ блока матрицы и одного локального блока умножаемого вектора. Каждый процесс выполняет перемножение обеих своих локальных частей, после чего для получения требуемого результата, по правилу умножения блочных матриц, полученные полноразмерные векторы из всех процессов должны быть просуммированы и снова распределены по процессам. Для этого мы можем воспользоваться функцией MPI_Allreduce, однако по той причине, что, как правило, для дальнейших вычислений каждому процессу необходима лишь часть вектора результата, мы можем использовать 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 = 1; i <= nlocs.vsize(); i++)
 nlocs(i) = nloc;
// суммирование всех aupart и рассеивание результата
MPI_Reduce_scatter(
 &aupart(1), &auloc(1), &nlocs(1),
 MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Реализованное описанными путями параллельное умножение матрицы на вектор может быть легко использовано при решении СЛАУ итерационными методами. Для выполнения последовательного решения СЛАУ $\bar{u} - A\bar{u} = \bar{f}$ с неизвестным вектором $\bar{u}$ методом простой итерации вполне достаточно следующей программы:

```
// матрица
matrix_type<double> a(n, n);
for (int i = 1; i <= a.vsize(); i++)
 for (int j = 1; j <= a.hsize(); j++)
  a(i, j) = /* ... инициализация матрицы */;
// вектор свободных коэффициентов
vector_type<double> f(n);
for (int i = 1; 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 < 20; i++)
 u = a * u + f;
```

Здесь для наглядности выполняется фиксированное количество итераций. Вообще же обычно используют оценку точности текущего приближения по величине относительной невязки.

Для распараллеливания такой программы одним из приведенных выше способов достаточно использовать вместо объявления полноразмерных матрицы и вектора объявление лишь необходимых локальных блоков, 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 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 < 20; i++)
{
 vector_type<double> uloc = aloc * u;
 MPI_Allgather(
  &uloc(1), uloc.vsize(), MPI_DOUBLE,
  &u(1), nloc, MPI_DOUBLE,
  MPI_COMM_WORLD);
 u += f;
};
```

Можно заметить, что в приведенном фрагменте каждым процессом выполняются отчасти лишние вычисления. 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 примет следующий вид:

```
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 < 20; i++)
{
 vector_type<double> u(n);
 MPI_Allgather(
  &uloc(1), uloc.vsize(), MPI_DOUBLE,
  &u(1), nloc, MPI_DOUBLE,
  MPI_COMM_WORLD);
 uloc = aloc * u + floc;
};
```

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 разбиение матрицы бывает удобно производить как по горизонтали, так и по вертикали. К примеру, такая ситуация будет описана далее. В таком случае схема умножения распределенной матрицы на вектор может быть получена путем комбинирования описанных приемов.

1.2.5 Перемножение матриц

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

Касательно распределенного хранения возможен вариант, когда между машинами распределяется хранение лишь одной матрицы, копия же второй присутствует в памяти каждого процесса целиком. Такая программа, разумеется, будет работать наиболее эффективно, 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 случаи распределенного хранения обеих матриц, поскольку именно они наиболее интересны при выполнении вычислительных задач в распределенных системах. Однако следует осознавать, что такой подход потребует повышенного обмена информацией между узлами.

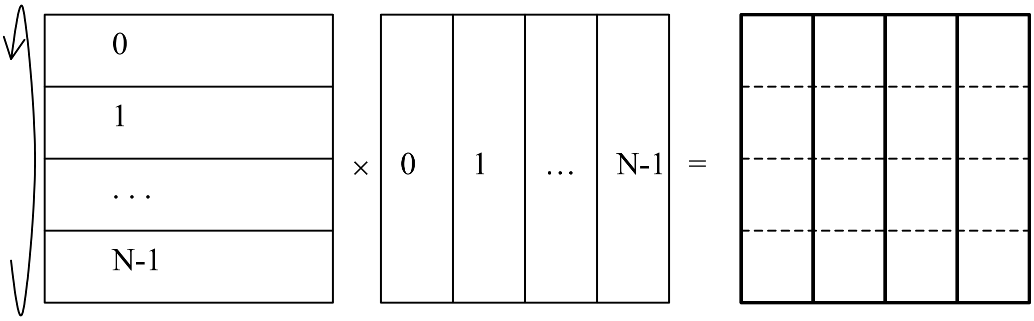

Рис. 4

Первым делом рассмотрим наиболее простой вариант. Допустим, требуется умножить слева матрицу MR на матрицу ML и получить матрицу результата RES:

$$RES = ML \cdot MR. \tag{7}$$

Тогда будем хранить матрицу ML горизонтальными блоками, MR – вертикальными (рис. 4). Будем снова для простоты считать, что размерность перемножаемых матриц кратна количеству процессов N и локальная размерность $m = n/N$. Каждый процесс содержит один блок $n \times m$ матрицы MR и один блок $m \times n$ матрицы ML. Помимо этого, каждый процесс выделяет область памяти для хранения блока матрицы результата перемножения RES. 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 RES, после чего осуществляется циклический сдвиг блоков левой матрицы ML между процессами. Если бы мы в каждом процессе хранили не вертикальный, а горизонтальный блок результата, следовало бы циклически сдвигать блоки правой матрицы MR. После сдвига выполняется следующий шаг, на котором прямоугольный блок матрицы результата RES пополняется еще одним квадратным блоком. На рис. 5 изображено распределение результатов умножения после первых двух шагов между четырьмя процессами.

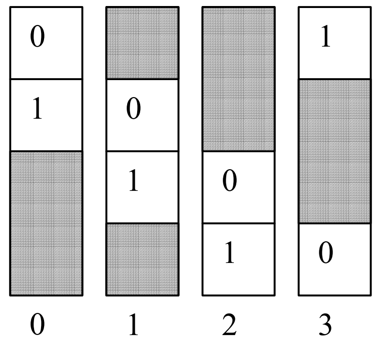

Рис. 5

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.

Всю описанную процедуру иллюстрирует следующий код:

```
int nloc = n / size;
// правая матрица (локальная часть)
matrix_type<double> mtx1loc(n, nloc);
// ... инициализация правой матрицы
// левая матрица (локальная часть)
matrix_type<double> mtx2loc(nloc, n);
// ... инициализация левой матрицы
// матрица - результат умножения (локальная часть)
matrix_type<double> resloc(n, nloc);

// вычисление рангов следующего и предыдущего процессов
int next = (rank + 1) % size;
int prev = (rank + size - 1) % size;
// выполнение умножения mtx1 на mtx2 слева (res = mtx2 * mtx1)
for (int i = 0; i < size; i++)
{
 // умножить вертикальный mtx1loc на горизонтальный mtx2loc
 matrix_type<double> mloc = mtx2loc * mtx1loc;
 // сохранить квадратный блок результата в resloc
 memcpy(
  &resloc(((rank + i) % size) * nloc + 1, 1),
  &mloc(1, 1),
  mloc.vsize() * mloc.hsize() * sizeof(double));
 // сдвинуть mtx2loc между процессами
 MPI_Status status;
 MPI_Sendrecv_replace(
  &mtx2loc(1, 1), mtx2loc.vsize() * mtx2loc.hsize(), MPI_DOUBLE,
  prev, (rank + i) % size,
```

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

После выполнения на каждом шаге перемножения двух прямоугольных блоков и копирования полученного результата в блок матрицы RES выполняется циклический сдвиг блоков матрицы ML между процессами по кольцевой топологии. 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 не является во всех отношениях оптимальным. В частности, операция копирования области памяти здесь излишняя, также как и само формирование дополнительной области памяти для хранения квадратного блока. Можно, миновав использование определенного для объекта матрицы оператора умножения, осуществить умножение блоков «вручную» с сохранением результата сразу в блок матрицы RES.

Кроме того, если программа работает на пределе использования памяти, во время циклического 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 все N процессов будут содержать N вертикальных блоков матрицы результата. В процессе выполнения умножения будет произведено N пересылок областей памяти.

Теперь рассмотрим другой вариант умножения. Он немногим более сложен для реализации, однако позволяет сократить количество пересылок областей памяти. На этот раз будем считать, что количество процессов является квадратом целого, а размерность перемножаемых матриц n кратна этому целому. Будем разбивать обе перемножаемые матрицы на квадратные блоки. На рис. 6 приведен пример размещения блоков левой и правой перемножаемых матриц, а также матрицы результата, для случая распределения по девяти процессам. Все множество процессов образует двумерную решетку, которая может быть разбита на строки процессов по горизонтали и на столбцы процессов по вертикали.

$$\begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array} \times \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline\end{array}$$

Рис. 6

Локальной размерностью будем называть величину $m = n/\sqrt{N}$, при этом каждый из N процессов хранит по одному квадратному блоку $m \times m$ из обеих перемножаемых матриц. При перемножении производится $N' = \sqrt{N}$ шагов, на каждом из которых выполняется умножение двух квадратных блоков из левой и правой перемножаемых матриц. Полученные 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 RES_{ij}:

$$\begin{gathered} RES_{ij} = \sum_{k=1}^{N'} ML_{ik} \cdot MR_{kj}, \\ rank = (i-1)N' + j - 1, \\ i, j = 1,\ldots,N'. \end{gathered} \tag{8}$$

Изначально каждый процесс хранит в своей памяти локальные блоки обеих матриц ML_{ij} и MR_{ij}. Разумеется, на каждом шаге оба перемножаемых процессом блока должны меняться. Для смены одного из блоков, к примеру, MR_{ij}, мы можем, как и в предыдущем случае, использовать циклический сдвиг в вертикальном направлении. Использовать аналогичный сдвиг и для смены блока ML_{ij} нам не позволяет тот факт, что в общем случае $i \neq j$. По этой причине мы используем рассылку блока между N' процессами от одного из них. Поясним это подробнее. Распишем результат полного умножения в каждом процессе RES_{ij} с учетом начального сдвига локальных блоков правой матрицы по процессам:

$$RES_{11} = ML_{11}MR_{11} + ML_{12}MR_{21} + \cdots + ML_{1N'}MR_{N'1};$$
$$RES_{12} = ML_{11}MR_{12} + ML_{12}MR_{22} + \cdots + ML_{1N'}MR_{N'2};$$
$$\ldots$$
$$RES_{1N'} = ML_{11}MR_{1N'} + ML_{12}MR_{2N'} + \cdots + ML_{1N'}MR_{N'N'};$$
$$RES_{21} = ML_{22}MR_{21} + \cdots + ML_{2N'}MR_{N'1} + ML_{21}MR_{11};$$
$$\ldots$$
$$RES_{2N'} = ML_{22}MR_{2N'} + \cdots + ML_{2N'}MR_{N'N'} + ML_{21}MR_{1N'};$$
$$\ldots$$
$$RES_{N'1} = ML_{N'N'}MR_{N'1} + ML_{N'1}MR_{11} + \cdots + ML_{N'N'-1}MR_{N'-11};$$
$$\ldots$$
$$RES_{N'N'} = ML_{N'N'}MR_{N'N'} + ML_{N'1}MR_{1N'} + \cdots + ML_{N'N'-1}MR_{N'-1N'};$$

Видно, что для всех RES_{ij} с одинаковым значением i, т.е. в пределах одной строки блоков, на каждом шаге блоки левой умножаемой матрицы совпадают. При этом номер в строке требуемого блока на каждом шаге $k = 1,\ldots,N'$ равен:

$$\begin{cases} i + k - 1, & i + k - 1 \le N' \\ i + k - 1 - N', & i + k - 1 > N' \end{cases} \tag{9}$$

Блок с вычисленным номером рассылается между всеми процессами, находящимися в одной горизонтали двумерной решетки, непосредственно перед выполнением перемножения двух блоков. После перемножения выполняется сдвиг блоков правой матрицы и переход к следующему шагу.

Описанный процесс иллюстрируется следующим кодом:

```
int sqrtsize = (int) floor(sqrt(1.0 * size) + 0.5);
int nloc = n / sqrtsize;
const int ndims = 2;
```

```
MPI_Comm cart, subcart;
// создаем коммуникатор с декартовой топологией
int dim[ndims] = {sqrtsize, sqrtsize}, period[ndims] = {true, true};
MPI_Cart_create(MPI_COMM_WORLD, ndims, dim, period, false, &cart);
int coords[ndims], &vcoord = coords[0], &hcoord = 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> mtx1loc(nloc, nloc);
// ... инициализация правой матрицы

// левая матрица (локальная часть)
matrix_type<double> mtx2loc(nloc, nloc);
// ... инициализация левой матрицы

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

// выполнение умножения mtx1 на mtx2 слева (res = mtx2 * mtx1)
for (int k = 0; k < sqrtsize; k++)
{
 // вычислим ранг процесса-источника для рассылки
 // блока левой матрицы среди строки процессов
 int src, dst, subroot;
 MPI_Cart_shift(cart, 1, hcoord - vcoord - k, &src, &dst);
 MPI_Group_translate_ranks(gcart, 1, &src, gsubcart, &subroot);

 // разошлем блок левой матрицы всем процессам строки
 matrix_type<double> mtx2copy(mtx2loc.vsize(), mtx2loc.hsize());
 if (rank == src)
  mtx2copy = mtx2loc;
 MPI_Bcast(
  &mtx2copy(1, 1),
  mtx2loc.vsize() * mtx2loc.hsize(), MPI_DOUBLE,
  subroot,
  subcart);

 // перемножим локальные блоки левой и правой матрицы
 matrix_type<double> mloc = mtx2copy * mtx1loc;
 // сохраним квадратный блок результата
 if (k == 0)
  resloc = mloc;
 else
  resloc += mloc;
```

```
 // вычислим следующий и предыдущий ранги
 // для сдвига по столбцу процессов
 MPI_Cart_shift(cart, 0, -1, &src, &dst);

 // сдвинем mtx1loc между процессами в столбце
 MPI_Status status;
 MPI_Sendrecv_replace(
  &mtx1loc(1, 1),
  mtx1loc.vsize() * mtx1loc.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 мы первым делом создаем коммуникатор cart с декартовой топологией, после чего выясняем координаты в нем нашего процесса. Созданная топология является двумерным тором, т.е. обе координаты обладают периодичностью. Следующим этапом с помощью функции MPI_Cart_sub все процессы разбиваются на подгруппы по горизонтальным полосам. 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. Определение производится по описанной выше схеме на основе номера шага и номера строки блоков vcoord, при этом, поскольку функция MPI_Cart_shift возвращает ранг со сдвигом относительно текущего процесса, вводится поправка на координату hcoord. Поскольку мы получаем ранг процесса в группе коммуникатора cart, нам требуется его трансляция в значение ранга в группе коммуникатора subcart, для чего используется функция MPI_Group_translate_ranks. После определения номера процесса-источника (он во всех процессах текущего коммуникатора subcart будет совпадать) производится рассылка хранимого им блока левой матрицы во все остальные процессы subcart. В качестве аргумента сдвига функции MPI_Cart_shift передается такой параметр, чтобы текущий номер по горизонтали умножаемого блока левой матрицы, который будет разослан, соответствовал текущему номеру по вертикали умножаемого блока правой матрицы.

Принятый блок левой матрицы перемножается с текущим блоком правой матрицы, после чего текущий блок RES_{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 и обрабатывать выход за пределы диапазона, что выполнялось нами в предыдущем примере. По завершении цикла производится очистка созданных коммуникаторов и групп.

На каждом шаге производится по две пересылки, поэтому во время выполнения такой операции умножения будет выполнено всего $2\sqrt{N}$ пересылок блоков памяти, что для количества процессов $N > 4$, несомненно, оказывается гораздо более эффективным по сравнению с предыдущим способом, поскольку позволяет существенно сократить накладные расходы на коммуникации.

По завершению умножения блоки распределенной по процессам матрицы результата RES_{ij} могут быть использованы в соответствии с их назначением. К примеру, может быть выполнено умножение распределенной матрицы на вектор. Схема такого умножения может быть построена путем комбинирования двух описанных ранее способов умножения распределенной матрицы на вектор.

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. Будем исходить из предположения, что решаемая задача обладает параллелизмом независимых ветвей, т.е. она состоит из некоторого количества подзадач, процесс выполнения каждой из которых не связан с другими. При этом каждая задача, возможно, по своим входным данным находится в зависимости от результатов выполнения других подзадач.

Способы решения задач, подобных описанной, относятся к методам сетевого планирования и управления [5, 12]. 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 ЯПФ

Графическое представление схемы зависимостей работ друг от друга называется сетевым графиком работ. Сетевой график работ представляется направленным ациклическим графом, т.е. никакая работа не должна прямо либо косвенно зависеть от самой себя. К примеру, на рис. 7 можно увидеть сетевой график некоторой комплексной работы, состоящей из восьми работ. Прямоугольниками здесь обозначены работы, стрелками – зависимости между ними по входным и выходным данным.

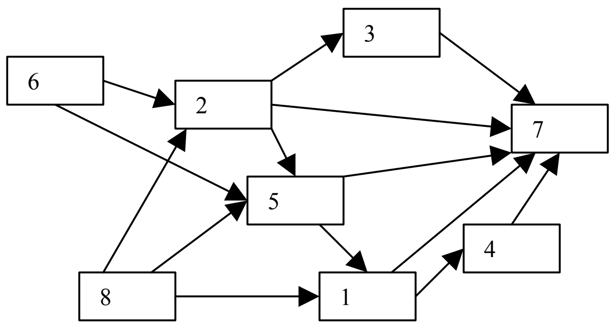

Рис. 7

В сетевом планировании используются два способа представления сетевого графика работ. В первом случае, как на рис. 7, работы представляются вершинами графа, зависимости – дугами.

Во втором случае работы представляются дугами, вершинами представляются события их завершения. Общие правила построения таких сетевых графиков приведены в [12].

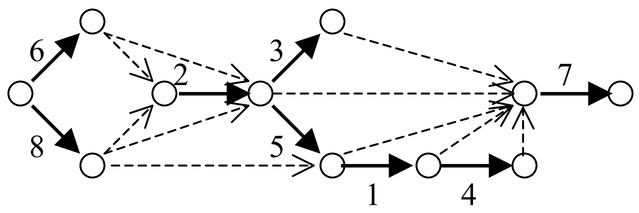

Рис. 8

Пример представления вторым способом комплекса работ, изображенного на рис. 7, приведен на рис. 8. Здесь пунктирными стрелками обозначены так называемые фиктивные работы, не занимающие ресурсов и характеризующие 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. 9).

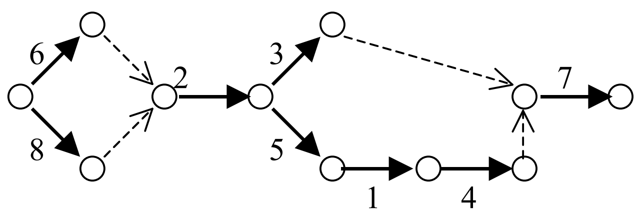

Рис. 9

Оба способа 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 (рис. 10).

Job	Dependence
1	5, 8
2	6, 8
3	2
4	1
5	2, 6, 8
6	-
7	1, 2, 3, 4, 5
8	-

Рис. 10

Решение как таковых задач сетевого планирования и управления сводится к определению оптимального распределения ресурсов с целью получения минимального времени выполнения всего комплекса на основе известных зависимостей времени выполнения отдельных работ от количества выделенных им ресурсов.

При выполнении параллельных вычислений в такой постановке рассматривать задачу оказывается затруднительно и зачастую бессмысленно, поскольку в однородной системе оказывается нечем управлять – мы не можем повлиять на скорость выполнения конкретной подзадачи, а в неоднородной оказывается сложно задать зависимости величин 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, 5, 12].

Процедура построения ярусно-параллельной формы (далее – ЯПФ) достаточно проста и заключается в представлении сетевого графика в форме последовательных ярусов работ, в пределах каждого из которых работы 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 других. 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 нему ярусов, среди которых должна быть как минимум одна работа непосредственно предшествующего яруса. На рис. 11 можно увидеть пример распределения по ярусам работ комплекса, изображенного на рис. 7.

Job	**Dependence**	**Level**
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. 11

После выполнения распределения по ярусам всех работ комплекса, сетевой график может быть изображен с учетом разбиения по ярусам. Пример разбитого по ярусам комплекса работ можно увидеть на рис. 12.

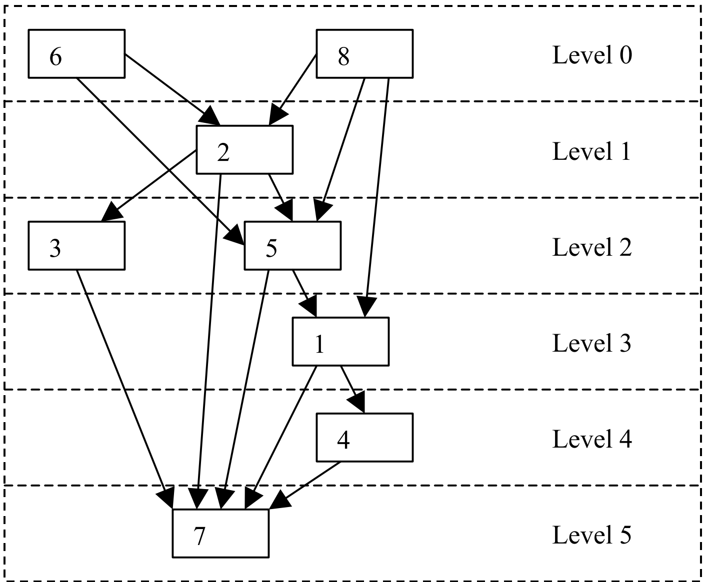

Рис. 12

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, количество ярусов – высотой ЯПФ, максимальная ширина яруса – шириной ЯПФ программы.

