Part 1

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,

Оглавление

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.

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 надеж-

ность 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


```
 // 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

```
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;
```


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 попу-

лярного 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 нагляд-

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:

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

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:

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

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;
```

гим 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.

0	1	2		0	1	2		0	1	2
3	4	5	×	3	4	5	=	3	4	5
6	7	8		6	7	8		6	7	8

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}:

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

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

но 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.


```
  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 немно-

ся еще одним квадратным блоком. На рис. 5 изображено распределение результатов умножения после первых двух шагов между четырьмя процессами.

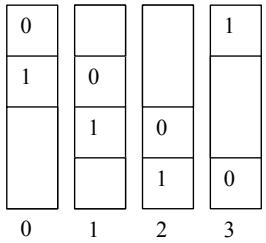

Рис. 5

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,
```


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:

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.

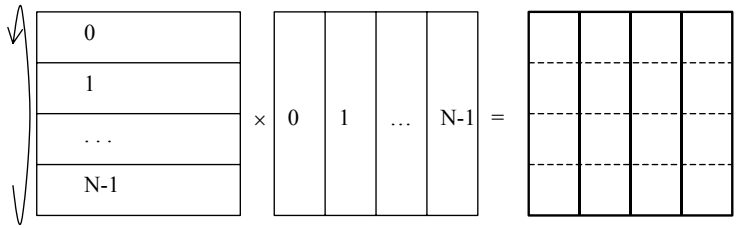

Рис. 4

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

$$RES = ML \cdot MR. \qquad (7)$$

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 пополняет-

```
// 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

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

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.

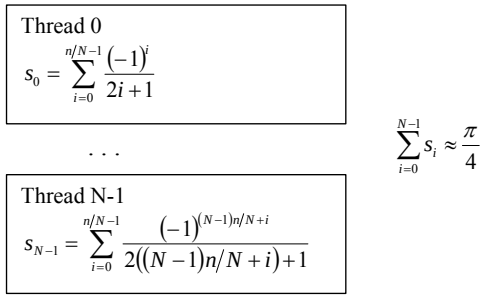

Рис. 1

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;
```


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
```

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;
```



```
};
// 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++)
```


нимый 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.

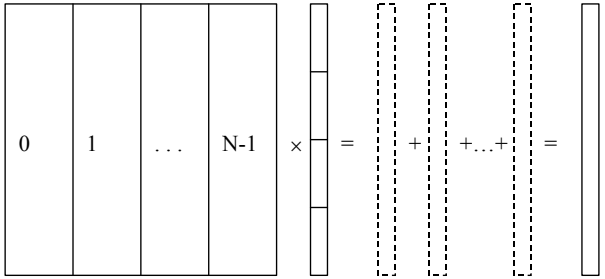

Рис. 3

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

```
 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 хра-


```
  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

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
```

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.

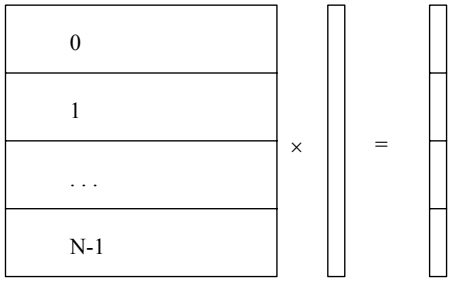

Рис. 2

В первом рассматриваемом нами случае элементы матрицы распределяются по процессам горизонтальными блоками (рис. 2). Каждый процесс содержит в своей памяти один горизонтальный

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 созда-

ваемых 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
```


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

```
// ... 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


```
 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 вы-

ченными 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:

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

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:

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

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();
```

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 полу-

полнению 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

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:

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

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

ся 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.

ков, 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

```
  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 одина-

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,

ты 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_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 рабо-

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

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,

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:

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 значе-

ниями 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