12. Параллельные методы решения дифференциальных уравнений в частных производных

12. Параллельные методы решения дифференциальных уравнений в частных производных 1
12.1. Последовательные методы решения задачи Дирихле 2
12.2. Организация параллельных вычислений для систем с общей памятью 3
12.2.1. Использование OpenMP для организации параллелизма 3
12.2.2. Проблема синхронизации параллельных вычислений 4
12.2.3. Возможность неоднозначности вычислений в параллельных программах 7
12.2.4. Проблема взаимоблокировки 8
12.2.5. Исключение неоднозначности вычислений 9
12.2.6. Волновые схемы параллельных вычислений 11
12.2.7. Балансировка вычислительной нагрузки процессоров 16
12.3. Организация параллельных вычислений для систем с распределенной памятью 17
12.3.1. Разделение данных 17
12.3.2. Обмен информацией между процессорами 18
12.3.3. Коллективные операции обмена информацией 20
12.3.4. Организация волны вычислений 21
12.3.5. Блочная схема разделения данных 22
12.3.6. Оценка трудоемкости операций передачи данных 24
12.4. Краткий обзор раздела 25
12.5. Обзор литературы 26
12.6. Контрольные вопросы 26
12.7. Задачи и упражнения 27

Дифференциальные уравнения в частных производных представляют собой широко применяемый математический аппарат при разработке моделей в самых разных областях науки и техники. К сожалению, явное решение этих уравнений в аналитическом виде оказывается возможным только в частных простых случаях, и, как результат, возможность анализа математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения. Объем выполняемых при этом вычислений обычно является значительным и использование высокопроизводительных вычислительных систем является традиционным для данной области вычислительной математики. Проблематика численного решения дифференциальных уравнений в частных производных является областью интенсивных исследований (см., например, Березин и Жидков (1966), Тихонов и Самарский (1977), Fox, et al. (1988)).

Рассмотрим в качестве учебного примера *проблему численного решения задачи Дирихле для уравнения Пуассона*, определяемую как задачу нахождения функции $u = u(x, y)$, удовлетворяющей в области определения D уравнению:

$$\begin{cases} \dfrac{\delta^2 u}{\delta x^2} + \dfrac{\delta^2 u}{\delta y^2} = f(x, y), & (x, y) \in D, \\ u(x, y) = g(x, y), & (x, y) \in D^0, \end{cases}$$

и принимающей значения $g(x, y)$ на границе D^0 области D (f и g являются функциями, задаваемыми при постановке задачи). Подобная модель может быть использована для описания установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и деформации упругих пластин и др. Данный пример часто используется в качестве учебно-практической задачи при изложении возможных способов организации эффективных параллельных вычислений (см. Немнюгин и Стесик (2002), Корнеев (1999), Pfister (1998)).

Для простоты изложения материала в качестве области задания D функции $u(x, y)$ далее будет использоваться единичный квадрат:

$$D = \{(x, y) \in D : 0 \le x, y \le 1\}.$$

12.1. Последовательные методы решения задачи Дирихле

Одним из наиболее распространенных подходов численного решения дифференциальных уравнений является *метод конечных разностей* (*метод сеток*) (см., например, Березин и Жидков (1966), Тихонов и Самарский (1977), Pfister (1995)). Следуя этому подходу, область решения D представляется в виде дискретного (как правило, равномерного) набора (*сетки*) точек (*узлов*). Так, например, прямоугольная сетка в области D может быть задана в виде (рис. 12.1):

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih,\ y_i = jh,\ 0 \le i, j \le N+1, \\ \qquad\qquad h = 1/(N+1), \end{cases}$$

где величина N задает количество узлов по каждой из координат области D.

Обозначим оцениваемую при подобном дискретном представлении аппроксимацию функции $u(x, y)$ в точках (x_i, y_j) через u_{ij}. Тогда, используя *пятиточечный шаблон* (см. рис. 12.1) для вычисления значений производных, мы можем представить уравнение Пуассона в *конечно-разностной форме*:

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}\,.$$

Данное уравнение может быть разрешено относительно u_{ij}:

$$u_{ij} = 0.25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij})\,.$$

Разностное уравнение, записанное в подобной форме, позволяет определять значение u_{ij} по известным значениям функции $u(x, y)$ в соседних узлах используемого шаблона. Данный результат служит основой для построения различных *итерационных схем* решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений u_{ij}, а затем эти значения последовательно уточняются в соответствии с приведенным соотношением. Так, например, *метод Гаусса-Зейделя* для проведения итераций уточнения использует правило:

$$u_{ij}^{k} = 0.25(u_{i-1,j}^{k} + u_{i+1,j}^{k-1} + u_{i,j-1}^{k} + u_{i,j+1}^{k-1} - h^2 f_{ij})\,,$$