Эти характеристики при некоторых допущениях позволяют нам судить о свойствах построенной таким образом программы. В частности, при условии, что все работы приблизительно одинаковы по длительности, ширина ЯПФ характеризует необходимое 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.

В [1] говорится, что распараллеливание программ с использованием ЯПФ плохо согласовано с конструкциями существующих языков программирования, чем обусловлена сложность его выполнения и соответствующее крайне редкое использование. Действительно, большинство популярных сегодня языков программирования, как правило, изначально предназначены для последовательных вычислений и поэтому плохо приспособлены для любого распараллеливания. Однако следует учитывать, что написание сложной программы (а параллельная программа, как правило, является сложной) всегда требует усилий, и поиск легких путей в данном случае работает в большинстве случаев против нас. В связи с этим для построения параллельных программ мы предложим далеко не новый подход, заключающийся в отделении функциональности используемой схемы распараллеливания, а именно построения ЯПФ и выполнения как такового распараллеливания, от специфики конкретной задачи. В этом случае написание кода, отвечающего за параллельное выполнение комплекса работ, разумеется, будет не легче, чем прежде, однако может быть выполнено лишь один раз, при этом использоваться такой код может при решении широкого круга задач.

2.2 Реализация выполнения комплекса работ при использовании фиксированного количества параллельных ресурсов

Прежде чем переходить к описанию предлагаемых вариантов реализации, следует оговорить следующий момент. В [1] было указано, 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 работ предыдущего яруса.

Следует отметить, что это не совсем верно, однако рассмотрение с такой точки зрения помимо своих минусов имеет и свои плюсы. При подобной реализации нам в любой момент известно необходимое количество параллельных ресурсов, требуемых для 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 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. никакая работа следующего яруса не начинается до тех пор, пока не закончат выполнение все работы текущего яруса.

Для простоты и наглядности абстрагируемся от необходимости передачи входных и выходных данных между работами и сконцентрируемся на распределении их выполнения между параллельными ресурсами. Будем считать, что данные передаются через разделяемые ресурсы внутри работ.

В приложении 2 приведен код предлагаемых классов, распараллеленный 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 классы работ в вызывающем коде должны быть унаследованы от этого класса и должны переопределять функцию run, внутри которой должно осуществляться выполнение работы.

Другой класс – класс комплекса работ jobcomplex_type. Поскольку комплекс работ, в сущности, также является работой, класс jobcomplex_type унаследован от абстрактного 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. Для удобства добавления зависимостей указатели на работы хранятся вместе с отображением на их номера. Номера работам присваиваются в соответствии с порядком добавления. Для добавления работ и зависимостей вызывающему 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 осуществления построения, в цикле по номерам ярусов выполняются все работы. Работы каждого яруса выполняются параллельно, для чего используется директива OpenMP, при этом вследствие наличия неявного барьера до момента завершения всех работ текущего яруса следующий не начинается.

Функция build, осуществляющая построение ЯПФ, начинается с определения ярусов всех работ. С этой целью определяется 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, после чего выполняется исключение найденных работ из множества неопределенных и переход к определению работ следующего яруса.

При попытке определить номер яруса работы осуществляется проход по всем ее зависимостям. Если какая-либо работа, от которой текущая зависит, 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 номеру яруса текущей работы.

После определения номеров ярусов всех работ на основе них строится как таковая ЯПФ, представляемая в программе в виде списка списков, каждый список в котором характеризует ярус работ, а элементы – их номера.

Использование приведенных классов оказывается довольно простым и наглядным. К примеру, фрагмент программы, реализующий выполнение комплекса работ, приведенного на рис. 7, 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(j5);
jobcomplex.add_job(j3);
jobcomplex.add_job(j2);
jobcomplex.add_job(j7);
jobcomplex.add_job(j4);
jobcomplex.add_job(j6);
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 класса, унаследованного от job_abstract_type. После их создания производится создание и заполнение объекта комплекса работ, а именно добавление работ и зависимостей между ними в соответствии со структурной таблицей комплекса. Наконец, производится выполнение комплекса работ.

Рассмотрим теперь вариант модификации приведенного в приложении 2 кода для распределенного выполнения с использованием интерфейса MPI. В сущности, изменениям подверглась лишь функция run:

```
// 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);

 // выполним по очереди каждый ярус работ
 for (int l = 0; l < multilevel.size(); l++)
 {
  MPI_Barrier(MPI_COMM_WORLD);
  // распределим каждый ярус работ циклически
  for (int i = rank; i < multilevel[l].size(); i += size)
   joblist[multilevel[l][i]]->run();
 };
}
```

Для распределенного выполнения работ потребовалось внести явную барьерную синхронизацию между ярусами и циклически распределить выполнение работ каждого яруса по процессам коммуникатора MPI_COMM_WORLD.

Для использования модифицированного таким образом набора классов приведенный выше фрагмент вызывающей программы должен быть дополнен лишь вызовами MPI_Init и MPI_Finalize.

2.3 Случай неравномерной длительности 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, будут возникать существенные задержки общего времени выполнения вследствие простоя тех ресурсов, которые закончили выполнение работы раньше. В случае если при этом некоторая работа текущего яруса зависит лишь от тех работ предыдущего, которые уже завершились, она может быть уже начата и, возможно даже, завершена, однако она вынуждена ожидать достижения барьера всеми остальными.

В этой связи снимем для запуска работы условие необходимости завершения всех работ предыдущего яруса. Вместо этого вернемся к изначальному условию необходимости завершения 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.

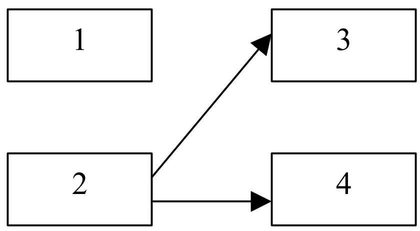

Рис. 13

Поясним это на примере. На рис. 13 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. 14 показано, как такой комплекс может выполняться во времени строго по ярусам с барьером между ними (слева) и при уплотнении с учетом индивидуальных зависимостей (справа). В таких условиях количество требуемых параллельных ресурсов в общем случае может очень существенно отличаться от ширины ЯПФ, вплоть до того, что в некоторых случаях может приближаться к общему количеству работ.

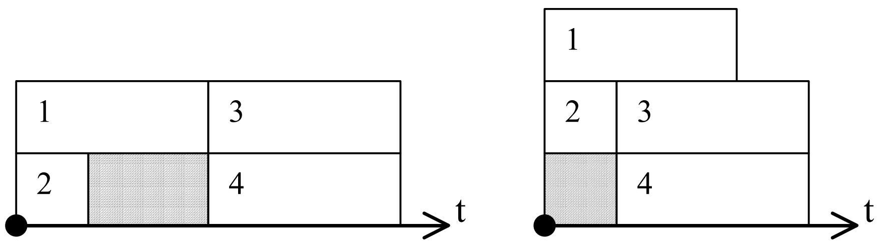

Рис. 14

В связи с неопределенностью необходимого количества параллельных ресурсов, реализация такого подхода с использованием MPI оказывается затруднительной вследствие статического существования группы процессов. Помимо этого, для подобной реализации необходимы достаточно низкоуровневые механизмы синхронизации, вследствие чего интерфейс OpenMP также оказывается неподходящим.

В этом случае может подойти какой-либо более низкоуровневый программный интерфейс, к примеру, для многопоточной реализации могут быть использованы интерфейсы Win32 API или POSIX Threads (pthreads). Мы приведем вариант многопоточной реализации с использованием интерфейса Win32 API.

Прежде всего, оговорим функциональность предлагаемого кода. Каждая работа будет выполняться в своем потоке. При этом вместо ожидания завершения каждого 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 должен уже существовать и находится 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.

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 2. Внешний интерфейс классов снова не изменяется, следовательно, код использующей их программы может оставаться прежним. Более того, как и в предыдущем случае, практически не меняется большинство реализующего классы кода. Изменилась, прежде всего, снова функция jobcomplex_type::run():

```
// 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
 std::vector<HANDLE> hdl(joblist.size());

 // список всех структур параметров
 std::vector<threadparam_type> thrparam(joblist.size());

 // запустим по очереди каждый ярус работ
 for (int l = 0; l < multilevel.size(); l++)
 {
  std::vector<HANDLE> lasthdl;

  // создадим все потоки текущего яруса
  for (int i = 0; i < multilevel[l].size(); i++)
  {
   // заполняем структуру параметров потока
   int idx = multilevel[l][i];
   thrparam[idx].phdl = &hdl[0];
   thrparam[idx].pdeps = &deplist[idx];
   thrparam[idx].pjob = joblist[idx];

   // создание потока
   DWORD dwId;
   hdl[idx] = ::CreateThread(
    NULL, 0,
    thr_proc, &thrparam[idx],
    0, &dwId);

   // добавляем в список хэндлов текущего яруса
   lasthdl.push_back(hdl[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(hdl);

 // и закрываем хэндлы
 for (int i = 0; i < hdl.size(); i++)
  ::CloseHandle(hdl[i]);
}
```

После построения необходимых структур, создаются список так называемых хэндлов (дескрипторов) потоков и список структур параметров, передаваемых каждому потоку. Список хэндлов потоков необходим для ожидания их завершения. Он заполняется по мере создания потоков, при этом в силу поэтапности создания нет риска, что какой-либо поток попытается ожидать завершения других потоков по незаполненным хэндлам. Список 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, после чего создается поток. 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.

Помимо приведенных изменений функции run, в код класса jobcomplex_type добавилось также объявление структуры параметров потока, функции потока и двух функций ожидания:

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

private:
 // структура параметров для каждого потока
 struct threadparam_type
 {
  // указатель на массив всех хэндлов
  HANDLE *phdl;

  // указатель на зависимости конкретного потока
  std::vector<int> *pdeps;

  // указатель на работу для выполнения
  job_abstract_type *pjob;
 };

 // ожидание завершения любого потока из переданного списка
 static int wait_any(const std::vector<HANDLE> &hdl)
 {
  assert(hdl.size() > 0);
  DWORD dw = ::WaitForMultipleObjects(
   hdl.size(), &hdl[0], FALSE, INFINITE);
  assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdl.size());
  return dw - WAIT_OBJECT_0;
 }
```

```
 // ожидание завершения всех потоков из переданного списка
 static void wait_all(const std::vector<HANDLE> &hdl)
 {
  if (hdl.size())
  {
   DWORD dw = ::WaitForMultipleObjects(
    hdl.size(), &hdl[0], TRUE, INFINITE);
   assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdl.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
  std::vector<HANDLE> hdl;
  for (int i = 0; i < thrp.pdeps->size(); i++)
   hdl.push_back(thrp.phdl[thrp.pdeps->at(i)]);

  // ожидание завершения работ
  wait_all(hdl);

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

  return 0;
 }

 // ...
};
```

Функция потока формирует на основе переданных полного массива хэндлов и списка зависимостей список хэндлов потоков, завершения которых текущий поток должен дождаться. После выполнения ожидания вызывается функция выполнения работы, и поток завершается.

Внутри функций ожидания в качестве передаваемого адреса массива хэндлов используется адрес первого элемента контейнера vector. Такой подход для чтения массива допустим по той причине, что, в соответствии со спецификацией STL, во-первых, все элементы контейнера vector хранятся в виде массива, во-вторых, операция индексирования возвращает ссылку на соответствующий элемент этого массива.

Глава 3. Сети конечных автоматов

В текущей 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, кроме как входными и выходными значениями на каждом такте. Таким образом, на протяжении каждого такта все автоматы могут работать параллельно.

Разумеется, для 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 [24, 25, 29], КА-технологии [19] и преобразованию алгоритмов в автоматные [28, 30], поэтому мы не будем останавливаться на ней. Скажем лишь, что есть немало областей программирования, в которых автоматный подход может существенно облегчить проектирование и реализацию.

3.1 Программирование конечных автоматов

Изложим вкратце основы автоматного программирования. 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, описания автомата различаются даже при ориентации на программную реализацию, особенно среди сравнительно недавних публикаций [9, 13].

Мы здесь приведем далеко не новое очень обобщенное описание конечного автомата [22], достаточное для понимания принципов создания сетей конечных автоматов для применения в параллельном программировании. Отметим также, что автомат Мура считается более удобным для аппаратной реализации [41], для программной же нет явных предпочтений, однако чаще используется автомат Мили или смешанный автомат Мура-Мили [27].

Конечный автомат характеризуется следующими параметрами [22]:

- множество состояний $\{S_0, S_1, \ldots, S_N\}$;
- множество совокупностей входных значений (входной алфавит);
- множество совокупностей выходных значений (выходной алфавит);
- начальное состояние S_0 (в котором автомат находится перед началом работы);
- 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 (зависимость следующего состояния от текущего состояния и текущей совокупности входных значений);
- таблица выходов (зависимость совокупности текущих выходных значений от текущего состояния и текущей совокупности входных значений).

Удобство совпадения конечного и начального состояний заключается в возможности повторного использования автомата.

Таблицы, указанные в последних двух пунктах, при некоторых допущениях полностью описывают все остальные перечисленные характеристики автомата. К примеру, на рис. 15 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 состояние, известны и они – $S_0 = A$. Входом такого автомата может быть, к примеру, двухбитное целое число или два логических значения, выход – символьный.

State	A	B	C
00	B	C	C
01	B	A	B
10	C	C	A
11	B	B	B

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

Рис. 15

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 при такой ситуации (условием перехода и действием на переходе). К примеру, на рис. 16 представлен граф переходов конечного автомата, описанного таблицами на рис. 15.

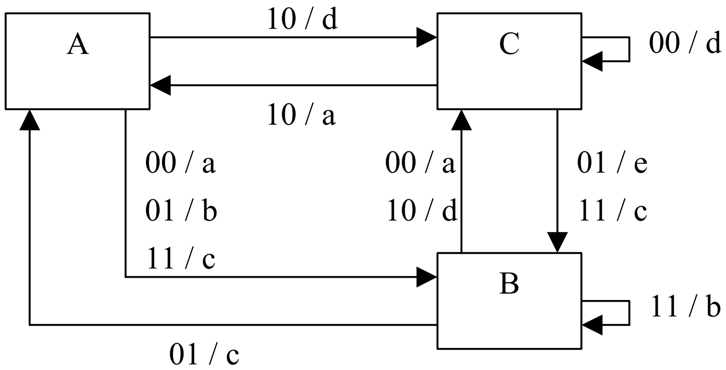

Рис. 16

Сказанное выше касалось скорее математического описания конечных автоматов. Применительно же к программной реализации следует добавить следующее. Жизненный цикл автомата представляется программным циклом со сменой состояний от начального до конечного. В теле цикла на каждом такте происходит проверка текущего состояния и входных данных, после чего осуществляется возможное изменение состояния и генерация выходных данных. В программе, разумеется, могут быть дополнительные действия, характерные для предметной области. К примеру, в управляющей программе могут быть выполнены вызовы соответствующих функций управления оборудованием.

Также в течение такта работы могут быть выполнены модификации каких-либо внутренних переменных автомата, даже в случае сохранения предыдущего состояния. Фактически, 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» большого количества схожих состояний как одного. К примеру, если автомат в цикле последовательно производит обработку элементов массива, производится 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 состояния, объединив все перечисленные состояния в единое состояние «Обработка массива», тем самым существенно упростив и программирование, и читабельность программы.

Буквы входного алфавита программного автомата могут быть не только заданы извне, но также могут быть вычислены самим автоматом на основе результатов каких-либо инкапсулированных автоматом действий. К примеру, могут быть считаны из конфигурационных файлов или базы данных.

В качестве иллюстрации приведем простой вариант реализации программного автомата, диаграмма состояний которого изображена на рис. 16.

```
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 представляется 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 значений, после чего на основе текущего состояния и считанных значений формируются и выводятся выходные значения автомата.

Автоматное программирование оказывается очень удобным при реализации широкого класса 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. 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 представленная в этом разделе автоматная реализация алгоритма суммирования сдваиванием.

Любая среда/средство/язык программирования, предназначенная для реализации каких-либо задач, должна, прежде всего, предоставлять простоту и наглядность описания стоящей задачи. Далеко не все алгоритмы описываются автоматами наглядно, несмотря на то, что, безусловно, в принципе, могут быть ими описаны.

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 графически диаграммой состояний, программный код такого автомата может быть сгенерирован автоматически соответствующими программными средствами. На сегодняшний день существует немало подобных средств генерации (к примеру, FSMDesigner, FSMGenerator и т.п.), однако мы не будем на них останавливаться.

Считается, что автоматные программы практически не требуют отладки, либо требуют ее в гораздо меньшей степени [4, 13, 19]. Отчасти это так. При классическом подходе алгоритм решения поставленной задачи, порой, пишется «на ходу», т.е. в процессе реализации много раз меняется. Тот исходный факт, что алгоритм решения задачи должен существовать в виде блок-схемы еще до начала процесса программирования, в реальности практически всегда игнорируется. В результате такого создания алгоритмов последние зачастую не имеют доказательной основы и оказываются нерабочими при некоторых исходных данных. В случае же автоматного программирования создание алгоритма в принудительном порядке отодвигается на ранние этапы разработки. Для преобразования некоторого алгоритма в автоматный исходный алгоритм должен уже быть создан и отлажен. Разумеется, после трансляции полученного таким образом автоматного алгоритма на формальный язык даже силами программиста программа будет содержать гораздо меньше ошибок, нежели при формировании алгоритма в процессе реализации. При программной же трансляции ошибок удается избежать практически полностью.

3.2 Параллелизм сетей конечных автоматов

Сами по себе реализованные программно конечные автоматы являются последовательными и, вопреки мнению некоторых авторов, 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, будь то процессы или потоки. Однако ситуация совершенно иная в отношении сетей конечных автоматов.

Сетью конечных автоматов называют некоторое количество связанных между собой входными и выходными данными (сигналами) конечных автоматов. 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. 17. Здесь изображена сеть из трех автоматов с двумя входам и четырьмя выходами. Каждый из автоматов имеет по четыре входа, при этом два из них имеют три выхода, еще один – два выхода.

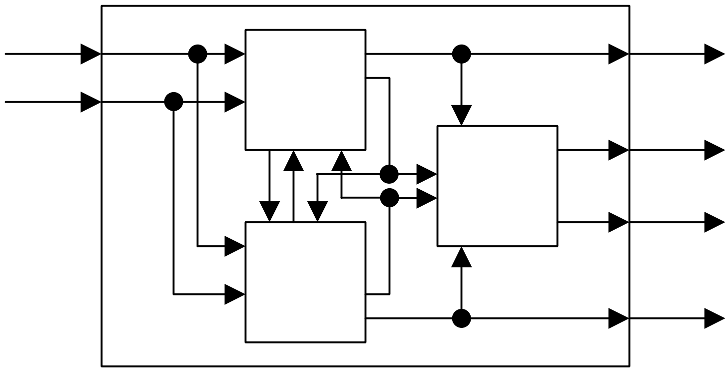

Рис. 17

С точки зрения функционирования сеть автоматов также представляет собой некий сложный автомат, состояние которого в каждый момент времени определяется совокупностью состояний внутренних автоматов сети. 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 описана ниже при описании примера программной реализации.

Все автоматы сети работают синхронно, т.е. начало работы следующего такта в каждом автомате не происходит 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 в большинстве случаев не является постоянной величиной. Естественным способом синхронизации параллельно работающих автоматов является установка барьера. В такой ситуации длительность такта работы сети определяется максимальной длительностью текущего 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 синхронизации тактов.

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).

Для более подробного ознакомления с теорией конечных автоматов рекомендуется обратиться к специально посвященной этой теме литературе [8, 22]. Помимо учебников, также доступно много достаточно современных статей [4, 13, 17, 18, 25, 27, 29]. В частности, немало публикаций можно найти в Internet на странице http://www.softcraft.ru/auto.shtml.

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 программной реализации моменты.

Прежде всего, в автоматной сети должна быть 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 перед следующим тактом. Эта область должна содержать именно «выходы» автоматов, а не «входы», вследствие типа связи «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 должно быть жестко отделено от записи выходных данных всеми автоматами. В нашем случае действия автомата разделены на три функции – чтение входных данных, как таковые внутренние действия автомата и запись выходных данных. Интерпретация данных, вычисление локальных событий, действие на переходе и 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. В нашем случае действие автомата выполняется сразу после чтения.

Будем предполагать в нашем примере, что все входные и выходные данные автоматов одного типа – целочисленные. Общая область памяти для значений выходов автоматов строится программно на основе описаний автоматов и связей их 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:

$$(NET\_IN, index) \to (FSM\_num, index);$$

$$(FSM\_num, index) \to (NET-OUT, index);$$

$$(FSM\_num, index) \to (FSM\_num, index).$$

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. 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 сети в целом), а также может быть выполнена проверка отсутствия «висящих концов» (отсутствия 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 начальной генерации программного описания конкретного автомата, могут быть созданы специальные программные средства, подобные описанным в [4].

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 не всем автоматам нужны результаты работы всех других автоматов. Иначе говоря, выходы конкретного автомата могут требоваться на входе лишь небольшому количеству других автоматов сети, остальным же рассылка его выходных значений не требуется. В таких ситуациях бывает удобно объединить независимые группы автоматов в отдельные сети, не затрагивая при этом универсальность реализации сети. Каждая независимая группа автоматов обменивается данными с другими автоматами своей группы, не производя ненужного обмена с остальными автоматами.

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 проверять факт завершения работы всеми автоматами сети оказывается чересчур накладно, поэтому может быть удобным при параллельных вычислениях переопределять процедуру проверки завершения работы сети более простыми средствами, нежели коммуникации всех процессов. К примеру, проверка завершения может осуществляться путем анализа одного из выходов сети, предназначенного для указания наличия финального результата работы сети. Здесь, правда, 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 значений.

Далее рассмотрим конкретные варианты параллельной реализации с использованием описанного подхода. Позже будут приведены примеры построения конкретных автоматных сетей с использованием описанных общих классов.

3.3.1 Реализация с использованием OpenMP

В приложении 3 приведен исходный текст последовательной реализации базовых классов конечного автомата и автоматной сети. Класс автоматной сети распараллелен с помощью директив OpenMP. Оба этих класса унаследованы от абстрактного класса конечного автомата (fsm_abstract_type), содержащего шесть функций:

```
// абстрактный тип автомата
class fsm_abstract_type
{
public:
 // тип входных и выходных данных
 typedef int data_type;
 // тип состояния автомата
 typedef int state_type;

 // количество входных каналов
 virtual int number_input(void) const = 0;
```

```
 // количество выходных каналов
 virtual int number_output(void) const = 0;
 // передать автомату вектор входных данных
 virtual void put_input(const std::vector<data_type> &input) = 0;
 // выполнить такт работы автомата
 virtual void do_work(void) = 0;
 // получить от автомата вектор выходных данных
 virtual std::vector<data_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 тип элементарного конечного автомата (fsm_type). Функции этого класса определяют в общем виде работу элементарного конечного автомата за исключением 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. Каждому состоянию, объявленному унаследованным классом, им же сопоставляется некий обработчик и добавляется в таблицу обработчиков функцией add_handler.

Во время каждого такта в зависимости от состояния вызывается тот или иной обработчик. Один обработчик может быть подключен более чем к одному состоянию. В большинстве литературы для выбора обработчиков рекомендуют использовать оператор switch или его аналоги в других языках [4, 13, 27]. Однако такая конструкция не всегда удобна при динамическом формировании автомата во время выполнения. В этой ситуации может быть использована альтернативная конструкция на основе if – else if – else, однако время поиска нужного обработчика в таком случае будет расти линейно в зависимости от количества состояний, что при большом количестве состояний обусловит длительные задержки. Поэтому мы воспользовались стандартным шаблоном map, который реализуется двоичным деревом и потому обеспечивает время поиска, пропорциональное логарифму количества состояний.

Наконец, на основе абстрактного типа автомата объявляется тип 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 вызывающим кодом на основе абстрактного класса factory_abstract_type, передается конструктору класса fsmnet_type и содержит всю необходимую информацию о создаваемой автоматной сети.

Класс fsmnet_type содержит вложенный класс shared_area_type, инкапсулирующий создание и использование общей области памяти автоматной сети для хранения промежуточных значений выходов автоматов. Создание 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 данных создаются в конструкторе класса shared_area_type на основе списка связей, переданных от фабрики сети.

Список связей формируется в фабрике сети вызывающим кодом с помощью вспомогательного класса linkstore_type, который введен для простоты описания добавления связи.

Два слова о фабрике сети. Этот класс введен по той причине, что автоматы в явном виде должны создаваться вызывающим кодом. В то же время список созданных автоматов не должен передаваться конструктору готовым, так как в целях удобства и унификации распараллеливания конструктор сам должен принимать решение о создании конкретных автоматов. Помимо этого, класс фабрики сети выполняет еще одну роль, о которой подробнее будет сказано 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.

Для создания автоматной сети в вызывающем коде должен быть объявлен набор классов автоматов, унаследованных от fsm_type, а также класс фабрики сети, унаследованный от factory_abstract_type. Во время выполнения вызывающим кодом должен быть создан объект фабрики сети, который должен существовать на протяжении всего жизненного цикла сети, после чего должен быть создан объект fsmnet_type с передачей конструктору созданного объекта фабрики сети. Объекты конкретных автоматов из объявленного набора классов должны создаваться и уничтожаться внутри функций фабрики сети create_fsm и destroy_fsm соответственно.

К примеру, объявление сети из двух автоматов может выглядеть следующим образом:

```
class fsm1_type: public fsm_type
{
public:
 enum {
  INPUT_0,
  // ... прочие объявления входов
  INPUT_NUMBER};
 enum {
  OUTPUT_0,
  // ... прочие объявления выходов
  OUTPUT_NUMBER};
 enum {
  STATE_0,
  // ... прочие объявления состояний
  };

private:
 state_type handle_off(state_type state)
 {
  if (/* ... */)
  {
   state = STATE_0;
   m_output[OUTPUT_0] = /* ... */;
   // ... заполнение оставшихся выходов
  }
  // ... обработка прочих условий
  return state;
 }
 // ... объявления прочих обработчиков

public:
```

```
 fsm1_type():
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&fsm1_type::handle_off));
  // ... добавление прочих обработчиков
 }
};

class fsm2_type: public fsm_type
{
 // ...
};

class fsmnet1_type: public fsmnet_type
{
public:
 enum {FSM_1, FSM_2, FSM_NUMBER};
 enum {
  INPUT_0,
  // ... прочие объявления входов
  INPUT_NUMBER};
 enum {
  OUTPUT_0,
  // ... прочие объявления выходов
  OUTPUT_NUMBER};

 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; }
  linkstore_type get_links(void) const
  {
   linkstore_type store;

   store.add(
    PSEUDOFSM_NETINPUT, INPUT_0,
    FSM_1, fsm1_type::INPUT_0);
   // ... прочие соединения со входами сети

   store.add(
    FSM_2, fsm2_type::OUTPUT_0,
    PSEUDOFSM_NETOUTPUT, OUTPUT_0);
   // ... прочие соединения с выходами сети

   store.add(
    FSM_1, fsm1_type::OUTPUT_0,
    FSM_2, fsm2_type::INPUT_0);
   // ... прочие межавтоматные соединения

   return store;
  }
```

```
  fsm_abstract_type *create_fsm(int i)
  {
   if (i == FSM_1)
    return new fsm1_type();
   // ... создание прочих автоматов
  }
  void destroy_fsm(int i, fsm_abstract_type *pfsm)
  {
   if (i == FSM_1)
    delete dynamic_cast<fsm1_type *>(pfsm);
   // ... уничтожение прочих автоматов
  }
 };

 fsmnet1_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 осуществлена следующим циклом:

```
// два варианта входных векторов
vector<fsm_type::data_type> vin(fsmnet1_type::INPUT_NUMBER);
vector<fsm_type::data_type> vinzero(fsmnet1_type::INPUT_NUMBER);
fill(vinzero.begin(), vinzero.end(), 0);
// ... инициализация начального вектора

// массив выходных векторов
vector<vector<fsm_type::data_type> > vouts;

// фабрика автоматной сети и сама сеть
fsmnet1_type::factory_type factory;
fsmnet1_type fsmnet(factory);

// цикл работы сети, в начальный момент выключена
bool isoff = true;
do
{
 // подать входные данные, если сеть выключена
 fsmnet.put_input(isoff ? vin : vinzero);
 // выполнить такт работы сети
 fsmnet.do_work();
 // проверить, выключена ли сеть
 isoff = fsmnet.is_off();
 // получить очередные выходные данные
 if (!isoff)
  vouts.push_back(fsmnet.get_output());
} while (!isoff);

// по завершении работы сети vouts.back() содержит
// значения выходов сети на предпоследнем такте
```

В приведенном примере на вход сети подается лишь два варианта векторов – некий начальный вектор, инициирующий работу сети, и вектор нулевых значений. После выполнения этого кода объект vouts содержит по порядку все векторы выходных значений, кроме вектора последнего такта. Вектор выходных значений последнего такта нас не интересует, поскольку после последнего такта сеть пребывает в выключенном состоянии, поэтому полезные сигналы, в т.ч. выходные, отсутствуют.

В последнем примере на каждом такте осуществляется передача входных параметров сети и изъятие выходных. В некоторых частных случаях это не является необходимым, поскольку входные параметры на протяжении всей работы сети не меняются или их изменение во время работы сети не учитывается, а выходные параметры могут отсутствовать либо нас не интересовать (к примеру, когда сеть выполняет какую-либо внутреннюю работу с инкапсулированной в автоматах передачей данных вовне, не заключающуюся в выдаче полезного результата на выход сети). В такой ситуации цикл можно было существенно упростить, исключив вызовы put_input и get_output из тела цикла:

```
fsmnet.put_input(vin);
do
 fsmnet.do_work();
while (!fsmnet.is_off());
```

В этом примере вектор входных значений помещается в сеть перед первым тактом и не меняется на протяжении всего цикла работы сети.

3.3.2 Простая реализация с использованием MPI

Приведенный в приложении 3 набор общих базовых классов сделан по возможности универсальным с точки зрения инкапсуляции создания и работы автоматов для того, чтобы использующий эти классы код подвергался минимальным изменениям при смене инструмента распараллеливания. Как видно, этот код был выполнен последовательным, после чего был распараллелен 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 будем приводить лишь те фрагменты кода, приведенного в приложении 3, которые подверглись изменению.

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

Предположим для простоты, что количество процессов в группе коммуникатора MPI_COMM_WORLD соответствует количеству автоматов сети, и на каждый автомат приходится свой процесс. Тогда в каждом процессе создается 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_fsmnum;
 // ...
};
```

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. В деструкторе, соответственно, выполняется уничтожение одного автомата с известным номером:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
public:
 fsmnet_type(factory_abstract_type &factory):
  m_factory(factory), m_shared(m_factory.get_links().get())
 {
  MPI_Comm_rank(MPI_COMM_WORLD, &m_fsmnum);
  // создание автомата
  m_pfsm.push_back(m_factory.create_fsm(m_fsmnum));
 }

 ~fsmnet_type(void)
```

```
 {
  // уничтожение автомата
  m_factory.destroy_fsm(m_fsmnum, m_pfsm.back());
 }
 // ...
};
```

Поскольку автомат в текущем процессе один, изменяется также код функции класса fsmnet_type, 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_fsmnum));
  m_pfsm.back()->do_work();
  // записать выходные (запись должна быть отделена от чтения)
  m_shared.put_output(m_fsmnum, m_pfsm.back()->get_output());
 }
 // ...
};
```

Наконец, поскольку автоматы находятся в разных процессах, в функцию проверки выключения сети вносится операция глобальной редукции:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 bool is_off(void) const
 {
  // сеть в начальном состоянии, когда все автоматы в начальном
  int isoff = m_pfsm.back()->is_off();
  int allisoff;
  MPI_Allreduce(
   &isoff, &allisoff, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);
  return allisoff;
 }
 // ...
};
```

Все эти изменения коснулись класса fsmnet_type. Однако для взаимодействия автоматов сети между собой также потребуется изменение класса shared_area_type. Поскольку после записи выходных данных каждого автомата требуется их рассылка всем остальным процессам, в конец функции записи выходов автомата добавлен вызов MPI_Allgatherv, выполняющий сбор областей выходных данных разных размеров от всех автоматов в одну область и копирование полученной области 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 std::vector<data_type> &output)
 {
  // синхронизация всех автоматов - для отделения от чтения входов
  MPI_Barrier(MPI_COMM_WORLD);
  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 коде. Необходимыми из них являются лишь вставки вызовов MPI_Init и MPI_Finalize.

3.3.3 Реализация с поддержкой вложенных сетей

Описанные модификации базовых классов с использованием MPI, как уже было оговорено выше, позволяют создавать автоматные сети, состоящие лишь из элементарных автоматов, не являющихся вложенными автоматными сетями. Вследствие этого при реализации сложной сети могут возникнуть неоправданно большие накладные расходы на зачастую ненужную передачу выходных данных между всеми процессами. Для реализации же полноценного параллелизма автоматных сетей с использованием MPI требуется внести небольшие изменения в предоставляемый описанными классами интерфейс. Как и в предыдущем случае, будем приводить лишь подвергнувшиеся изменению фрагменты кода, приведенного в приложении 3.

Требуемые изменения предоставляемого классами интерфейса диктуются следующими обстоятельствами. Каждой автоматной сети для выполнения обмена данными между 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. С этой целью к классу фабрики сети добавляется еще одна функция:

```
// абстрактный тип фабрики автоматной сети
class factory_abstract_type
{
public:
 // ...
 // полное количество параллелей в автомате
 virtual int fsm_size(int i) const = 0;
 // ...
 // создание i-го автомата в сети
 virtual fsm_abstract_type *create_fsm(int i, MPI_Comm commfull) =0;
 // ...
};
```

Добавленный параметр в функции создания автомата нужен для передачи коммуникатора конструктору fsmnet_type или fsm_type, о чем подробнее будет сказано позже. Новая функция fsm_size возвращает требуемое количество параллельных ресурсов для каждого автомата сети. Будем в дальнейшем называть это число размером автомата. Поскольку автомат может быть автоматной сетью, количество выполняющих его работу процессов может быть более одного. Более 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.

Разумеется, 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 такому подходу препятствует тот факт, что для выяснения размера конкретного автомата потребуется его создание, причем далеко не только в тех процессах, в которых он действительно должен быть создан для последующего функционирования.

Информация о требуемом размере автомата должна быть известна фабрике сети. Поскольку ей по определению известно соответствие номеров автоматов конкретным автоматам, это не составляет проблему. При создании внутренней сети ее конструктору также должен быть передан объект фабрики внутренней сети. Он может содержаться внутри объекта фабрики внешней сети, тогда информация о размере внутренней сети также будет доступна фабрике внешней. Таким образом, объект фабрики внешней сети удобно представляется «деревом» фабрик сети, т.е. содержит фабрики всех внутренних сетей (рис. 18). Именно по этой причине, поскольку такой объект дерева фабрик должен содержаться в каждом процессе, все классы фабрик должны быть «легкими» для создания. Затраты вычислительных ресурсов должны требоваться лишь при вызове функций создания/уничтожения конкретного автомата или получения списка связей.

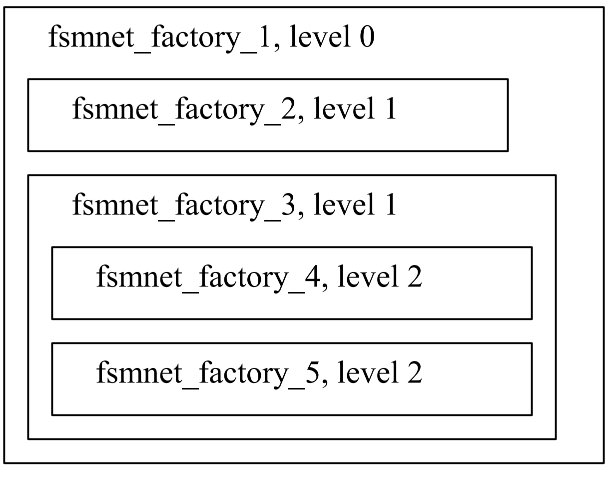

Рис. 18

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
```

```
{
 // ...
private:
 // номер автомата текущей сети, реализуемого нашим процессом
 int m_fsmnum;
 // 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.get_links().get(), *this),
  m_commfull(commfull)
 {
  // заполним массив локальных участников (нижних мастеров)
  std::vector<int> loc;
  // и локализуем свою принадлежность конкретному нижнему автомату
  int rank;
  MPI_Comm_rank(m_commfull, &rank);
  m_fsmnum = - 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.fsm_size(i);
   // запомним индекс последнего автомата, если он наш
   if (m_fsmnum < 0 && rank < cur)
    m_fsmnum = i;
  };

  // создадим коммуникатор локальных участников сети
  MPI_Group group, newgroup;
  MPI_Comm_group(m_commfull, &group);
  MPI_Group_incl(group, loc.size(), &loc[0], &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_fsmnum, rank, &m_commlower);

  // создадим нижний автомат или подсеть
  m_pfsm.push_back(m_factory.create_fsm(m_fsmnum, m_commlower));
 }

 ~fsmnet_type(void)
 {
  // уничтожение автомата
  m_factory.destroy_fsm(m_fsmnum, 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, также как и к конструктору fsm_type, добавляется дополнительный параметр, характеризующий коммуникатор, в рамках которого будут работать все нижележащие автоматы текущей сети. Объектом автомата, унаследованным от fsm_type, переданный коммуникатор может быть использован по своему усмотрению. Конструктор сети самого верхнего уровня вызывается с каким-либо определенным вне сети коммуникатором с нужным количеством процессов, т.е. равным полному размеру всей сети. К примеру, это может быть коммуникатор MPI_COMM_WORLD.

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 сети), 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. С этой целью все процессы переданного через конструктор коммуникатора m_commfull разделяются на группы, каждая из которых соответствует одному из абстрактных автоматов сети. Для выполнения этой операции путем опроса фабрики сети на предмет размеров ее автоматов предварительно выясняется номер m_fsmnum нижележащего абстрактного автомата текущей сети, который будет реализовываться данным процессом. После этого коммуникатор m_commfull разделяется на коммуникаторы m_commlower по значению номера текущего нижележащего автомата m_fsmnum.

Из каждой созданной группы выделяется нулевой процесс, ответственный за «представительство» нижележащего автомата в текущей сети. Этот же процесс (мастер) 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 называемых локальных участников. Список номеров процессов локальных участников формируется в процессе опроса размеров автоматов. На основе созданной группы локальных участников создается коммуникатор m_commlocal, в контексте которого будет производиться взаимодействие в рамках обмена выходными данными между абстрактными автоматами текущей сети.

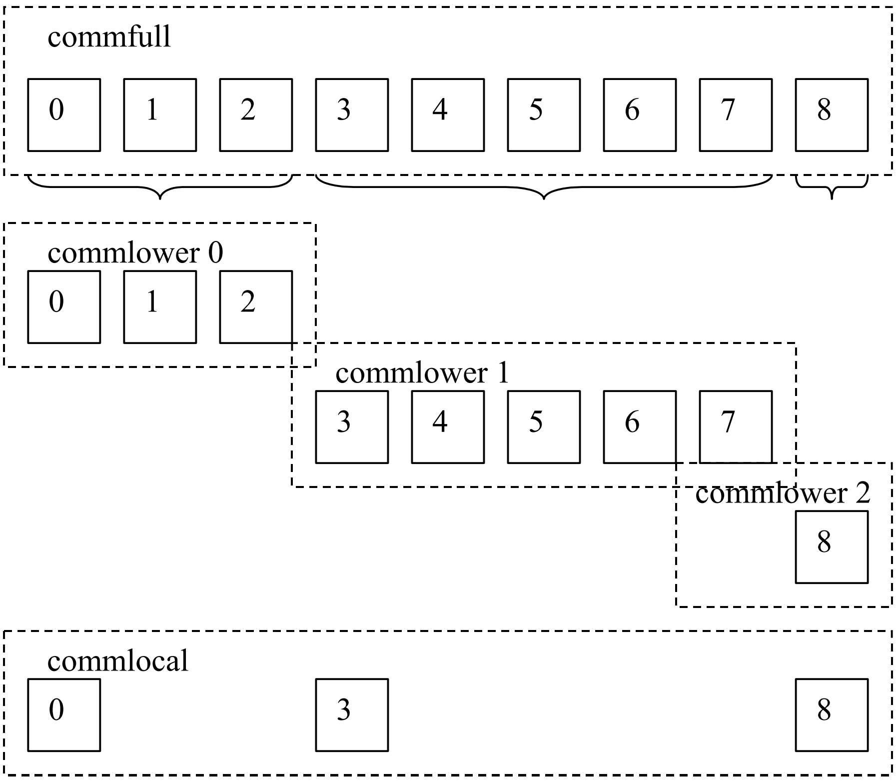

Рис. 19

Пример распределения процессов по коммуникаторам можно увидеть на рис. 19. 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 текущего автомата. Всего таких коммуникаторов три – также по количеству автоматов.

Наконец, в конце работы конструктора fsmnet_type вызывается создание нижележащего автомата с только что созданным коммуникатором m_commlower в качестве параметра, который должен быть передан создаваемому 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 нижележащего автомата, после чего производит освобождение созданных в конструкторе коммуникаторов. Коммуникатор m_commlocal освобождают только те процессы, для которых он действителен, т.е. только те, которые являются мастерами 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 простого распараллеливания с использованием MPI одноуровневой автоматной сети. Функция же проверки выключения сети, в отличие от предыдущего случая, опрашивает на предмет выключения лишь локальных участников, причем результат получает лишь мастер-процесс текущей сети:

```
class fsmnet_type: public fsm_abstract_type
{
 // ...
 bool is_off(void) const
 {
  // сеть в начальном состоянии, когда все автоматы в начальном
  int isoff = m_pfsm.back()->is_off();
  // сольем статусы от локальных участников 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;
 }
 // ...
};
```

Поскольку проверка выключения производится во всех процессах сети, результат рассылается мастером в контексте коммуникатора m_commfull.