по которому очередное *k*-ое приближение значения u_{ij} вычисляется по последнему *k*-ому приближению значений $u_{i-1,j}$ и $u_{i,j-1}$ и предпоследнему (*k-1*)-ому приближению значений $u_{i+1,j}$ и $u_{i,j+1}$. Выполнение итераций обычно продолжается до тех пор, пока получаемые в результате итераций изменения значений u_{ij} не станут меньше некоторой заданной величины (*требуемой точности вычислений*). Сходимость описанной процедуры (получение решения с любой желаемой точностью) является предметом всестороннего математического анализа (см., например, Березин и Жидков (1966), Тихонов и Самарский (1977), Pfister (1995)), здесь же отметим, что последовательность решений, получаемых методом сеток, равномерно сходится к решению задачи Дирихле, а погрешность решения имеет порядок h^2.

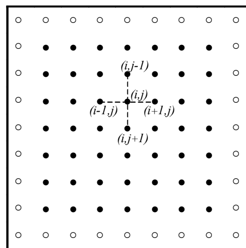

Рис. 12.1. Прямоугольная сетка в области D (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах - сверху вниз)

Рассмотренный алгоритм (метод Гаусса-Зейделя) на псевдокоде, приближенном к алгоритмическому языку C++, может быть представлен в виде:

```
// Алгоритм 12.1
// Метод Гаусса-Зейделя
do {
  dmax = 0; // максимальное изменение значений u
```

```
  for ( i=1; i<N+1; i++ )
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                 u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      dm = fabs(temp-u[i][j]);
      if ( dmax < dm ) dmax = dm;
    }
} while ( dmax > eps );
```

Алгоритм 12.1. Последовательный алгоритм Гаусса-Зейделя

(напомним, что значения u_{ij} при индексах $i, j = 0, N+1$ являются граничными, задаются при постановке задачи и не изменяются в ходе вычислений).

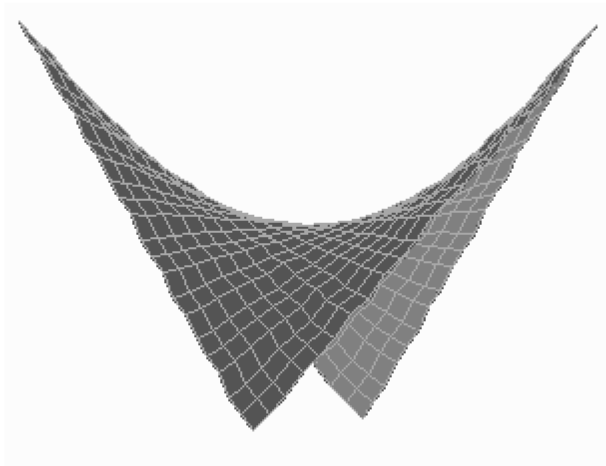

Рис. 12.2. Вид функции $u(x, y)$ в примере для задачи Дирихле

Для примера на рис. 12.2 приведен вид функции $u(x, y)$, полученной для задачи Дирихле при следующих граничных условиях:

$$\begin{cases} f(x,y) = 0, & (x,y) \in D, \\ 100 - 200x, & y = 0, \\ 100 - 200y, & x = 0, \\ -100 + 200x, & y = 1, \\ -100 + 200y, & x = 1, \end{cases}$$

Общее количество итераций метода Гаусса-Зейделя составило 210 при точности решения $eps = 0.1$ и $N = 100$ (в качестве начального приближения величин u_{ij} использовались значения, сгенерированные датчиком случайных чисел из диапазона [-100,100]).

12.2. Организация параллельных вычислений для систем с общей памятью

Как следует из приведенного описания, сеточные методы характеризуются значительной вычислительной трудоемкостью:

$$T_1 = kmN^2,$$

где N есть количество узлов по каждой из координат области D, m - число операций, выполняемых методом для одного узла сетки, k - количество итераций метода до выполнения условия остановки.

12.2.1. Использование OpenMP для организации параллелизма

Рассмотрим возможные способы организации параллельных вычислений для сеточных методов на многопроцессорных вычислительных системах с общей памятью. При изложении материала будем

предполагать, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (*symmetric multiprocessors, SMP*).

Обычный подход при организации вычислений для подобных систем – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки не зависимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное программирование, так и уже имеющиеся языки программирования, расширенные некоторым набором операторов для параллельных вычислений.

Оба указанных подхода приводят к необходимости значительной переработки существующего программного обеспечения, и это в значительной степени затрудняет широкое распространение параллельных вычислений. Как результат, в последнее время активно развивается еще один подход к разработке параллельных программ, когда указания программиста по организации параллельных вычислений добавляются в программу при помощи тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный операторный текст программы остается неизменным, и по нему в случае отсутствия препроцессора компилятор может построить исходный последовательный программный код. Препроцессор же, будучи примененным, заменяет директивы параллелизма на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки).

Рассмотренный выше подход является основой *технологии OpenMP* (см., например, Chandra, et al. (2000)), наиболее широко применяемой в настоящее время для организации параллельных вычислений на многопроцессорных системах с общей памятью. В рамках данной технологии директивы параллелизма используются для выделения в программе *параллельных областей* (*parallel regions*), в которых последовательный исполняемый код может быть разделен на несколько раздельных командных *потоков* (*threads*). Далее эти потоки могут исполняться на разных процессорах вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков программного кода (см. рис. 12.3). Подобный принцип организации параллелизма получил наименование "*вилочного*" (*fork-join*) или *пульсирующего параллелизма*. Более полная информация по технологии OpenMP может быть получена в в разделе 5, в дополнительной литературе (см., например, Chandra, et al. (2000)) или в информационных ресурсах сети Интернет; в данном разделе возможности OpenMP будут излагаться в объеме, необходимом для демонстрации возможных способов разработки параллельных программ для рассматриваемого учебного примера решения задачи Дирихле.

12.2.2. Проблема синхронизации параллельных вычислений

Первый вариант параллельного алгоритма для метода сеток может быть получен, если разрешить произвольный порядок пересчета значений u_{ij} . Программа для данного способа вычислений может быть представлена в следующем виде:

```
//Алгоритм 12.2
// Параллельный метод Гаусса-Зейделя (первый вариант)
omp_lock_t dmax_lock;
omp_init_lock (dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax) \
                         private(i,temp,d)
  for ( i=1; i<N+1; i++ ) {
#pragma omp parallel for shared(u,n,dmax) \
                         private(j,temp,d)
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                 u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
```

```
      d = fabs(temp-u[i][j])
      omp_set_lock(dmax_lock);
        if ( dmax < d ) dmax = d;
      omp_unset_lock(dmax_lock);
    } // конец вложенной параллельной области
  } // конец внешней параллельной области
} while ( dmax > eps );
```

Алгоритм 12.2. Первый вариант параллельного алгоритма Гаусса-Зейделя

Следует отметить, что программа получена из исходного последовательного кода путем добавления директив и операторов обращения к функциям библиотеки OpenMP (эти дополнительные строки в программе выделены темным шрифтом, обратная наклонная черта в конце директив означает продолжение директив на следующих строках программы).

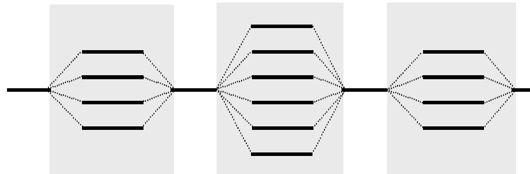

Рис. 12.3. Параллельные области, создаваемые директивами OpenMP

Как следует из текста программы, параллельные области в данном примере задаются директивой **parallel for**, являются вложенными и включают в свой состав операторы цикла **for**. Компилятор, поддерживающий технологию OpenMP, разделяет выполнение итераций цикла между несколькими потоками программы, количество которых обычно совпадает с числом процессоров в вычислительной системе. Параметры директивы **shared** и **private** определяют доступность данных в потоках программы – переменные, описанные как **shared**, являются общими для потоков, для переменных с описанием **private** создаются отдельные копии для каждого потока, которые могут использоваться в потоках независимо друг от друга.

Наличие общих данных обеспечивает возможность взаимодействия потоков. В этом плане разделяемые переменные могут рассматриваться как *общие ресурсы потоков* и, как результат, их использование должно выполняться с соблюдением *правил взаимоисключения* (изменение каким-либо потоком значений общих переменных должно приводить к блокировке доступа к модифицируемым данным для всех остальных потоков). В данном примере таким разделяемым ресурсом является величина **dmax**, доступ потоков к которой регулируется специальной служебной переменной (*семафором*) **dmax_lock** и функциями **omp_set_lock** (блокировка доступа) и **omp_unset_lock** (снятие запрета на доступ). Подобная организация программы гарантирует единственность доступа потоков для изменения разделяемых данных. Участки программного кода (блоки между обращениями к функциям **omp_set_lock** и **omp_unset_lock**), для которых обеспечивается взаимоисключение, обычно именуются *критическими секциями*.