Наконец, изменения должны коснуться функций помещения выходных данных автомата и входных данных сети в область общих данных сети. В обеих функциях используется коммуникатор локальных участников сети, который с этой целью должен быть каким-либо образом передан конструктору класса shared_area_type:

```
class shared_area_type
{
 // ...
private:
 // ссылка на сеть-владельца
 fsmnet_type &m_fsmnet;

public:
 // ...
 shared_area_type(
  const std::vector<link_type> &links, fsmnet_type &fsmnet):
  m_fsmnet(fsmnet)
 {
  // ...
 }
 // сохранение выходных данных конкретного автомата
 void put_output(int i, const std::vector<data_type> &output)
 {
  // рассмотрению подлежат только локальные участники
  if (m_fsmnet.m_commlocal != MPI_COMM_NULL)
  {
   // синхронизация всех автоматов - для отделения от чтения входов
   MPI_Barrier(m_fsmnet.m_commlocal);
   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 std::vector<data_type> &input)
 {
  // рассмотрению подлежат только локальные участники
  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)
    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 участником в текущей сети, т.е. мастером нижележащего автомата. В остальном функция put_output отличается от приведенной выше для простой реализации одноуровневой сети лишь коммуникатором. Функция же put_input сохраняет переданные входные данные сети лишь в случае, когда текущий процесс является мастером текущей сети, т.е. нулевым среди всех процессов переданного извне коммуникатора m_commfull. Этот же процесс по построению является нулевым для коммуникатора m_commlocal. После сохранения входных данных сети они должны быть разосланы всем процессами локальных участников.

Приведенное описание модифицированных классов позволяет строить распределенные автоматные сети с многоуровневым вложением и внутренним распараллеливанием элементарных автоматов.

3.4 Бильярдные шары

Рассмотрим теперь варианты реализации конкретных задач с использованием приведенных классов. Задача, выбранная в качестве первого примера, названа бильярдом, хотя на самом деле упрощена настолько, что называться так не должна. Тем не менее, эта задача даже в таком упрощении весьма подходит для автоматной реализации. Более того, при попытке реализации ее классическими методами, так или иначе, в большинстве случаев будет получен алгоритм, близкий к автоматному.

Реализуемая задача заключается в отображении некоего стола, по которому без потери скорости катаются шары. Эти шары могут отталкиваться от границ стола и сталкиваться друг с другом. При этом, разумеется, направление движения шаров меняется. Удобство реализации такой задачи в виде автоматной сети основывается на том, что в каждый момент времени шару для принятия решения о направлении дальнейшего движения необходима информация о своем текущем направлении, а также о направлении движения и координатах других шаров.

Поскольку в наши цели входит лишь иллюстрация вариантов использования автоматов и сетей, мы снимем множество условий, приближающих задачу к реальности, чем сильно упростим задачу и сделаем ее реализацию более наглядной. Реализуем задачу для случая стола с двумя шарами. Будем считать, что шары двигаются по клеткам под углом 45 градусов к границам стола в одном из четырех возможных в этом случае направлений. При столкновении с границей стола каждый шар меняет направление естественным образом. При столкновении шаров между собой оба продолжают двигаться дальше в противоположном направлении (рис. 20).

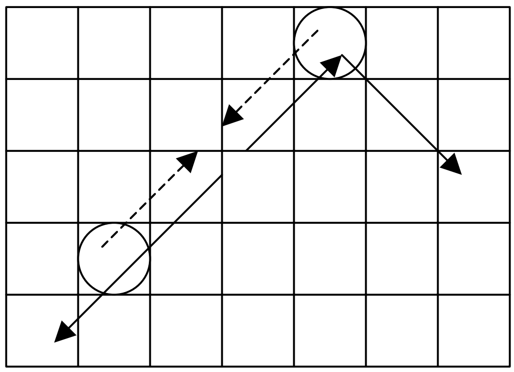

Рис. 20

Будем реализовывать автоматную сеть в виде трех автоматов (рис. 21). Один автомат играет роль стола. В его задачи входит получение данных о текущих координатах шаров и соответствующее отображение какими-либо средствами. Также на входе стол принимает количество циклов выполнения. На выходе стол выдает информацию о своих границах – левой, правой, верхней и нижней. Еще два автомата играют роль шаров, каждый из которых принимает инициализационные данные о своих координатах и направлении движения, данные о границах стола и текущие координаты другого шара. Передача информации о границах стола в явном виде на каждом такте позволяет использовать стол с двигающимися границами, хотя мы эту возможность использовать не будем.

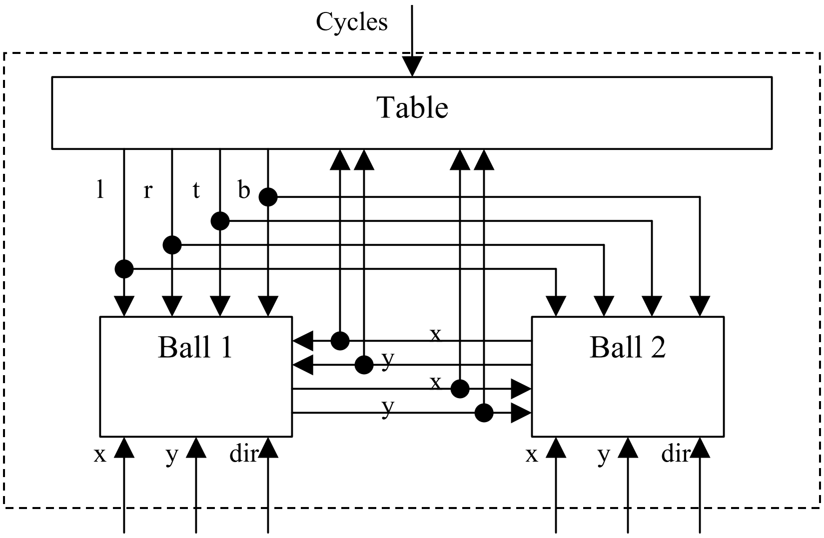

Рис. 21

Код класса автомата, реализующего поведение шара, может быть следующим:

```
class ball_type: public fsm_type
{
public:
 enum {
  INPUT_INIT_X,
  INPUT_INIT_Y,
  INPUT_INIT_DIR,
  INPUT_BOUND_LEFT,
  INPUT_BOUND_RIGHT,
  INPUT_BOUND_TOP,
  INPUT_BOUND_BOTTOM,
  INPUT_OTHER_X,
  INPUT_OTHER_Y,
  INPUT_NUMBER
 };
 enum {OUTPUT_X, OUTPUT_Y, OUTPUT_NUMBER};
 enum {
  STATE_UP_LEFT,
  STATE_UP_RIGHT,
  STATE_DOWN_LEFT,
  STATE_DOWN_RIGHT,
  DIR_NUMBER
 };

private:
```

```
// текущие координаты
int m_x, m_y;

state_type do_start(state_type state)
{
 // инициализация координат и направления
 m_x = m_input[INPUT_INIT_X];
 m_y = m_input[INPUT_INIT_Y];
 state = m_input[INPUT_INIT_DIR];
 // выводим координаты
 m_output[OUTPUT_X] = m_x;
 m_output[OUTPUT_Y] = m_y;
 return state;
}
state_type do_move(state_type state)
{
 // вычисляем изменения координат
 int dx, dy;
 dx = (state == STATE_UP_LEFT || state == STATE_DOWN_LEFT)? -1 : 1;
 dy = (state == STATE_UP_LEFT || state == STATE_UP_RIGHT) ? -1 : 1;

 // ищем возможное направление
 bool found = false;
 for (int i = 0; !found && i < DIR_NUMBER; i++)
 {
  // проверим возможность столкновения с другим шаром
  if (m_x + dx == m_input[INPUT_OTHER_X] &&
   m_y + dy == m_input[INPUT_OTHER_Y])
  {
   // меняем направление на противоположное
   dx *= -1;
   dy *= -1;
  }
  // проверим возможность столкновения с границами
  else if (m_x + dx <= m_input[INPUT_BOUND_LEFT])
   dx *= -1;
  else if (m_x + dx >= m_input[INPUT_BOUND_RIGHT])
   dx *= -1;
  else if (m_y + dy <= m_input[INPUT_BOUND_TOP])
   dy *= -1;
  else if (m_y + dy >= m_input[INPUT_BOUND_BOTTOM])
   dy *= -1;
  else
   found = true;
 };

 // останавливаемся, если двигаться некуда
 if (!found)
  state = STATE_OFF;
 else
 {
  if (dx * dy > 0)
   state = (dx > 0) ? STATE_DOWN_RIGHT : STATE_UP_LEFT;
```

```
   else
    state = (dx > 0) ? STATE_UP_RIGHT : STATE_DOWN_LEFT;
  };

  // если не выключаемся
  if (state != STATE_OFF)
  {
   // меняем текущие координаты
   m_x += dx;
   m_y += dy;
   // выводим их
   m_output[OUTPUT_X] = m_x;
   m_output[OUTPUT_Y] = m_y;
  }
  else
   fill(m_output.begin(), m_output.end(), 0);
  return state;
 }

public:
 ball_type():
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&ball_type::do_start));
  add_handler(STATE_UP_LEFT, handler_type(&ball_type::do_move));
  add_handler(STATE_UP_RIGHT, handler_type(&ball_type::do_move));
  add_handler(STATE_DOWN_LEFT, handler_type(&ball_type::do_move));
  add_handler(STATE_DOWN_RIGHT, handler_type(&ball_type::do_move));
 }
};
```

Автомат может находиться в пяти состояниях – в четырех состояниях соответственно каждому возможному направлению движения и в выключенном состоянии. В конструкторе класса ball_type каждому из этих состояний задается обработчик. В качестве обработчика выключенного состояния назначается функция do_start, которая выполняет инициализацию шара. Здесь мы опускаем множество проверок входных данных, которые, безусловно, должны быть в реальной программе.

Состояниям движения назначается обработчик do_move. В его задачи входит вычисление следующих координат шара на основе текущих координат, направления движения и наличия препятствий на пути следования. В начале функции do_move вычисляются планируемые изменения текущих координат на основе текущего направления. Далее в цикле по количеству возможных направлений выполняется проверка наличия препятствия в новой позиции. Если препятствие есть, направление движения меняется в соответствии с описанным выше, после чего происходит проверка наличия препятствия в новой планируемой позиции. Если возможное направление не найдено (к примеру, стол сузился в обоих направлениях или шар прижат в угол другим шаром), шар останавливается и переходит в выключенное состояние. В противном случае происходит изменение текущих координат и направления, а также запись выходных параметров.

Как уже не раз говорилось, приведенная схема для наглядности чрезвычайно упрощена и имеет вследствие этого немало недостатков. В частности, когда между двумя шарами на одной диагонали, двигающихся навстречу друг другу, находится одна пустая клетка, в соответствии с приведенным механизмом они «не заметят» друг друга, и оба встанут на эту клетку, после чего продолжат движение в исходных направлениях.

Безусловно, такое поведение является некорректным и требует внесения дополнительных проверок в код реальной программы. Разумеется, при этом потребуется передача от каждого шара помимо его координат еще и направления движения, что в данном случае было опущено для простоты. Однако здесь для начала потребуется ответить на вопрос, какое поведение обоих шаров являлось бы в такой ситуации корректным. К примеру, оба шара могут встать на пустую клетку, после чего поменять направление, либо же сразу поменять направление, не попадая в клетку и, тем самым, не осуществив столкновения. Оба варианта не самым лучшим образом отражают естественный ход событий, что является следствием упрощения модели и дискретности движения.

Автомат, отвечающий за поведение стола, вследствие своих более простых задач реализуется гораздо проще:

```
class table_type: public fsm_type
{
public:
 enum {
  INPUT_CYCLES,
  INPUT_X1,
  INPUT_Y1,
  INPUT_X2,
  INPUT_Y2,
  INPUT_NUMBER
```

```
 };
 enum {
  OUTPUT_BOUND_LEFT,
  OUTPUT_BOUND_RIGHT,
  OUTPUT_BOUND_TOP,
  OUTPUT_BOUND_BOTTOM,
  OUTPUT_NUMBER
 };
 enum {STATE_ON};

private:
 // размеры стола
 int m_hsize, m_vsize;
 // количество циклов движения
 data_type m_cycles;

 state_type do_out(state_type state)
 {
  // если состояние - включено, отобразим шары
  if (state == STATE_ON)
  {
   // какое-либо отображение стола и шаров
   // ...
   // количество выполненных циклов отображения
   m_cycles++;
  };
  // вычисление следующего состояния
  state = (m_cycles < m_input[INPUT_CYCLES]) ? STATE_ON : STATE_OFF;
  if (state != STATE_OFF)
  {
   // выводим данные о границах
   m_output[OUTPUT_BOUND_LEFT] = 0;
   m_output[OUTPUT_BOUND_RIGHT] = m_hsize + 1;
   m_output[OUTPUT_BOUND_TOP] = 0;
   m_output[OUTPUT_BOUND_BOTTOM] = m_vsize + 1;
  }
  else
   // обнуление выходов влечет остановку шаров
   fill(m_output.begin(), m_output.end(), 0);
  return state;
 }

public:
 table_type(int hsize, int vsize):
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER),
  m_hsize(hsize), m_vsize(vsize), m_cycles(0)
 {
  add_handler(STATE_OFF, handler_type(&table_type::do_out));
  add_handler(STATE_ON, handler_type(&table_type::do_out));
 }
};
```

Приведенный автомат может пребывать в двух состояниях –

включенном и выключенном. В конструкторе класса table_type обоим состояниям назначается обработчик do_out.

Внутри функции do_out выполняется отображение шаров и наращивание значения счетчика циклов, после чего на основе этого значения автомат может перейти в выключенное состояние или остаться во включенном. В конце функции выполняется вывод границ стола. В случае выключения все границы обнуляются, в связи с чем на следующем такте завершат работу шары.

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_CYCLES,
  INPUT_INIT_X1,
  INPUT_INIT_Y1,
  INPUT_INIT_DIR1,
  INPUT_INIT_X2,
  INPUT_INIT_Y2,
  INPUT_INIT_DIR2,
  INPUT_NUMBER
 };
 enum {OUTPUT_NUMBER};

 class factory_type: public factory_abstract_type
 {
  int m_hsize, m_vsize;
 public:
  factory_type(int hsize, int vsize):
   m_hsize(hsize), m_vsize(vsize)
  {}
  int number_fsm(void) const { return FSM_NUMBER; }
  int number_input(void) const { return INPUT_NUMBER; }
  int number_output(void) const { return OUTPUT_NUMBER; }
  linkstore_type get_links(void) const
  {
   linkstore_type store;
   // вход - количество циклов
   store.add(
    PSEUDOFSM_NETINPUT, INPUT_CYCLES,
    FSM_TABLE, table_type::INPUT_CYCLES);
   // входные координаты и направление первого шара
   for (int i = 0; i <= INPUT_INIT_DIR1 - INPUT_INIT_X1; i++)
   {
```

```
   store.add(
    PSEUDOFSM_NETINPUT, INPUT_INIT_X1 + i,
    FSM_BALL1, ball_type::INPUT_INIT_X + i);
  };
  // входные координаты и направление второго шара
  for (int i = 0; i <= INPUT_INIT_DIR2 - INPUT_INIT_X2; i++)
  {
   store.add(
    PSEUDOFSM_NETINPUT, INPUT_INIT_X2 + i,
    FSM_BALL2, ball_type::INPUT_INIT_X + i);
  };
  // выходы стола - его границы
  for (int i = 0; i < table_type::OUTPUT_NUMBER; i++)
  {
   store.add(
    FSM_TABLE, i,
    FSM_BALL1, ball_type::INPUT_BOUND_LEFT + i);
   store.add(
    FSM_TABLE, i,
    FSM_BALL2, ball_type::INPUT_BOUND_LEFT + i);
  };
  // выходы шаров - на вход стола и друг другу
  for (int i = 0; i < ball_type::OUTPUT_NUMBER; i++)
  {
   // входы стола - координаты и направления шаров
   store.add(
    FSM_BALL1, i,
    FSM_TABLE, table_type::INPUT_X1 + i);
   store.add(
    FSM_BALL2, i,
    FSM_TABLE, table_type::INPUT_X2 + i);
   // оба шара связаны между собой
   store.add(
    FSM_BALL1, i,
    FSM_BALL2, ball_type::INPUT_OTHER_X + i);
   store.add(
    FSM_BALL2, i,
    FSM_BALL1, ball_type::INPUT_OTHER_X + i);
  };
  return store;
 }
 fsm_abstract_type *create_fsm(int i)
 {
  fsm_abstract_type *pfsm;
  if (i == FSM_TABLE)
   pfsm = new table_type(m_hsize, m_vsize);
  else
   pfsm = new ball_type();
  return pfsm;
 }
 void destroy_fsm(int i, fsm_abstract_type *pfsm)
 {
  if (i == FSM_TABLE)
```

```
    delete dynamic_cast<table_type *>(pfsm);
   else
    delete dynamic_cast<ball_type *>(pfsm);
  }
 };

public:
 billiard_type(factory_type &factory):
  fsmnet_type(factory)
 {}
};
```

Как говорилось выше, входами сети являются количество циклов выполнения и инициализационные данные для каждого шара. Функция get_links класса фабрики сети задает связи между автоматами, входами и выходами сети в соответствии с рис. 21.

Выполнение цикла работы такой сети может осуществлять следующий фрагмент кода:

```
billiard_type::factory_type factory(5, 6);
billiard_type fsmnet(factory);

vector<fsm_type::data_type> vin(billiard_type::INPUT_NUMBER);
vin[billiard_type::INPUT_CYCLES] = 20;
vin[billiard_type::INPUT_INIT_X1] = 1;
vin[billiard_type::INPUT_INIT_Y1] = 1;
vin[billiard_type::INPUT_INIT_DIR1] = ball_type::STATE_DOWN_RIGHT;
vin[billiard_type::INPUT_INIT_X2] = 4;
vin[billiard_type::INPUT_INIT_Y2] = 4;
vin[billiard_type::INPUT_INIT_DIR2] = ball_type::STATE_UP_LEFT;

fsmnet.put_input(vin);
do
 fsmnet.do_work();
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 работы сети. Автоматная сеть устроена так, что входной вектор может не меняться на протяжении работы сети, выходных же значений нет вообще, вследствие чего чтение и запись входных и выходных значений внутри цикла выполнения не производится.

3.5 Сумматор

Теперь рассмотрим другой пример, в отличие от предыдущего, мало привязанный к реальности, но в некоторой степени также подходящий 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. Результаты снова разбиваются попарно и суммируются, и т.д. Для любого n сумма всей последовательности будет получена за количество шагов, равное $\lceil \log_2 n \rceil$, если алгоритм работает 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 вычислителей.

Реализуем автоматную сеть в виде, представленном на рис. 22. Сеть содержит N автоматов, выполняющих непосредственно вычисление суммы двух чисел, и один автомат, управляющий вычислением суммы всей последовательности.

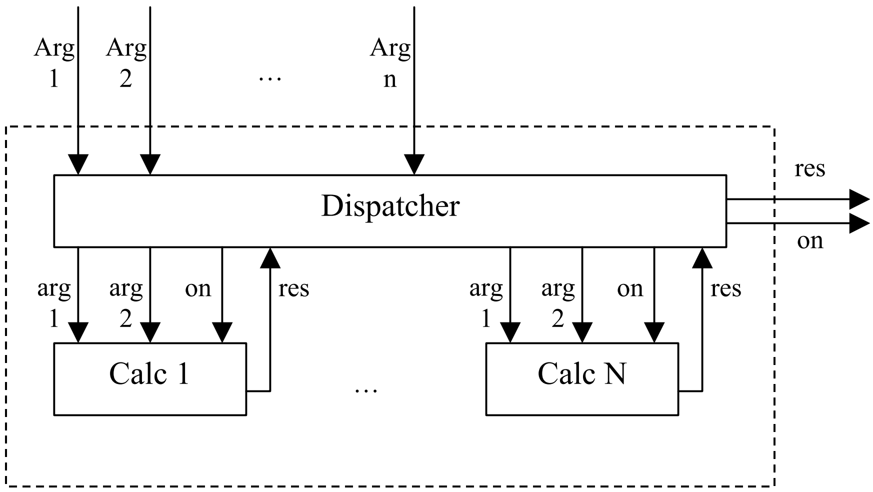

Рис. 22

Каждый вычисляющий автомат (далее – вычислитель) получает на входе пару чисел и управляющий сигнал, на выходе выдает результат суммирования. Управляющий автомат (далее – контроллер) в качестве входных сигналов получает всю последовательность суммируемых чисел, а также результаты суммирования с вычислителей. В качестве выхода подает на каждый вычислитель очередную пару значений и управляющий сигнал, а также 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.

Вместе с парой чисел на каждый вычислитель подается управляющий сигнал «вкл». Когда длина очереди становится меньше удвоенного количества вычислителей, те вычислители, на которые не хватило работы, получают управляющий сигнал «выкл».

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 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.

Каждый вычислитель пребывает либо в выключенном состоянии, либо в состоянии выдачи результата. В последнее состояние вычислитель переходит при получении управляющего сигнала «вкл» после выполнения суммирования двух переданных на вход аргументов.

Состояние контроллера характеризует количество работающих в текущий момент вычислителей, т.е. количество распределенных в последний раз пар чисел. В начале работы обычное состояние контроллера – «N». Когда процесс близится к завершению вычислений, состояние меняется на меньшее «N», в соответствии с текущим количеством распределенных пар. По достижении состояния «0», если в очереди контроллера остался лишь один элемент, состояние меняется на «Вывод результата». На выходе при этом единственный элемент очереди и сигнал о выводе результата. На следующем такте выключается контроллер и вся сеть.

Класс вычислителя calculator_type определяет один обработчик на оба состояния:

```
class calculator_type: public fsm_type
{
public:
 enum {INPUT_ARG1, INPUT_ARG2, INPUT_ON, INPUT_NUMBER};
 enum {OUTPUT_RESULT, OUTPUT_NUMBER};
 enum {STATE_OUT};

private:
 state_type do_calc(state_type state)
 {
  if (m_input[INPUT_ON])
  {
   state = STATE_OUT;
   m_output[OUTPUT_RESULT]= m_input[INPUT_ARG1]+m_input[INPUT_ARG2];
  }
  else
  {
   state = STATE_OFF;
   m_output[OUTPUT_RESULT] = 0;
  };
  return state;
 }
public:
 calculator_type():
  fsm_type(INPUT_NUMBER, OUTPUT_NUMBER)
 {
  add_handler(STATE_OFF, handler_type(&calculator_type::do_calc));
  add_handler(STATE_OUT, handler_type(&calculator_type::do_calc));
 }
};
```

В случае наличия входного сигнала включения, вычислитель производит суммирование и вывод результата. В противном случае переходит в выключенное состояние.

Класс контроллера control_type определяет три обработчика состояний:

```
class control_type: public fsm_type
{
public:
 enum {OUTPUT_RESULT, OUTPUT_ON, OUTPUT_ARG1_0, OUTPUT_ARG2_0,
OUTPUT_CALC_ON_0};
 enum {STATE_OUT, STATE_CALC_NUMBER_0};
```

```
private:
 deque<data_type> m_queue;
 int m_argnum; // количество аргументов сумматора
 int m_calcnum; // количество вычислителей

 // собрать со входов все ненулевые результаты вычислений
 void collect(void)
 {
  for (int i = 0; i < m_calcnum; i++)
  {
   data_type result = m_input[m_argnum + i];
   if (result)
    m_queue.push_back(result);
  };
 }

 // распределить пары данных, сколько есть в наличии, на вычислители
 int dispatch(void)
 {
  int num =
   (m_queue.size() >= (m_calcnum << 1)) ?
    m_calcnum : (m_queue.size() >> 1);
  for (int i = 0; i < num; i++)
  {
   m_output[OUTPUT_ARG1_0 + calculator_type::INPUT_NUMBER * i] =
    m_queue.front();
   m_queue.pop_front();
   m_output[OUTPUT_ARG2_0 + calculator_type::INPUT_NUMBER * i] =
    m_queue.front();
   m_queue.pop_front();
   m_output[OUTPUT_CALC_ON_0 + calculator_type::INPUT_NUMBER * i]=1;
  };
  for (int i = num; i < m_calcnum; i++)
  {
   m_output[OUTPUT_ARG1_0 + calculator_type::INPUT_NUMBER * i] = 0;
   m_output[OUTPUT_ARG2_0 + calculator_type::INPUT_NUMBER * i] = 0;
   m_output[OUTPUT_CALC_ON_0 + calculator_type::INPUT_NUMBER * i]=0;
  };
  return num;
 }

 state_type do_start(state_type state)
 {
  // проверим наличие сигналов на входе (поищем ненулевой)
  bool noinput = (m_input.end() ==
   find_if(m_input.begin(), m_input.end(),
    not1(logical_not<data_type>())));
  // если есть, переключимся на обработку
  if (!noinput)
  {
   // сложим все входные данные в очередь
   m_queue.assign(m_input.begin(), m_input.begin() + m_argnum);
   state = STATE_CALC_NUMBER_0 + dispatch();
```

```
   m_output[OUTPUT_RESULT] = 0;
   m_output[OUTPUT_ON] = 0;
  }
  else
  {
   // иначе продолжим быть выключенными
   state = STATE_OFF;
   fill(m_output.begin(), m_output.end(), 0);
  };
  return state;
 }

 state_type do_calc(state_type state)
 {
  // сложить все ненулевые предыдущие результаты в конец очереди
  collect();
  // 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_output[OUTPUT_ON] = 1;
  }
  else
  {
   // распределить пары из начала очереди
   state = STATE_CALC_NUMBER_0 + dispatch();
   m_output[OUTPUT_RESULT] = 0;
   m_output[OUTPUT_ON] = 0;
  };
  return state;
 }

 state_type do_off(state_type state)
 {
  // перейдем в выключенное состояние
  state = STATE_OFF;
  fill(m_output.begin(), m_output.end(), 0);
  return state;
 }

public:
 control_type(int argnum, int calcnum):
  fsm_type(argnum + calcnum,
   calcnum * calculator_type::INPUT_NUMBER + 2),
  m_argnum(argnum), m_calcnum(calcnum)
 {
  add_handler(STATE_OFF, handler_type(&control_type::do_start));

  for (int i = 0; i <= calcnum; i++)
   add_handler(
    STATE_CALC_NUMBER_0 + i, handler_type(&control_type::do_calc));
```

```
  add_handler(STATE_OUT, handler_type(&control_type::do_off));
 }
};
```

Обработчик выключенного состояния do_start проверяет наличие на входе контроллера последовательности ненулевых чисел, после чего, в случае наличия, помещает их 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» определен обработчик do_calc. В начале функции do_calc осуществляется сбор результатов вычислений предыдущего такта со всех вычислителей с помощью функции collect. Далее выполняется проверка факта завершения вычислений, который определяется наличием в очереди лишь одного элемента и отсутствием активных вычислителей. В случае если хотя бы одно из этих условий не выполняется, производится попытка следующего распределения. Если же вычисления закончены, происходит переход к состоянию «Вывод результата». Его обработчик do_off 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 связи между ними в соответствии с рис. 22:

```
class summator_type: public fsmnet_type
{
public:

 enum {FSM_CONTROL, FSM_CALCULATOR_0};
 enum {OUTPUT_RESULT, OUTPUT_ON, 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; }

linkstore_type get_links(void) const
{
 linkstore_type store;

 // входы сети прицепим к контроллеру
 for (int i = 0; i < m_inputnum; i++)
  store.add(PSEUDOFSM_NETINPUT, i, FSM_CONTROL, i);

 // выходы сети - первые два выхода от контроллера
 for (int i = 0; i < m_outputnum; i++)
  store.add(FSM_CONTROL, i, PSEUDOFSM_NETOUTPUT, i);

 // все вычислители к контроллеру подцепим
 for (int i = 0; i < m_fsmnum - FSM_CALCULATOR_0; i++)
 {
  // аргументы на вычислители с контроллера (после выходов сети)
  for (int j = 0; j < calculator_type::INPUT_NUMBER; j++)
   store.add(
    FSM_CONTROL,
    m_outputnum + calculator_type::INPUT_NUMBER * i + j,
    FSM_CALCULATOR_0 + i, j);

  // выход вычислителя - к контроллеру, после входов сети
  store.add(
   FSM_CALCULATOR_0 + i, calculator_type::OUTPUT_RESULT,
   FSM_CONTROL, m_inputnum + i);
 };
 return store;
}

fsm_abstract_type *create_fsm(int i)
{
 fsm_abstract_type *pfsm;
 if (i == FSM_CONTROL)
  pfsm = new control_type(m_inputnum, m_fsmnum - 1);
 else
  pfsm = new calculator_type();
 return pfsm;
}

void destroy_fsm(int i, fsm_abstract_type *pfsm)
{
 if (i == FSM_CONTROL)
  delete dynamic_cast<control_type *>(pfsm);
 else
```

```
    delete dynamic_cast<calculator_type *>(pfsm);
   }
  };

public:

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

Такой набор классов может быть использован следующим кодом:

```
// инициализация вектора входных значений сети
vector<fsm_type::data_type> vin;
// ...

// количество автоматов - контроллер+вычислители
int fsanum = 1 + DEF_CALCNUM;

// фабрика сети и сеть
summator_type::factory_type
 factory(vin.size(), summator_type::OUTPUT_NUMBER, fsanum);
summator_type fsmnet(factory);

// передать входные данные
fsmnet.put_input(vin);

// вычисленное значение суммы
fsm_type::data_type sum = 0;
do
{
 // выполнить такт работы сети
 fsmnet.do_work();

 // получить очередные выходные данные
 vector<fsm_type::data_type> vout = fsmnet.get_output();

 // если результат есть, сохраним его
 if (vout[summator_type::OUTPUT_ON])
  sum = vout[summator_type::OUTPUT_RESULT];

} while (!fsmnet.is_off());
```

Количество входов сети определяется на основе длины вектора входных значений. После передачи входной последовательности в сеть выполняется полный цикл работы сети. В течение цикла осуществляется проверка наличия выходного результата и его сохранение.

Глава 4. Сети Петри

Текущая глава посвящена вопросу создания параллельных программ на основе сетей Петри. Сети Петри представляют собой аппарат моделирования динамических дискретных систем и являются одним из наиболее адекватных способов описания асинхронного выполнения параллельных процессов, в том числе в распределенных системах.

Следует отметить, что, поскольку сети Петри изначально предназначены больше для моделирования систем, представление на их базе архитектуры проектируемой программной системы, в том числе параллельной, а также последующая ее реализация, для некоторых разработчиков оказываются сопряженными с некоторыми сложностями. Это является 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 сильно углубляться в описание теории сетей Петри, поскольку это довольно широкая тема, а ограничимся лишь поверхностным описанием, достаточным для иллюстрации их использования при построении параллельных программ. Для более детального ознакомления следует обратиться к специально посвященной этой теме литературе [11, 15, 21].

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 роде сеть Петри близка по смыслу к диаграмме состояний автомата, который может находиться одновременно в нескольких состояниях. Каждый переход обуславливается каким-либо подмножеством текущих состояний и заменяет его другим подмножеством состояний. Позиции и переходы сети Петри в такой интерпретации играют роль состояний автомата и действий на переходах соответственно.

Графически сеть Петри представляется в виде двудольного ориентированного графа с двумя типами вершин – позициями и переходами (рис. 23). 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. Наличие фишек в позиции на графе обозначается числом либо жирными точками в соответствующем количестве. Совокупность всех фишек, размещенных в позициях сети Петри, называется разметкой сети.

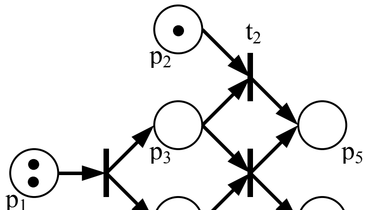

Рис. 23

Дуги сети Петри могут быть направлены лишь от позиций к переходам либо от переходов к позициям. Дуги могут быть кратными, т.е. иметь вес – также целое неотрицательное число. Наличие кратной дуги между позицией и переходом эквивалентно наличию между ними соответствующего количества простых дуг в том же направлении. 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 следующими параметрами:

- 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).

Функционирование сети Петри осуществляется в виде последовательности срабатывания переходов. В каждый момент времени в сети есть некоторое количество разрешенных переходов, т.е. таких, которые могут сработать. Переход считается разрешенным, если во всех его входных позициях количество фишек не меньше, чем кратность соответствующих входных дуг. 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, выбора пользователем и т.п. При срабатывании простого перехода из каждой его входной позиции изымается количество фишек, равное кратности соответствующей входной дуги, после чего к каждой выходной позиции добавляются фишки в соответствии с кратностью выходных дуг. После каждого срабатывания перехода 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 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 ней может запретить другие переходы. Такой ситуацией моделируются конфликты, когда для выполнения нескольких операций требуется использование общего ресурса. Пример такой ситуации виден на рис. 23. В самом начале работы сети разрешенным является один переход – t_1. После его первого срабатывания разрешенными являются все три перехода сети. Если же теперь сработает, к примеру, переход t_2, он запретит переход t_3, и наоборот, срабатывание t_3 запретит переход t_2, поскольку переходы t_2 и t_3 имеют общую входную позицию p_3.

Срабатывание простого перехода считается мгновенным. Поскольку вероятность одновременного происхождения двух мгновенных событий равна нулю, два простых перехода не могут сработать одновременно [21]. Отсюда вытекает довольно парадоксальное свойство сетей Петри: притом, что они моделируют асинхронное выполнение параллельных процессов, сама по себе работа сети Петри происходит строго последовательно. Именно невозможностью одновременного срабатывания каких-либо переходов определяется асинхронная природа сетей Петри. Параллелизм же выполнения выражается в том, что в один момент времени разные участки сети могут отражать выполнение разных независимых процессов.

Мы не будем касаться вопросов анализа и верификации сетей Петри [11, 21], считая, что этот этап уже пройден. Помимо приведенного классического описания сетей Петри, существуют различные расширенные модели. 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 разрешенных переходов группой переходов с наивысшим приоритетом. Существуют и другие расширения, из которых мы рассмотрим строго иерархические сети [11].

4.1.2 Строго иерархические сети

Ранее уже звучало, что срабатывание простого перехода сети Петри мгновенно. Однако зачастую переходами моделируется выполнение каких-либо далеко не мгновенных операций. В этом случае выполнению операции может быть сопоставлен длительный переход. Для наглядности длительные переходы, в отличие от простых, часто обозначаются прямоугольником (рис. 24, слева).

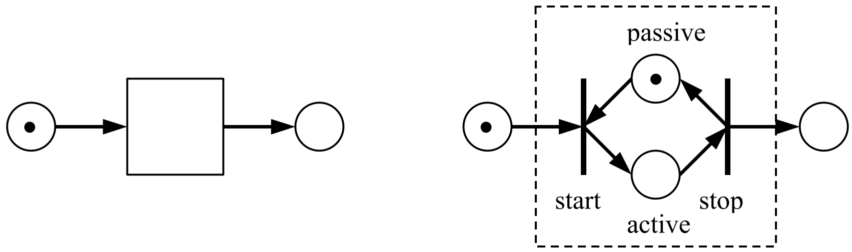

Рис. 24

Длительный переход, как и простой, начинает выполнение с изъятия фишек из входных позиций и завершает, соответственно, помещая фишки в выходные. Основное отличие от простого перехода заключается в том, что между этими двумя моментами могут срабатывать другие переходы. Иначе говоря, срабатывание длительного перехода не атомарно.

С момента начала выполнения до момента завершения длительный переход считается активным, в противном случае – пассивным. Длительный переход не может сработать, если он активен, т.е. уже выполняется, даже если он разрешен по наличию фишек во входных позициях.

Выполнение длительного перехода эквивалентно выполнению фрагмента сети, изображенного на рис. 24, 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.

Поскольку длительный переход не мгновенен и его срабатывание не атомарно, выполнение различных длительных переходов может перекрываться во времени, т.е. осуществляться параллельно.

Операция, выполняемая в течение длительного перехода, может содержать полный жизненный цикл какой-либо вложенной сети (рис. 25). На основе такого включения могут быть организованы иерархические сети произвольной вложенности. Если сеть не содержит дуг, соединяющих позиции и переходы разных уровней вложенности либо разных сетей одного уровня, то такая сеть называется строго иерархической [11]. В дальнейшем мы будем говорить лишь о строго иерархических сетях. Длительный переход, содержащий вложенную сеть, будем называть составным.

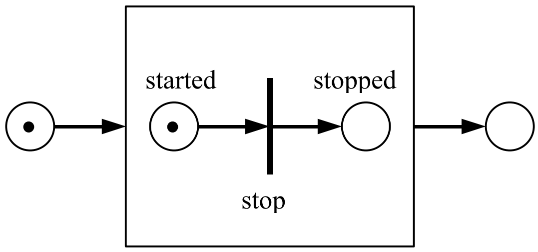

Рис. 25

В момент активации составного перехода из его входных позиций извлекаются фишки, после чего вложенная сеть начинает функционирование исходя из своей начальной разметки. Когда после очередного срабатывания внутреннего перехода вложенной сети в ней не остается разрешенных переходов, она прекращает работу. В этот момент соответствующий составной переход помещает фишки в выходные позиции и переходит в пассивное состояние. При следующей активации составного перехода вложенная сеть снова размечается в соответствии со своей начальной разметкой и начинает новый жизненный цикл.

Вследствие отсутствия связей внутренних позиций и переходов вложенной сети с внешними по отношению к ней, внутренняя структура вложенной сети скрыта от внешней и может быть реализована полностью независимо. С точки зрения внешней сети любой ее составной переход произвольной сложности, так же как и вообще любой длительный переход, могут быть представлены в виде составного перехода с простой вложенной сетью, изображенного на рис. 25. Здесь единственный простой переход характеризует завершение работы длительного перехода.

Именно этот факт позволяет нам довольно просто использовать сети Петри в параллельном программировании – 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. 26 изображена одна из довольно популярных на сегодняшний день конструкций 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.

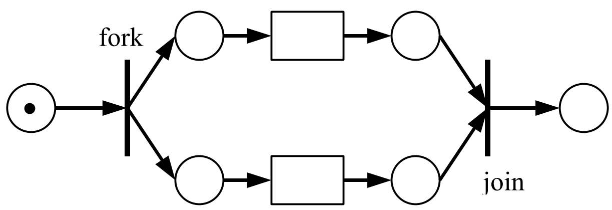

Рис. 26

Используемые сегодня объекты синхронизации также легко моделируются сетями Петри. К примеру, на рис. 27 изображен фрагмент сети Петри, моделирующий барьерную синхронизацию трех процессов.

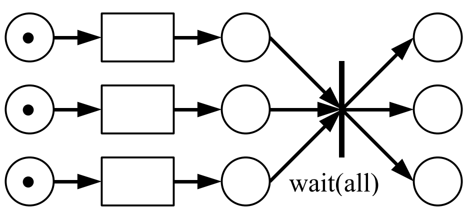

Рис. 27

В начале работы сети разрешенными являются три длительных перехода, характеризующих выполнение некоторых подзадач. Три независимых процесса выполняют эти подзадачи, и лишь после их полного завершения становится разрешенным переход-барьер wait(all). После его срабатывания все три процесса снова продолжают свою работу.

Похожим образом может быть смоделировано ожидание завершения любой из подзадач, первой завершившей свое выполнение. К примеру, на рис. 28 показан такой фрагмент сети. При завершении выполнения любого длительного перехода становится разрешенным переход wait(any). После его срабатывания дальнейшая работа продолжается, в то время как остальные подзадачи завершают 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.

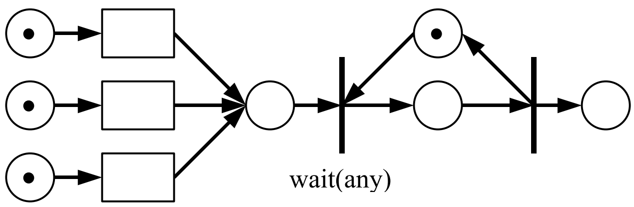

Рис. 28

Другой пример синхронизации параллельных процессов – критическая секция. Фрагмент сети на рис. 29 обеспечивает взаимоисключающий доступ к некоторому общему ресурсу для двух процессов.

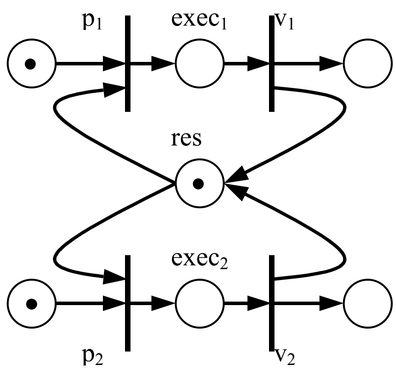

Рис. 29

Первый же процесс, захвативший доступ к критической секции, лишит фишки соответствующую позицию 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, позволяющего одновременный доступ к ресурсу нескольких процессов в количестве не более заданного.

Важная особенность сетей Петри заключается в атомарности срабатывания перехода и, как следствие, возможности атомарного захвата одновременно нескольких ресурсов, что исключает 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 с помощью критических секций. Вариант такой реализации, описанный в виде фрагмента сети Петри, изображен на рис. 30.

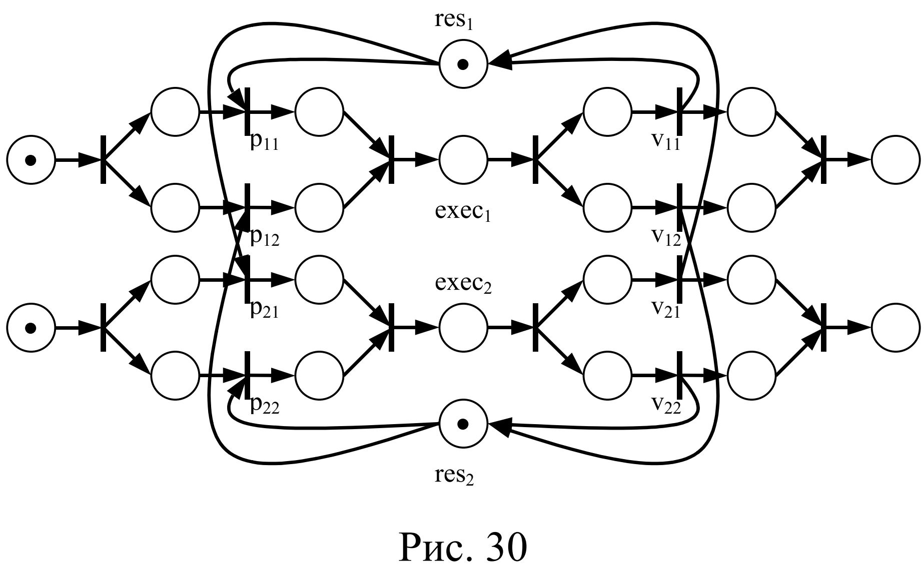

Рис. 30

Для выполнения некоторой операции каждому из двух процессов требуются два ресурса, доступность которых характеризуют фишки в позициях res_1 и res_2. Каждый процесс пытается захватить оба ресурса в произвольной последовательности. Если переходы p_{11} и p_{12} в произвольной последовательности сработают раньше, чем переходы p_{21} и p_{22}, либо наоборот, оба процесса благополучно закончат свое выполнение. Если же первыми сработают переходы p_{11} и p_{22}, либо p_{12} и p_{21}, каждый процесс окажется навсегда заблокированным в ожидании другого.

Разумеется, в данном случае 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. Однако такой подход существенно усложняется при усложнении задачи в целом. В то же время на рис. 31 мы видим фрагмент сети Петри, успешно решающий поставленную задачу без введения порядка захвата – захват ресурсов осуществляется одновременно в момент срабатывания переходов pp_1 или pp_2.

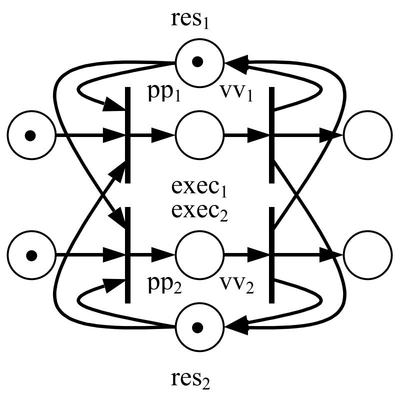

Рис. 31

В последнем примере видны преимущества сетей Петри перед популярными сегодня в программировании средствами синхронизации. С помощью сетей Петри оказывается гораздо проще организовать сложное взаимодействие различных участков системы, особенно в ситуациях, когда в разрабатываемой системе происходит широкое использование 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 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 размышляют, поскольку для еды им не хватает вилок.

Основная проблема, иллюстрируемая этой задачей – проблема возможности взаимоблокировки, которая была описана раньше. 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 ожидании ее освобождения.

Как уже говорилось выше, проблема взаимоблокировки легко решается сетями Петри, поскольку они предоставляют возможность атомарного захвата одновременно нескольких ресурсов.

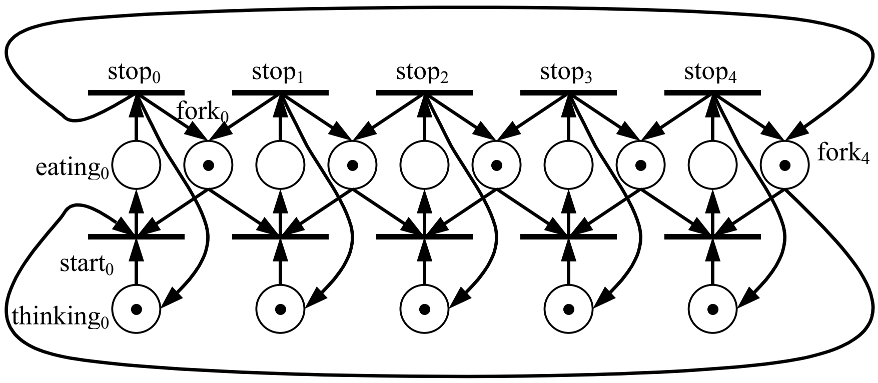

Рис. 32

На рис. 32 изображена сеть Петри, решающая задачу об обедающих философах [21]. Позиции eating и thinking здесь характеризуют пребывание соответствующего философа в состоянии еды или размышлений. Позициями fork обозначается доступность вилок. Переходы start и stop характеризуют переход философа к еде и к размышлениям соответственно.

Однократный процесс приема пищи каждым философом представляет собой длительную операцию и может быть представлен в виде длительного составного перехода, изображенного на рис. 25. В этом случае срабатывание перехода stop характеризует завершение приема пищи философом и, соответственно, завершение длительного перехода. На рис. 33 изображена сеть с использованием таких составных переходов, содержимое которых здесь опущено для компактности.

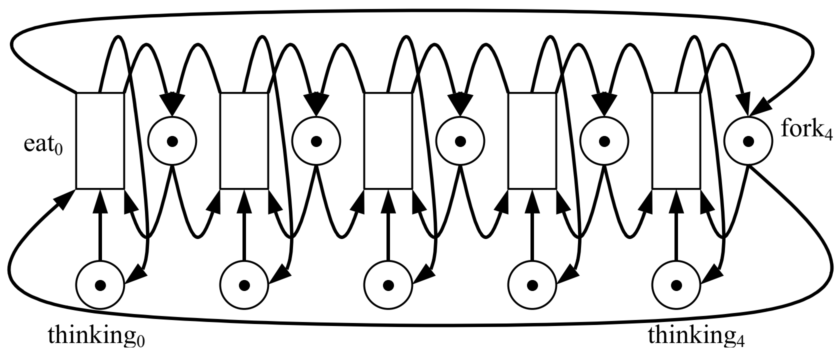

Рис. 33

Помимо проблемы взаимоблокировки, задача об обедающих философах демонстрирует возможность голодания (заговора соседей), т.е. такую ситуацию, в которой один или более философов никогда не получат доступ к еде. Проблема голодания легко решается не только сетями Петри, поэтому их применение здесь не столь иллюстративно, как решение проблемы взаимоблокировки, однако вскользь мы рассмотрим и ее.

Одним из довольно простых решений этой проблемы, хотя, конечно, далеко не самым лучшим, является следующее. К сети, изображенной на рис. 33, добавим барьерную синхронизацию после того, как каждый философ осуществит прием пищи по одному разу. Для этого добавим к каждому философу по две позиции – позицию todo, количество фишек в которой отражает количество предстоящих подходов философа к еде до следующего барьера, и позицию done, отражающую количество выполненных подходов. Полученная сеть изображена на рис. 34.

В случае если синхронизация после каждого подхода к еде является слишком частой мерой, количество фишек в каждой позиции todo может быть увеличено. Более того, начальные количества фишек в этих позициях могут быть заданы разными, в зависимости от потребности в интенсивном питании каждого из философов. В соответствии с начальным количеством фишек в позициях todo должна быть изменена и кратность дуг, ведущих в переход-барьер и из него.

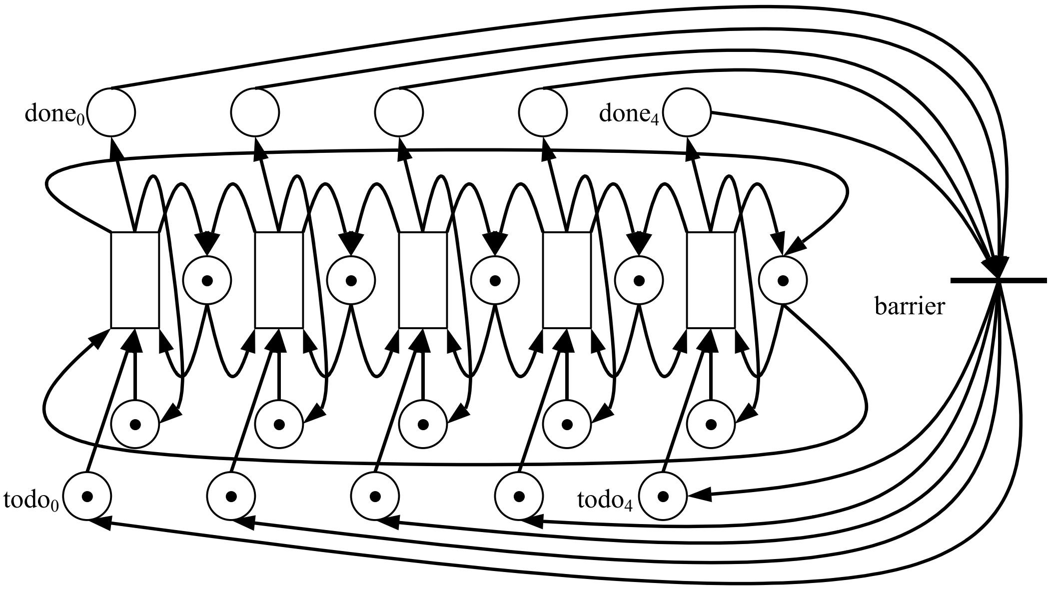

Рис. 34

С учетом такой модификации сети каждый философ будет получать доступ к еде в обычном порядке, пока не закончатся фишки в соответствующей позиции todo, после чего остановится на размышлениях. Таким образом, пока каждый из пяти философов не получит доступ к еде заданное количество раз, переход к следующему циклу осуществлен не будет. Когда все фишки из позиций todo постепенно переместятся в соответствующие позиции done, переход-барьер снова переместит их в позиции todo, после чего начнется следующий цикл приема пищи.

4.2 Пример реализации механизма сетей Петри

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 при разработке параллельной программы 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 реализации 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. Таким образом, работа сетей, в том числе иерархических, будет нами реализована в виде последовательной программы. При этом, однако, на ее основе могут быть построены параллельные программы, процесс выполнения которых отражается выполнением соответствующих сетей.

В приложении 4 приведен используемый в качестве примера исходный код классов, реализующих работу иерархических сетей. Среди них объявляется класс позиции place_type, а также абстрактный класс перехода transition_abstract_type. На основе последнего строятся класс простого перехода transition_simple_type и класс составного перехода transition_composite_type. Абстрактным классом перехода описывается следующий предоставляемый каждым переходом интерфейс:

```
class transition_abstract_type
{
public:
 typedef std::vector<transition_abstract_type *> enabledlist_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;
 // 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. Наконец, функция fire выполняет срабатывание конкретного перехода из списка, полученного с помощью функции get_enabled. На вход ей передается номер позиции перехода в этом списке. Функции get_enabled 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 переходе в активное и пассивное состояние. Эти обработчики при желании могут быть переопределены в унаследованных классах.

Класс простого перехода transition_simple_type предоставляет реализацию перечисленных функций с учетом специфики простого перехода. Простой переход активен лишь мгновение, т.е. при проверке состояния всегда пассивен. Помимо этого, простой переход не содержит вложенной сети, поэтому возвращает пустой список внутренних разрешенных переходов, функция же срабатывания внутреннего перехода бессмысленна.

Класс составного перехода transition_composite_type реализует работу одноуровневой сети Петри. 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 с этой целью введен класс content_type, чтение и преобразование структур которого производится в момент создания объекта составного перехода.

Класс 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 входных и выходных дугах, а также о начальной разметке. Класс предоставляет вызывающему коду функции добавления позиции, перехода, входной и выходной дуг, а также помещения фишки в позицию. Помимо этого, для класса составного перехода предоставляются функции получения в упорядоченной форме списка позиций, списка переходов, матриц входных и выходных позиций и вектора начальной разметки. Все эти данные передаются в соответствующие структуры класса составного перехода в момент создания соответствующего объекта.

В процессе выполнения составной переход постоянно содержит текущий полный список внутренних разрешенных переходов m_enabled_full. Для поддержания актуальности этого списка используются две функции refresh_enabled, о которых подробнее поговорим позже. Помимо этого, в объекте содержится вектор текущей разметки m_marking, набор флагов разрешения либо активности переходов текущего уровня (без учета вложенности), а также две структуры данных, используемых для локализации срабатывающего перехода. Функция activate размечает переход в соответствии с начальной разметкой и обновляет текущее состояние списка разрешенных переходов. Если этот список не пуст, при вызове функции is_active возвращается значение истины. Построенный список может быть получен с помощью функции get_enabled. Наконец, функция fire локализует переход, номер которого передан ей в качестве параметра, и осуществляет действия по срабатыванию. Подробнее об этом мы поговорим после обсуждения функций обновления списка разрешенных переходов.

Определение, разрешен ли какой-либо переход, выполняется на основе матрицы входных дуг и текущей разметки. При большом количестве позиций эта операция может быть довольно тяжеловесной, поэтому следует избегать частого вычисления разрешения для всех переходов. По этой причине мы имеем две функции обновления текущего списка разрешенных переходов. Одна из них выполняет полное обновление списка разрешенных переходов и вызывается из функции активации составного перехода activate, другая – частичное обновление, при котором анализируются лишь переходы, попадающие в переданное в качестве параметра множество. Последняя функция вызывается после каждого срабатывания внутреннего перехода из функции fire.

В задачу функции полного обновления входит построение множества номеров переходов, включающего все переходы текущего уровня (далее – локальные переходы), после чего она вызывает функцию частичного обновления с этим множеством в качестве параметра.

Функция частичного обновления первым делом вычисляет флаги разрешения или активности локальных переходов с переданными номерами. Результаты попадают в набор флагов m_enabled_or_active, в котором после предыдущих обновлений уже содержатся значения относительно локальных переходов, не попавших в переданное множество. Значения в этом списке флагов говорят о том, требует ли соответствующий переход внимания при построении полного списка разрешенных переходов m_enabled_full.

Как было сказано ранее, если переход активен, он не может сработать, даже если он разрешен. Поэтому, при построении полного списка нас интересует либо сам переход (если он разрешен и пассивен), либо его внутренние разрешенные переходы (если он активен). Переход может быть активен, только если он составной, а значит, содержит непустой список внутренних разрешенных переходов. Полный список строится путем последовательного соединения локальных разрешенных переходов и внутренних списков локальных активных переходов. В процессе построения полного списка 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 списка m_enabled_full. Обе эти структуры нужны для локализации сработавшего перехода по его позиции в полном списке.

В самом начале функции fire осуществляется локализация перехода, позиция которого передана в качестве параметра, а именно определение локального перехода, к области которого относится этот переход, и позиции в его полном списке внутренних разрешенных переходов. Последний параметр имеет смысл, если соответствующий локальный переход окажется активным. Если же он пассивен, значит, он просто разрешен и представлен в списке лишь одним элементом.

После определения локального перехода функция запоминает текущую разметку для анализа ее изменений в дальнейшем. Затем осуществляются основные действия по срабатыванию. Если переход активен, то вычисленная позиция в его внутреннем списке передается его функции срабатывания. Если же пассивен, осуществляются действия по изъятию фишек из его входных позиций и активации перехода. В этот момент пассивный переход может стать активным или 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 помещаются фишки в выходные позиции.

Наконец, в конце функции срабатывания осуществляется частичное обновление текущего списка разрешенных переходов. Для этого строится множество номеров подлежащих рассмотрению локальных переходов. В это множество попадает сработавший локальный переход, а также все переходы, разметка входных позиций которых изменилась в течение текущего срабатывания. Изменения определяются на основе предыдущей разметки, запомненной в начале функции.

При рассмотренном подходе приходится мириться с тем, что смена разметки сети при срабатывании перехода не является мгновенной операцией, вследствие чего само по себе программное выполнение большой сети может занимать значительное количество вычислительных ресурсов. 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. 25 и содержащего простую сеть, срабатывание единственного перехода которой говорит о завершении работы исходной вложенной сети. Такой подход в нашем случае позволяет возложить задачу выполнения чересчур сложных составных переходов в виде автономных сетей на другие параллельные ресурсы. Вопросы конкретной реализации параллельного выполнения будут рассмотрены позже.

Наконец, класс petrinet_type реализует работу всей иерархической сети. Поскольку вся сеть является частным случаем составного перехода, класс сети Петри наследуется от класса transition_composite_type. Класс определяет функцию выполнения жизненного цикла сети live, в которой осуществляется активация соответствующего составного перехода и последовательность срабатываний внутренних переходов до тех пор, пока переход не перестанет быть активен. Выбор срабатывающего перехода каждый раз осуществляется извне с помощью переданного объекта окружения, реализующего интерфейс environment_abstract_type. Этот интерфейс содержит одну функцию, которая принимает полный список разрешенных переходов сети, ожидает срабатывания любого из них и возвращает номер соответствующей позиции в переданном списке.

Классы, 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 срабатывания, на основе которой вычисляется вероятность срабатывания среди конкретного набора разрешенных переходов. Мы же приведем вариант реализации для случая одинаковой интенсивности срабатывания всех переходов:

```
class environment_random_type: public
petrinet_type::environment_abstract_type
{
public:
 // инициализация датчика псевдослучайных чисел
 static void srandom(void)
 {
  srand((unsigned) time(NULL));
```

```
 }
 // 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)
 {
  return random(enabled.size());
 }
};
```

С использованием описанных классов создание и выполнение сети Петри, изображенной на рис. 23, может выглядеть следующим образом:

```
environment_random_type::srandom();
environment_random_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, не объявленные ранее. Они унаследованы от класса простого перехода, переопределяют обработчик активации и используются для отслеживания последовательности срабатываний:

```
class transition_1_type: public transition_simple_type
{
 void on_activate(void)
 {
  cout << "transition 1 fires" << endl;
 }
};
// ...
```

Описанные классы предназначены в общем случае для построения и выполнения иерархических сетей, поэтому приведем пример построения сети, изображенной на рис. 25:

```
class transition_stop_type: public transition_simple_type
{
 void on_activate(void)
 {
  cout << "transition stop fires" << endl;
 }
};

// ...

environment_random_type env;

// создание и заполнение составного перехода
place_type started, stopped;
transition_stop_type stop;
transition_composite_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_composite_type composite(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(composite);
content.add_arc(begin, composite);
content.add_arc(composite, end);
content.add_token(begin);
petrinet_type petrinet(content);

// выполнение сети
petrinet.live(env);
```

Наконец, рассмотрим варианты решения задачи об обедающих философах. Прежде всего, рассмотрим вариант одноуровневой сети (рис. 32). Такая сеть потребует два типа переходов:

```
// переход "приступает к еде"
class transition_eating_start_type: public transition_simple_type
{
private:
 int m_num;
 void on_activate(void)
 {
  cout << "Философ " << m_num << " приступил к еде" << endl;
 }
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)
 {
  cout << "Философ " << m_num << " приступил к размышлениям" <<endl;
 }
public:
 transition_eating_stop_type(int num): m_num(num) {}
};
```

Оба типа принимают на вход параметр – порядковый номер философа, который будет использоваться для отображения последовательности срабатываний. Код же создания и выполнения соответствующей сети Петри может быть следующим:

```
const int N = 5;
// создаем элементы сети
vector<place_type> eating(N), thinking(N), fork(N);
vector<transition_eating_start_type> start;
vector<transition_eating_stop_type> stop;
for (int i = 0; i < N; i++)
{
```

```
 start.push_back(transition_eating_start_type(i));
 stop.push_back(transition_eating_stop_type(i));
};

// заполняем содержимое сети
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]);
 // добавляем в сеть переходы
 content.add_transition(start[i]);
 content.add_transition(stop[i]);
};
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], start[i]);
 content.add_arc(fork[(i + 1) % N], 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], thinking[i]);
 content.add_arc(stop[i], fork[i]);
 content.add_arc(stop[i], fork[(i + 1) % N]);
};
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);
```

Прежде всего, осуществляется создание элементов, указателями на которые будет впоследствии манипулировать объект класса content_type. После этого в содержимое вносятся все позиции и переходы, затем дуги и фишки. Наконец, создается и выполняется сеть. Следует учитывать, что такая сеть будет жива всегда, поэтому, фактически, в последней строке осуществляется вечный цикл.

Теперь рассмотрим построение иерархической сети, изображенной на рис. 33. В этом случае нам нужен другой тип перехода, который мы определим следующим образом:

```
// составной переход "философ ест"
class transition_eating_type: public transition_composite_type
{
private:
 int m_num;

 void on_activate(void)
 {
  cout << "Философ " << m_num << " приступил к еде" << endl;
 }
 void on_passivate(void)
 {
  cout << "Философ " << m_num << " приступил к размышлениям" <<endl;
 }

public:
 transition_eating_type(const content_type &content, int num):
  transition_composite_type(content),
  m_num(num)
 {}
};
```

Обработчики этого перехода снова используются для отслеживания последовательности срабатываний, однако на этот раз срабатывают начало и завершение составного перехода. Каждый составной переход содержит вложенную сеть в соответствии с рис. 25. Ниже следует код, реализующий создание и выполнение такой сети:

```
const int N = 5;
// создаем содержимое составных переходов
vector<place_type> started(N), stopped(N);
vector<transition_simple_type> stop(N);
// и сами составные переходы
vector<transition_eating_type> eating;
for (int i = 0; i < N; i++)
{
 transition_composite_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]);
 eating.push_back(transition_eating_type(cnt, i));
};

// заполняем содержимое сети
vector<place_type> thinking(N), fork(N);
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]);
 // добавляем в сеть переходы
 content.add_transition(eating[i]);
};
for (int i = 0; i < N; i++)
{
 // входные дуги перехода eating
 content.add_arc(thinking[i], eating[i]);
 content.add_arc(fork[i], eating[i]);
 content.add_arc(fork[(i + 1) % N], eating[i]);
 // выходные дуги перехода eating
 content.add_arc(eating[i], thinking[i]);
 content.add_arc(eating[i], fork[i]);
 content.add_arc(eating[i], fork[(i + 1) % N]);
};
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);
```

Первым делом создаются и заполняются составные переходы, после чего они используются при включении элементов в содержимое внешней сети. До текущего момента мы не затрагивали вопрос параллельного выполнения, и в данном случае длительные переходы работают параллельно лишь теоретически. Далее же мы рассмотрим возможности реального распараллеливания выполнения длительных переходов.

4.2.2 Выполнение параллельных процессов

Для реализации параллельных программ мы будем отождествлять длительные подзадачи с составными переходами, аналогичными изображенному на рис. 25. Предложенный ранее вариант окружения, в котором ожидание срабатывания основывалось на случайном выборе перехода, в данном случае уже не подходит по следующей причине. Каждый составной переход содержит один простой переход, срабатывающий не произвольно, а вследствие завершения выполнения соответствующей подзадачи. При этом выполнение длительной подзадачи вынесено в отдельный параллельный ресурс.

Интерфейсы OpenMP и MPI не обеспечивают необходимой гибкости для реализации сетей Петри, поэтому обратимся к низкоуровневым средствам распараллеливания. Для примера мы рассмотрим многопоточное распараллеливание, хотя с использованием сетевых коммуникаций может быть выполнено и распараллеливание в системах с распределенной памятью.

Выполним распараллеливание выполнения длительных переходов с помощью интерфейса Win32 API. Для этого создадим следующий класс окружения:

```
// класс окружения с поддержкой многопоточного выполнения
class environment_type: public
petrinet_type::environment_abstract_type
{
public:
 // интерфейс работы, выполняемой во время длительного перехода
 class longjob_abstract_type
 {
 public:
  // функция выполнения работы
  virtual void run(void) = 0;
 };

 // длительный переход, характеризующий выполнение работы
 class transition_long_type: public transition_composite_type
 {
 private:
  // окружение
  environment_type *m_penv;
  // номер перехода в списке длительных переходов окружения
  int m_index;

  // вызовы инициализации и финализации выполнения работы окружением
  void on_activate(void)
  {
   m_penv->initialize_longjob(m_index);
  }
  void on_passivate(void)
  {
   m_penv->finalize_longjob(m_index);
  }
 public:
  transition_long_type(
   const content_type &content, environment_type *penv, int index):
   transition_composite_type(content), m_penv(penv), m_index(index)
  {}
 };

private:
```

```
 // структура данных, связанная с каждым длительным переходом
 struct longjob_data_type
 {
  // содержимое соответствующего длительного перехода
  place_type started, stopped;
  transition_simple_type stop;
  // связанный поток
  HANDLE thread;
  // указатель на соответствующую работу (параметр потока)
  longjob_abstract_type *pjob;
 };

 // отображение номеров длительных переходов на связанные структуры
 // в виде вектора хранить нельзя, поскольку тогда при добавлении
 // нового элемента становятся невалидными указатели на старые
 std::map<int, longjob_data_type> m_ljobdata;
 // отображение переходов завершения (stop)
 // на номера соответствующих длительных переходов
 std::map<transition_abstract_type *, int> m_stopmap;

 // функция потока - выполняет вызов функции выполнения работы
 static DWORD WINAPI thr_proc(LPVOID param)
 {
  // выполнение работы
  static_cast<longjob_abstract_type *>(param)->run();
  return 0;
 }

public:
 transition_long_type allocate_long_transition(
  longjob_abstract_type &longjob)
 {
  // номер создаваемого длительного перехода
  int index = m_ljobdata.size();

  // разместим данные длительного перехода
  longjob_data_type &ljdata = m_ljobdata[index];
  ljdata.pjob = &longjob;

  // добавим переход завершения в отображение для поиска номера
  m_stopmap[&ljdata.stop] = index;

  // создадим и заполним содержимое длительного перехода
  transition_composite_type::content_type content;
  content.add_place(ljdata.started);
  content.add_place(ljdata.stopped);
  content.add_transition(ljdata.stop);
  content.add_arc(ljdata.started, ljdata.stop);
  content.add_arc(ljdata.stop, ljdata.stopped);
  content.add_token(ljdata.started);
  return transition_long_type(content, this, index);
 }
 void initialize_longjob(int i)
```

```
 {
  // инициализация работы - создание потока
  DWORD dwId;
  m_ljobdata[i].thread = ::CreateThread(
   NULL, 0,
   thr_proc, m_ljobdata[i].pjob,
   0, &dwId);
  assert(m_ljobdata[i].thread != NULL);
 }
 void finalize_longjob(int i)
 {
  // завершительные действия - освобождение ресурсов
  // к этому моменту поток уже закончил работу
  ::CloseHandle(m_ljobdata[i].thread);
 }

public:
 // инициализация датчика псевдослучайных чисел
 static void srandom(void)
 {
  srand((unsigned) time(NULL));
 }
 // генерация псевдослучайного числа в интервале [0, bound)
 static int random(int bound)
 {
  assert(bound > 0);
  return int(rand() / (RAND_MAX + 1.0) * bound);
 }
 // ожидание срабатывания перехода
 int wait(const petrinet_type::enabledlist_type &enabled)
 {
  // списки позиций 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> hdl;
  // проходим по всему списку переходов
  int pos = 0;
  petrinet_type::enabledlist_type::const_iterator it;
  for (it = enabled.begin(); it != enabled.end(); it++)
  {
   // если это не переход завершения работы, кладем в свободные
   if (m_stopmap.find(*it) == m_stopmap.end())
    free.push_back(pos);
   else
   {
    // иначе получаем хэндл связанного потока
    HANDLE h = m_ljobdata[m_stopmap[*it]].thread;
    // и проверяем, завершился ли он
    if (::WaitForSingleObject(h, 0) == WAIT_OBJECT_0)
     finished.push_back(pos);
```

```
    else
    {
     // если не завершился, добавляем хэндл на ожидание
     busy.push_back(pos);
     hdl.push_back(h);
    };
   };
   pos++;
  };
  // формируем код возврата
  int rc;
  // если есть завершившиеся работы, возвращаем
  // позицию одного из соответствующих переходов
  if (finished.size() > 0)
   rc = finished[random(finished.size())];
  // если есть свободные переходы, вернем позицию одного из них
  else if (free.size() > 0)
   rc = free[random(free.size())];
  else
  {
   // в остальных случаях дождемся завершения одного из потоков
   DWORD dw = ::WaitForMultipleObjects(
    hdl.size(), &hdl[0], FALSE, INFINITE);
   assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdl.size());
   // и вернем позицию соответствующего перехода
   rc = busy[dw - WAIT_OBJECT_0];
  };
  return rc;
 }
};
```

Приведенный код куда более громоздок по сравнению с приведенным ранее классом environment_random_type. Прежде всего, объявляются используемые классом окружения типы. Среди них интерфейс длительной работы, выполняемой в рамках одного длительного перехода, и класс длительного перехода. Последний унаследован от класса составного перехода и определяет обработчики событий активации и деактивации, которые инициируют начало и завершение выполнения длительной работы в параллельном потоке. Эти операции непосредственно реализует объект окружения, класс же длительного перехода передает ему необходимые для их инициирования данные.

С каждым длительным переходом сети ассоциирована структура данных longjob_data_type. Она содержит объявления внутренних элементов составного перехода, а также данные, необходимые для выполнения длительной работы в отдельном потоке. Заполнение такой структуры и добавление ее к контейнеру m_ljobdata осуществляется вместе с созданием и заполнением соответствующего составного перехода функцией allocate_long_transition по инициативе вызывающего кода. Она же добавляет соответствующий завершению длительной операции простой переход в свой внутренний список, который будет использоваться в дальнейшем для идентификации длительного перехода.

Функция initialize_longjob создает новый поток на основе функции thr_proc, передавая ему в качестве параметра указатель на требующую выполнения работу. Соответственно, функция finalize_longjob освобождает ресурсы после завершения потока.

Наконец, функция ожидания срабатывания перехода, прежде всего, анализирует весь список переданных переходов и делит их на две группы – переходы, которые не попадают во множество переходов завершения длительных операций, и, соответственно, попадающие в него. Для каждого перехода завершения определяется соответствующий номер длительной операции и факт завершения соответствующего потока. По признаку завершения потока эти переходы также делятся на две группы. После разделения на три группы осуществляется выбор срабатывающего перехода.

Первым делом проверяется множество переходов завершения, для которых соответствующие потоки уже завершены. Если оно не пусто, возвращается произвольный номер 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 номер. Наконец, если все разрешенные переходы – переходы завершения, потоки которых до сих пор не завершены, выполняется ожидание завершения любого из них, после чего возвращается соответствующий номер перехода.

При такой реализации функции ожидания практически все время тратится на ожидание завершения параллельных потоков, выполняющих длительные работы. В случае же, когда переход характеризует простые операции, к примеру, объекты синхронизации, работа сети продолжается без ожидания, вследствие чего построенная таким образом параллельная программа может быть достаточно эффективной.

Следуя принятому нами ранее подходу, реализуем код окружения с аналогичной функциональностью с помощью альтернативного программного интерфейса – POSIX Threads. Создание и уничтожение потоков происходит 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 environment_type: public
petrinet_type::environment_abstract_type
{
 // ...
private:
 // объекты mutex и cond для ожидания завершения потоков
 pthread_mutex_t m_mutex;
 pthread_cond_t m_cond;
 // номер последнего завершившегося потока
 int m_lastindex;

 // проверка кода возврата функций pthread_xxx
 static inline void chkzero(int retcode) { assert(retcode == 0); }

 // ...
};
```

Здесь также приведено объявление функции проверки кода возврата для всех функций pthreads, а также переменной, содержащей номер последнего завершившегося длительного перехода. Эта переменная нам требуется для обеспечения функциональности, аналогичной возврату значения при ожидании завершения одного из потоков.

Использование объектов синхронизации требует их инициализации и уничтожения, для чего классу окружения потребуется конструктор и деструктор:

```
class environment_type: public
petrinet_type::environment_abstract_type
{
 // ...
```

```
public:
 // вместе с объектом окружения создаются и уничтожаются
 // объекты синхронизации mutex и cond
 environment_type(void)
 {
  chkzero(::pthread_mutex_init(&m_mutex, NULL));
  chkzero(::pthread_cond_init(&m_cond, NULL));
 }
 ~environment_type(void)
 {
  chkzero(::pthread_cond_destroy(&m_cond));
  chkzero(::pthread_mutex_destroy(&m_mutex));
 }
 // ...
};
```

Для взаимодействия функции потока с объектами синхронизации ей на этот раз передается не только указатель на выполняемую работу, а следующая структура данных thr_param:

```
struct longjob_data_type
{
 // ...
 // связанный поток
 pthread_t thread;
 // структура параметров потока
 struct thr_param
 {
  // указатель на соответствующую работу
  longjob_abstract_type *pjob;
  // окружение
  environment_type *penv;
  // номер в списке длительных переходов окружения
  int index;
  // признак завершения выполнения
  bool finished;
 } param;
};
```

Соответственно, в функции allocate_long_transition потребуется инициализация этой структуры:

```
transition_long_type allocate_long_transition(longjob_abstract_type
&longjob)
{
 // ...
 // разместим данные длительного перехода
 longjob_data_type &ljdata = m_ljobdata[index];
 ljdata.param.pjob = &longjob;
 ljdata.param.penv = this;
 ljdata.param.index = index;

 // добавим переход завершения в отображение для поиска номера
 // ...
```

```
}
```

В соответствии с используемым программным интерфейсом изменяются функции инициализации выполнения работы и последующего освобождения ресурсов:

```
void initialize_longjob(int i)
{
 // установка признака - поток не завершен
 m_ljobdata[i].param.finished = false;
 // инициализация работы - создание потока
 chkzero(::pthread_create(
  &m_ljobdata[i].thread, NULL,
  thr_proc, &m_ljobdata[i].param));
}
void finalize_longjob(int i)
{
 // завершительные действия - освобождение ресурсов
 // к этому моменту поток уже закончил работу
 chkzero(::pthread_join(m_ljobdata[i].thread, NULL));
}
```

Изменению подлежит и функция потока, которая теперь помимо выполнения работы должна сообщить функции ожидания о факте завершения, при этом выложив номер соответствующего длительного перехода:

```
static void *thr_proc(void *param)
{
 longjob_data_type::thr_param *p =
  static_cast<longjob_data_type::thr_param *>(param);
 // выполнение работы
 p->pjob->run();
 // установка блокировки
 chkzero(::pthread_mutex_lock(&p->penv->m_mutex));
 // модификация признаков завершения потока
 p->penv->m_lastindex = p->index;
 p->finished = true;
 // 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:

```
int wait(const petrinet_type::enabledlist_type &enabled)
{
 chkzero(::pthread_mutex_lock(&m_mutex));
 // списки позиций 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 позиции
 // переходов завершения работ, которые еще не завершены
 std::map<int, int> busybyindex;
 // проходим по всему списку переходов
 int pos = 0;
 petrinet_type::enabledlist_type::const_iterator it;
 for (it = enabled.begin(); it != enabled.end(); it++)
 {
  // если это не переход завершения работы, кладем в свободные
  if (m_stopmap.find(*it) == m_stopmap.end())
   free.push_back(pos);
  else
  {
   // иначе проверяем, завершился ли поток
   if (m_ljobdata[m_stopmap[*it]].param.finished)
    finished.push_back(pos);
   else
    // если не завершился, добавляем информацию для ожидания
    busybyindex[m_stopmap[*it]] = pos;
  };
  pos++;
 };
 // формируем код возврата
 int rc;
 // если есть завершившиеся работы, возвращаем
 // позицию одного из соответствующих переходов
 if (finished.size() > 0)
  rc = finished[random(finished.size())];
 // если есть свободные переходы, вернем позицию одного из них
 else if (free.size() > 0)
  rc = free[random(free.size())];
 else
 {
  // в остальных случаях дождемся завершения одного из потоков
  chkzero(::pthread_cond_wait(&m_cond, &m_mutex));
  // и вернем позицию соответствующего перехода
  rc = busybyindex[m_lastindex];
 };
 chkzero(::pthread_mutex_unlock(&m_mutex));
 return rc;
}
```

В начале функции выполняется блокировка объекта m_mutex, по причине чего становится возможной проверка данных длительных переходов в части информации об их завершении. При выполнении же ожидания внутри функции pthread_cond_wait объект m_mutex временно освобождается, в результате чего завершившийся 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, в конце же функции wait блокировка снимается.

Реализуем на основе описанного окружения задачу с параллельным выполнением длительных операций. Не уходя далеко за примерами, рассмотрим снова задачу об обедающих философах. Будем считать, что каждому философу во время однократного приема пищи требуется произвести вычисления произвольной длительности. Отождествим длительную операцию с одним подходом к еде, как изображено на рис. 33. Для реализации этой операции создадим соответствующий класс длительной работы:

```
// длительная операция "философ ест"
class longjob_eating_type:
 public environment_type::longjob_abstract_type
{

private:
 int m_num;

 void run(void)
 {
  cout << "Философ " << m_num << " приступил к еде" << endl;

  // ... какие-либо длительные вычисления

  cout << "Философ " << m_num << " приступил к размышлениям" <<endl;
 }

public:
 longjob_eating_type(int num): m_num(num) {}

};
```

Код, реализующий создание и выполнение сети с параллельным выполнением таких работ, отличается от приведенного ранее кода на основе составных переходов лишь в части, касающейся создания этих переходов:

```
environment_type env;
const int N = 5;

// формируем список работ
vector<longjob_eating_type> eatingjob;
for (int i = 0; i < N; i++)
 eatingjob.push_back(longjob_eating_type(i));
```

```
// формируем длительные переходы на выполнение работ
vector<environment_type::transition_long_type> eating;
for (int i = 0; i < N; i++)
 eating.push_back(env.allocate_long_transition(eatingjob[i]));

// заполняем содержимое сети
vector<place_type> thinking(N), fork(N);
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]);

 // добавляем в сеть переходы
 content.add_transition(eating[i]);
};

for (int i = 0; i < N; i++)
{
 // входные дуги перехода eating
 content.add_arc(thinking[i], eating[i]);
 content.add_arc(fork[i], eating[i]);
 content.add_arc(fork[(i + 1) % N], eating[i]);

 // выходные дуги перехода eating
 content.add_arc(eating[i], thinking[i]);
 content.add_arc(eating[i], fork[i]);
 content.add_arc(eating[i], fork[(i + 1) % N]);
};

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 осуществляется создание длительных работ, после чего с помощью окружения создаются соответствующие составные переходы, которые позже вносятся в содержимое сети. В остальном же вызывающий код полностью идентичен рассмотренному ранее коду с использованием составных переходов.

Заключение

В связи с неотвратимо приближающимся достижением современными компьютерами своего предела тактовых частот параллельное программирование постепенно перестает быть узкоспециализированной дисциплиной для высокопроизводительных вычислений и приобретает все большую актуальность. Вследствие этого появляется немало публикаций, посвященных теоретическим вопросам распараллеливания последовательных программ, а также немало учебных пособий и статей, посвященных рассмотрению существующего инструментария. Помимо этого уже довольно давно создано немало моделей, успешно описывающих параллельное выполнение процессов и являющихся основой многих реальных разработок и некоторых малоизвестных инструментов организации параллельного выполнения программ. Таким моделям посвящено немало учебного материала теоретического характера, однако ощущается некоторый дефицит в области практического их рассмотрения с доступными примерами программной реализации с использованием современных средств. В результате рассмотрения таких моделей лишь с теоретической точки зрения они, порой, остаются вне практического арсенала программиста, в связи с чем, в силу консерватизма мышления многих программистов, возможность использования таких моделей зачастую даже не рассматривается.

Представленное описание некоторых возможных подходов к параллельному программированию, помимо очевидных целей любого учебного пособия, имеет также целью показать следующие два взаимосвязанных момента.

Во-первых, стояла 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 принципиальная сложность наглядного для человека представления с использованием таких средств параллельных алгоритмов и программ. В связи с этим встает ребром вопрос необходимости в использовании для параллельного программирования принципиально иных средств, не являющихся модификациями или надстройками существующих популярных языков. Одним из возможных направлений является декларативное программирование, что отчасти и было нами продемонстрировано, поскольку приведенные программы, хоть и были реализованы на императивном языке, зачастую по структуре представлялись декларативно.

Во-вторых, была сделана попытка проиллюстрировать следующий факт, вытекающий из рассмотрения тех же обстоятельств с обратной стороны. Несмотря на указанные сложности, зачастую, все же, императивные языки могут быть использованы для 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 совершенно разными. Такой подход позволит не уходить от имеющейся на сегодняшней 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 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.

Приложение 1. Шаблоны классов матрицы и вектора

Приводится пример реализации шаблона класса матрицы, предоставляющего интерфейс выполнения основных операций, а также унаследованный от него шаблон вектора, являющийся матрицей шириной в один столбец.

```
// матрица
template <class e_t>
class matrix_type
{
public:

 typedef e_t element_type;

private:

 int m_vsize, m_hsize;
 element_type *m_data;

public:

 // конструктор
 matrix_type(int vsize, int hsize):
  m_vsize(vsize), m_hsize(hsize)
 {
  assert(m_vsize > 0 && m_hsize > 0);

  m_data = new element_type[m_vsize * m_hsize];
 }

 // конструктор копирования
 matrix_type(const matrix_type &src):
  m_vsize(src.vsize()), m_hsize(src.hsize())
 {
  m_data = new element_type[m_vsize * m_hsize];
  this->operator =(src);
 }

 // деструктор
 ~matrix_type(void)
 {
  delete[] m_data;
 }


 // 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 единицы)
const element_type & operator ()(int i, int j) const
{
 assert(i > 0 && j > 0 && i <= vsize() && j <= hsize());

 return m_data[(i - 1) * hsize() + j - 1];
}

element_type & operator ()(int i, int j)
{
 return const_cast<element_type &>(
  static_cast<const matrix_type *>(this)->operator ()(i, j));
}


// присваивание матриц одинаковых размеров
matrix_type & operator =(const matrix_type &src)
{
 assert(vsize() == src.vsize() && hsize() == src.hsize());

 memcpy(m_data, src.m_data,
  m_vsize * m_hsize * sizeof(element_type));

 return *this;
}


// прибавление матрицы
matrix_type & operator +=(const matrix_type &src)
{
 assert(hsize() == src.hsize() && vsize() == src.vsize());

 #pragma omp parallel for
 for (int i = 1; i <= vsize(); i++)
  for (int j = 1; 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 = 1; i <= vsize(); i++)
   for (int j = 1; 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() && src1.vsize() == src2.vsize());

  matrix_type mtx = src1;
  return (mtx += src2);
 }

 // разность двух матриц
 friend
 matrix_type operator -(
  const matrix_type &src1, const matrix_type &src2)
 {
  assert(
   src1.hsize() == src2.hsize() && src1.vsize() == src2.vsize());

  matrix_type mtx = src1;
  return (mtx -= 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 = 1; i <= mtx.vsize(); i++)
  {
   for (int j = 1; j <= mtx.hsize(); j++)
   {
    element_type sum(0);
    for (int k = 1; k <= src1.hsize(); k++)
     sum += src1(i, k) * src2(k, j);
    mtx(i, j) = sum;
   };
  };
```

```
  return mtx;
 }
};


// вектор - матрица в один столбец
template <class 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
 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 единицы)
 const element_type & operator ()(int i) const
 {
  return static_cast<const base_type *>(this)->operator ()(i, 1);
 }

 element_type & operator ()(int i)
 {
  return static_cast<base_type *>(this)->operator ()(i, 1);
 }


 // присваивание
 vector_type & operator =(const base_type &src)
 {
  return static_cast<vector_type &>(
   static_cast<base_type *>(this)->operator =(src));
 }
};
```

Приложение 2. Классы построения и выполнения комплекса работ

Приводятся классы, используемые во второй главе 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 директив 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;

 // таблица зависимостей работ по номерам
 typedef std::vector<std::vector<int> > deplist_type;

 // номера работ в ярусно-параллельной форме
 typedef std::vector<std::vector<int> > 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)
{
 // вычислим ярусы всех работ на основе зависимостей
 std::vector<int> levlist(deplist.size());

 // множество номеров работ, ярусы которых еще не определены
 std::set<int> nondetermined;
 for (int i = 0; i < levlist.size(); i++)
  nondetermined.insert(i);

 // цикл определения ярусов работ
 while (nondetermined.size() > 0)
 {
  std::vector<int> determined;
  // пройдемся по всем неопределенным еще работам
  std::set<int>::iterator it;
  for (it = nondetermined.begin(); it != nondetermined.end(); it++)
  {
   // если зависимостей нет - нулевой ярус
   int lev = 0;
   // пройдемся по всем зависимостям, если есть
   for (int j = 0; j < deplist[*it].size(); j++)
   {
    // если зависимость найдена среди неопределенных
    if (nondetermined.find(deplist[*it][j]) !=
     nondetermined.end())
    {
```

```
     // значит текущая работа остается неопределенной
     lev = -1;
     break;
    }
    else
     // иначе ярус на единицу больше максимального
     lev = std::max(lev, levlist[deplist[*it][j]] + 1);
   };

   // если ярус определили, добавляем в определенные
   if (lev >= 0)
   {
    levlist[*it] = lev;
    determined.push_back(*it);
   };
  };

  // если нечего не удалось определить,
  // видимо, у нас циклические зависимости
  assert(determined.size() > 0);

  // выкинем из неопределенных те, что определили
  std::set<int> diff;
  std::set_difference(
   nondetermined.begin(), nondetermined.end(),
   determined.begin(), determined.end(),
   std::inserter(diff, diff.end()));
  nondetermined.swap(diff);
 };

 // построим ярусно-параллельную структуру
 // из всех работ на основе вычисленных ярусов
 multilevel_type multilevel(height(levlist));
 for (int i = 0; i < deplist.size(); i++)
  multilevel[levlist[i]].push_back(i);

 return multilevel;
}


// получение высоты ярусно-параллельной структуры
static int height(const std::vector<int> &level)
{
 int maxlevel = 0;
 for (int i = 0; i < level.size(); i++)
  maxlevel = std::max(maxlevel, level[i]);
 return maxlevel + 1;
}


// получение ширины ярусно-параллельной структуры
static int width(const multilevel_type &multilevel)
{
```

```
  int maxwidth = 0;
  for (int i = 0; i < multilevel.size(); i++)
   maxwidth = std::max(maxwidth, (int) multilevel[i].size());
  return maxwidth;
 }

public:

 // добавление работы
 void add_job(job_abstract_type &job)
 {
  // добавляемой работы не должно быть в списке
  assert(m_jobs.find(&job) == m_jobs.end());

  m_jobs.insert(jobs_type::value_type(&job, m_jobs.size()));
 }


 // 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());

  // выполним по очереди каждый ярус работ
  for (int l = 0; l < multilevel.size(); l++)
  {
   #pragma omp parallel for num_threads(multilevel[l].size())
   for (int i = 0; i < multilevel[l].size(); i++)
    joblist[multilevel[l][i]]->run();
  };
 }
};
```

Приложение 3. Классы построения и выполнения сетей конечных автоматов

Приводится последовательная версия классов, используемых в третьей главе для построения и выполнения автоматных сетей, содержащая директивы распараллеливания OpenMP.

```
// абстрактный тип автомата
class fsm_abstract_type
{
public:

 // тип входных и выходных данных
 typedef int data_type;

 // тип состояния автомата
 typedef int state_type;

 // количество входных каналов
 virtual int number_input(void) const = 0;

 // количество выходных каналов
 virtual int number_output(void) const = 0;

 // передать автомату вектор входных данных
 virtual void put_input(const std::vector<data_type> &input) = 0;

 // выполнить такт работы автомата
 virtual void do_work(void) = 0;

 // получить от автомата вектор выходных данных
 virtual std::vector<data_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:
```

```
 // количество входов и выходов автомата
 int m_inputnum, m_outputnum;

 // состояние
 state_type m_state;

 // таблица обработчиков
 typedef std::map<state_type, handler_type> handlertable_type;
 handlertable_type m_handlertable;

protected:

 // входные и выходные параметры автомата
 std::vector<data_type> m_input, m_output;

protected:

 void add_handler(state_type state, handler_type handler)
 {
  m_handlertable.insert(
   handlertable_type::value_type(state, handler));
 }


public:

 fsm_type(int inputnum, int outputnum):
  m_state(STATE_OFF),
  m_inputnum(inputnum), m_outputnum(outputnum),
  m_input(m_inputnum), m_output(m_outputnum)
 {
 }

 int number_input(void) const
 {
  return m_inputnum;
 }

 int number_output(void) const
 {
  return m_outputnum;
 }

 void put_input(const std::vector<data_type> &input)
 {
  m_input.assign(input.begin(), input.end());
 }

 void do_work(void)
 {
  // найти обработчик по текущему состоянию и вызвать его
  handlertable_type::iterator it = m_handlertable.find(m_state);
```

```
  if (it != m_handlertable.end())
   m_state = (this->*(it->second))(m_state);
 }

 std::vector<data_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
{
public:

 typedef std::pair<std::pair<int, int>, std::pair<int, int> >
  link_type;
 enum {PSEUDOFSM_NETINPUT = -1, PSEUDOFSM_NETOUTPUT = -2};

 // этот класс - для упрощения добавления связей
 class linkstore_type
 {
 private:
  std::vector<link_type> m_links;
 public:
  void add(int srcfsm, int srcnum, int dstfsm, int dstnum)
  {
   m_links.push_back(link_type(
    link_type::first_type(srcfsm, srcnum),
    link_type::second_type(dstfsm, dstnum)));
  }
  std::vector<link_type> get(void) const
  {
   return m_links;
  }
 };

 // абстрактный тип фабрики автоматной сети
 class factory_abstract_type
 {
 public:

  // количество автоматов в сети
  virtual int number_fsm(void) const = 0;

  // количество входных каналов сети
```

```
  virtual int number_input(void) const = 0;

  // количество выходных каналов сети
  virtual int number_output(void) const = 0;

  // получение списка всех связей
  virtual linkstore_type get_links(void) const = 0;

  // создание i-го автомата в сети
  virtual fsm_abstract_type *create_fsm(int i) = 0;

  // уничтожение его же
  virtual void destroy_fsm(int i, fsm_abstract_type *pfsm) = 0;

 };

private:

 // класс для работы с общими данными
 class shared_area_type
 {
  // общие данные (выходы всех автоматов и входы сети)
  std::vector<data_type> m_data;
  // позиции и размеры областей выходов автоматов и входов сети
  std::vector<int> m_outpos, m_outsize;
  // позиции входов автоматов и выходов сети
  std::vector<std::vector<int> > m_inpos;

 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 std::vector<link_type> &links)
  {
   // количество автоматов = максимальный указанный + 1
   int fsmnum = 0;
   for (int i = 0; i < links.size(); i++)
   {
    fsmnum = std::max(fsmnum, links[i].first.first);
    fsmnum = std::max(fsmnum, links[i].second.first);
   };
   fsmnum++;

   // кол-ва выходов автоматов (сначала) и входов сети (последний)
   std::vector<int> outsize(fsmnum + 1);
   // кол-ва входов автоматов (сначала) и выходов сети (последний)
   std::vector<int> insize(fsmnum + 1);
   // ищем максимальные номера входов и выходов для всех
   for (int i = 0; i < links.size(); i++)
   {
    int idxout = links[i].first.first;
    int idxin = links[i].second.first;
    idxout = (idxout != PSEUDOFSM_NETINPUT) ?
```

```
    idxout : outsize.size() - 1;
   idxin = (idxin != PSEUDOFSM_NETOUTPUT) ?
    idxin : insize.size() - 1;

   outsize[idxout] =
    std::max(outsize[idxout], links[i].first.second);
   insize[idxin] =
    std::max(insize[idxin], links[i].second.second);
  };
  // корректируем до размеров
  for (int i = 0; i < outsize.size(); i++)
   outsize[i]++;
  for (int i = 0; i < insize.size(); i++)
   insize[i]++;

  // формируем список позиций начал областей выходов
  int fullsize = 0;
  for (int i = 0; i < outsize.size(); i++)
  {
   m_outpos.push_back(fullsize);
   fullsize += outsize[i];
  };
  // создаем массив общих данных
  m_data.resize(fullsize);
  fill(m_data.begin(), m_data.end(), 0);
  // сохраняем размеры выходных областей в массиве общих данных
  m_outsize.assign(outsize.begin(), outsize.end());

  // создаем список списков позиций входов
  for (int i = 0; i < insize.size(); i++)
   m_inpos.push_back(std::vector<int>(insize[i]));

  // заполняем список списков позиций входов
  for (int i = 0; i < links.size(); i++)
  {
   int idxout = links[i].first.first;
   int idxin = links[i].second.first;
   idxout = (idxout != PSEUDOFSM_NETINPUT) ?
    idxout : m_outpos.size() - 1;
   idxin = (idxin != PSEUDOFSM_NETOUTPUT) ?
    idxin : m_inpos.size() - 1;
   int out = links[i].first.second;
   int in = links[i].second.second;

   m_inpos[idxin][in] = m_outpos[idxout] + out;
  };
 }

 // сохранение выходных данных конкретного автомата
 void put_output(int i, const std::vector<data_type> &output)
 {
  copy(output.begin(), output.end(), m_data.begin() + m_outpos[i]);
 }
```

```
  // сохранение входных данных сети
  void put_input(const std::vector<data_type> &input)
  {
   copy(input.begin(), input.end(),
    m_data.begin() + m_outpos.back());
  }

  // восстановление входных данных конкретного автомата
  std::vector<data_type> get_input(int i) const
  {
   std::vector<data_type> input;
   for (int j = 0; j < m_inpos[i].size(); j++)
    input.push_back(m_data[m_inpos[i][j]]);
   return input;
  }

  // восстановление выходных данных сети
  std::vector<data_type> get_output(void) const
  {
   std::vector<data_type> output;
   for (int j = 0; j < m_inpos.back().size(); j++)
    output.push_back(m_data[m_inpos.back()[j]]);
   return output;
  }
 };

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.get_links().get())
 {
  // создание автоматов
  m_pfsm.resize(m_factory.number_fsm());
  #pragma omp parallel for
  for (int i = 0; i < m_pfsm.size(); i++)
   m_pfsm[i] = m_factory.create_fsm(i);
 }

 ~fsmnet_type(void)
 {
  // уничтожение автоматов
  #pragma omp parallel for
  for (int i = m_pfsm.size() - 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 std::vector<data_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_pfsm.size(); 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_pfsm.size(); i++)
     m_shared.put_output(i, m_pfsm[i]->get_output());
   };
  }

  std::vector<data_type> get_output(void) const
  {
   return m_shared.get_output();
  }

  bool is_off(void) const
  {
   // сеть в начальном состоянии, когда все автоматы в начальном
   bool rc = true;
   #pragma omp parallel for reduction(&&: rc)
   for (int i = 0; i < m_pfsm.size(); i++)
    rc = (rc && m_pfsm[i]->is_off());
   return rc;
  }
};
```

Приложение 4. Классы построения и выполнения сетей Петри

Приводится набор классов, предоставляющий возможность построения и выполнения рассмотренных в четвертой главе сетей Петри и 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:

 typedef std::vector<transition_abstract_type *> enabledlist_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;

 // 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(); }

 void fire(int number) { assert(false); }

};


// составной переход (сеть Петри)
class transition_composite_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]);
  if (arcmap.find(arc) == arcmap.end())
   arcmap.insert(arcmap_type::value_type(arc, weight));
  else
   arcmap[arc] += weight;
 }

 // построение матрицы кратности дуг
 arcmatrix_type build_matrix(const arcmap_type &arcmap) const
 {
  arcmatrix_type matrix(m_trmap.size());
  for (int i = 0; i < matrix.size(); i++)
  {
   std::vector<int> vec(m_plmap.size());
   for (int j = 0; j < vec.size(); j++)
   {
    arcmap_type::const_iterator it;
    it = arcmap.find(arcmap_type::key_type(i, j));
    vec[j] = (it != arcmap.end()) ? it->second : 0;
   };
   matrix[i] = vec;
  };
  return matrix;
 }

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(m_plmap.find(&pl) != m_plmap.end());
 assert(tokens > 0);

 if (m_tokmap.find(m_plmap[&pl]) == m_tokmap.end())
  m_tokmap.insert(tokmap_type::value_type(m_plmap[&pl], tokens));
 else
  m_tokmap[m_plmap[&pl]] += 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_matrixin(void) const
{
 return build_matrix(m_inmap);
}
arcmatrix_type get_matrixout(void) const
{
 return build_matrix(m_outmap);
```

```
  }

  // получение начальной разметки
  marking_type get_marking(void) const
  {
   marking_type marking(m_plmap.size());
   for (int i = 0; i < marking.size(); i++)
   {
    tokmap_type::const_iterator it = m_tokmap.find(i);
    marking[i] = (it != m_tokmap.end()) ? it->second : 0;
   };
   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;

 // набор состояний локальных переходов - разрешены или активны
 std::vector<bool> m_enabled_or_active;

 // полный список разрешенных сейчас переходов, включая внутренние
 enabledlist_type m_enabled_full;

 // размещение этих переходов по локальным номерам
 std::vector<int> m_location;

 // смещение начала области каждого локального перехода
 std::vector<int> m_offset;

 // полное обновление списка разрешенных переходов
 void refresh_enabled(void)
 {
  // сформируем набор всех переходов
  std::set<int> trchg;
  for (int i = 0; i < m_trlist.size(); i++)
   trchg.insert(i);
  // передадим на частичное обновление
```

```
  refresh_enabled(trchg);
 }

 // частичое обновление списка разрешенных переходов
 void refresh_enabled(const std::set<int> &trchg)
 {
  // перебираем все переданные переходы
  std::set<int>::const_iterator it;
  for (it = trchg.begin(); it != trchg.end(); it++)
  {
   int i = *it;
   // если переход не активен, рассматриваем вопрос разрешения
   bool enabled = true;
   if (!m_trlist[i]->is_active())
   {
    #pragma omp parallel for reduction(&&: enabled)
    for (int j = 0; j < m_mtxin[i].size(); j++)
     enabled = enabled && (m_marking[j] - m_mtxin[i][j] >= 0);
   };
   m_enabled_or_active[i] = enabled;
  };

  // соберем все разрешенные переходы и их размещение
  m_enabled_full.clear();
  m_location.clear();
  for (int i = 0; i < m_trlist.size(); i++)
  {
   m_offset[i] = m_enabled_full.size();
   if (m_enabled_or_active[i])
   {
    // если переход разрешен и не активен, добавим его
    if (!m_trlist[i]->is_active())
    {
     m_enabled_full.push_back(m_trlist[i]);
     m_location.push_back(i);
    }
    else
    {
     // если активен, получаем внутренние разрешенные
     enabledlist_type in = m_trlist[i]->get_enabled();
     // добавляем в конец текущего списка
     m_enabled_full.insert(
      m_enabled_full.end(), in.begin(), in.end());
     // формируем информацию о размещении
     m_location.insert(m_location.end(), in.size(), i);
    };
   };
  };
 }

public:

 transition_composite_type(const content_type &content):
```

```
 m_pllist(content.get_pllist()),
 m_trlist(content.get_trlist()),
 m_mtxin(content.get_matrixin()),
 m_mtxout(content.get_matrixout()),
 m_marking_init(content.get_marking()),
 m_enabled_or_active(m_trlist.size()),
 m_offset(m_trlist.size())
{}

void activate(void)
{
 m_marking = m_marking_init;
 refresh_enabled();
}

bool is_active(void) const
{
 return (m_enabled_full.size() > 0);
}

enabledlist_type get_enabled(void) const
{
 return m_enabled_full;
}

void fire(int number)
{
 assert(number >= 0 && number < m_enabled_full.size());

 int local = m_location[number];
 int lower = number - m_offset[local];

 // запомним текущую разметку
 std::vector<int> marking_pre = m_marking;

 // если переход активен, выполним внутренний
 if (m_trlist[local]->is_active())
  m_trlist[local]->fire(lower);
 else
 {
  // иначе изымем входные фишки
  #pragma omp parallel for
  for (int j = 0; j < m_mtxin[local].size(); 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
```

```
   #pragma omp parallel for
   for (int j = 0; j < m_mtxout[local].size(); j++)
    m_marking[j] += m_mtxout[local][j];
  };

  // сформируем набор локальных переходов,
  // разрешение которых могло измениться
  std::set<int> trchg;
  // прежде всего, добавим сработавший локальный переход
  trchg.insert(local);
  // переберем все позиции, разметка которых изменилась
  for (int j = 0; j < m_marking.size(); j++)
   if (m_marking[j] != marking_pre[j])
    // соберем все переходы, для которых они являются входными
    for (int i = 0; i < m_trlist.size(); i++)
     if (m_mtxin[i][j] > 0)
      trchg.insert(i);
  // обновим с учетом сформированного набора
  refresh_enabled(trchg);
 }
};


// сеть Петри верхнего уровня
class petrinet_type: public transition_composite_type
{
public:

 typedef transition_composite_type::content_type content_type;

 // абстрактный тип среды, в которой "живет" сеть Петри
 class environment_abstract_type
 {
 public:
  // ожидание срабатывания одного из переданных переходов
  virtual int wait(const enabledlist_type &enabled) = 0;
 };

public:

 petrinet_type(const content_type &content):
  transition_composite_type(content)
 {}

 void live(environment_abstract_type &env)
 {
  activate();
  while (is_active())
   fire(env.wait(get_enabled()));
 }
};
```

Библиографический список

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. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. – Ростов-на-Дону: Изд-во ООО «ЦВВР», 2003. – 208 с.
4. Вавилов К.В. Программирование за... 1 (одну) минуту // Компьютер Price. – 2002. – № 31. – С. 288–293.
5. Вентцель Е.С. Исследование операций. – М.: Сов. радио, 1972. – 552 с.
6. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. Серия «Научное издание». – СПб.: БХВ-Петербург, 2002. – 608 с.
7. Глебов А.Н. Параллельное программирование в функциональном стиле. – 2003 [Электронный ресурс] URL: http://www.softcraft.ru/parallel/ppfs.shtml.
8. Глушков В.М. Синтез цифровых автоматов. – М.: Физматгиз, 1962. – 476 с.
9. Дехтярь М.И. Введение в схемы, автоматы и алгоритмы. – 2007 [Электронный ресурс] URL: http://www.intuit.ru/department/ds/introsaa/.
10. Котов В.Е . Введение в теорию схем программ. – Новосибирск: Наука, 1978. – 258 с.
11. Котов В.Е. Сети Петри. – М.: Наука, 1984. – 160 с.
12. 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.
13. Кузнецов Б.П. Психология автоматного программирования // BYTE-Россия. – 2000. №11. – С. 22-29.
14. Кузнецов С.Д. Блеск и нищета легковесных процессов // Computerworld Россия. – 1996. №31.
15. Лескин А.А., Мальцев П.А., Спиридонов А.М. Сети Петри в моделировании и управлении. – Л.: Наука, 1989. – 133 с.

16. 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.
17. 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.
18. 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.
19. Любченко В.С. Конечно-автоматная технология программирования // Труды международной научно-методической конференции «Телематика'2001». СПб.: СПбГИТМО(ТУ), 2001. – С. 127-128.
20. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. Серия "Мастер программ". – СПб.: БХВ-Петербург, 2002. – 400 с.
21. Питерсон Дж. Теория сетей Петри и моделирование систем: Пер. с англ. – М.: Мир, 1984. – 264 с.
22. Трахтенброт Б. А., Барздинь Я. М. Конечные автоматы (поведение и синтез). – М.: Наука, 1970. – 400 с.
23. Хоар Ч. Взаимодействующие последовательные процессы: Пер. с англ. – М.: Мир, 1989. – 264 с.
24. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – 628 с.
25. 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.
26. 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.

27. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному // Мир ПК. – 2002, №2. – С.144-149.
28. Шалыто А.А., Туккель Н.И. Преобразование итеративных алгоритмов в автоматные // Программирование. – 2002. № 5. – С. 12-26.
29. Шалыто А.А., Туккель Н.И. Программирование с явным выделением состояний // Мир ПК. – 2001, №8, №9. – С.116-121; С.132-138.
30. 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.
31. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие. – Мн.: БГУ, 2002. – 323 с.
32. Элементы параллельного программирования / Вальковский В.А., Котов В.Е., Марчук А.Г., Миренков Н.Н. – М.: Радио и связь, 1983. – 240 с.
33. Lee Edward A. The problem with threads // IEEE Computer. – Vol. 39, № 5, May 2006. – P. 33-42.
34. Snir M., Otto S., Huss-Lederman S., etc. MPI – The Complete Reference: The MPI Core. – 2-nd edn. – Cambridge: MIT Press, 1998. – 426 p.
35. 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.
36. MPI: A Message-Passing Interface Standard. (Version 1.1: June, 1995) [Электронный ресурс] URL: http://www.mcs.anl.gov/mpi/standard.html.
37. MPI-2: Extensions to the Message-Passing Interface. [Электронный ресурс] URL: http://www.mcs.anl.gov/mpi/standard.html.

38. OpenMP C and C++ Application Program Interface (Version 1.0: October, 1998) [Электронный ресурс] URL: http://www.openmp.org/mp-documents/cspec10.pdf.
39. OpenMP C and C++ Application Program Interface (Version 2.0: March, 2002) [Электронный ресурс] URL: http://www.openmp.org/mp-documents/cspec20.pdf.
40. Wagner F., Schmuki R., Wagner T., Wolstenholme P. Modeling Software with Finite State Machines: A Practical Approach. – Auerbach Publications, 2006. – 392 p.
41. Wagner F. Moore or Mealy model? April 2005. [Электронный ресурс] URL: http://www.stateworks.com/technology/TN10-Moore-Or-Mealy-Model/.

Оглавление

Илья Евгеньевич Федотов

НЕКОТОРЫЕ ПРИЕМЫ
ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Учебное пособие

Редактор С.П. Куликов
Учебное пособие напечатано в авторской редакции

Подписано в печать 12.11.2008. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л.10,93 Усл. кр.-отт. 43,72. Уч.-изд. л. 11,75
Тираж 100 экз. С 649

Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный институт радиотехники,
электроники и автоматики (технический университет)»
119454, Москва, пр. Вернадского, 78