Результаты вычислительных экспериментов приведены в табл. 12.1 (здесь и далее для параллельных программ, разработанных с использованием технологии OpenMP, использовался четырехпроцессорный сервер кластера Нижегородского университета с процессорами Pentium III, 700 Mhz, 512 RAM).

Таблица 12.1. Результаты вычислительных экспериментов для параллельных вариантов алгоритма Гаусса-Зейделя (p=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 12.1)		Параллельный алгоритм 12.2			Параллельный алгоритм 12.3		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	1,97	0,03	210	0,03	2,03
200	273	0,34	273	11,22	0,03	273	0,14	2,43
300	305	0,88	305	29,09	0,03	305	0,36	2,43
400	318	3,78	318	54,20	0,07	318	0,64	5,90
500	343	6,00	343	85,84	0,07	343	1,06	5,64
600	336	8,81	336	126,38	0,07	336	1,50	5,88
700	344	12,11	344	178,30	0,07	344	2,42	5,00
800	343	16,41	343	234,70	0,07	343	8,08	2,03

900	358	20,61	358	295,03	0,07	358	11,03	1,87
1000	351	25,59	351	366,16	0,07	351	13,69	1,87
2000	367	106,75	367	1585,84	0,07	367	56,63	1,89
3000	370	243,00	370	3598,53	0,07	370	128,66	1,89

(***k*** – количество итераций, ***T*** – время в сек., ***S*** – ускорение)

Оценим полученный результат. Разработанный параллельный алгоритм является корректным, т.е. обеспечивающим решение поставленной задачи. Использованный при разработке подход обеспечивает достижение практически максимально возможного параллелизма – для выполнения программы может быть задействовано вплоть до N^2 процессоров. Тем не менее, результат не может быть признан удовлетворительным – программа будет работать медленно и ускорение вычислений от использования нескольких процессоров окажется не столь существенным (результаты экспериментов говорят скорее всего **о замедлении вычислений**). Основная причина такого положения дел – чрезмерно высокая *синхронизация* параллельных участков программы. В нашем примере каждый параллельный поток после усреднения значений u_{ij} должен проверить (и возможно, изменить) значение величины **dmax**. Разрешение на использование переменной может получить только один поток – все остальные

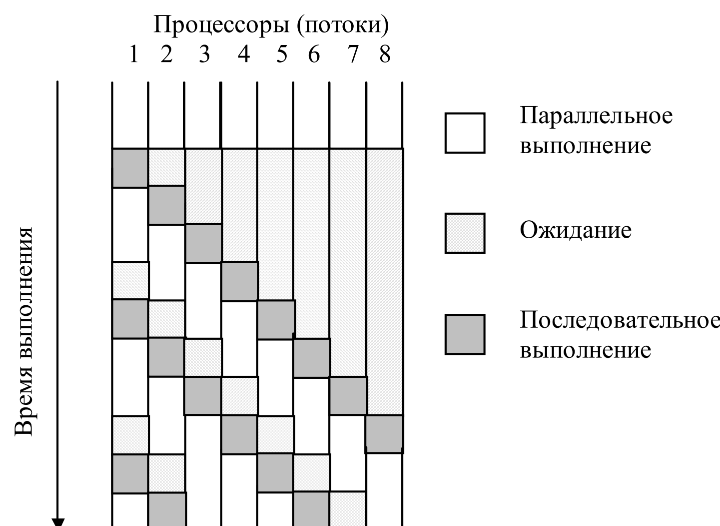

Рис. 12.4. Пример возможной схемы выполнения параллельных потоков при наличии синхронизации (взаимоисключения)

потоки должны быть блокированы. После освобождения общей переменной управление может получить следующий поток и т.д. В результате необходимости синхронизации доступа многопотоковая параллельная программа превращается фактически в последовательно выполняемый код, причем менее эффективный, чем исходный последовательный вариант, т.к. организация синхронизации приводит к дополнительным вычислительным затратам – см. рис. 12.4. Следует обратить внимание, что, несмотря на идеальное распределение вычислительной нагрузки между процессорами, для приведенного на рис. 12.4 соотношения параллельных и последовательных вычислений, в каждый текущий момент времени (после момента первой синхронизации) только не более двух процессоров одновременно выполняют действия, связанные с решением задачи. Подобный эффект вырождения параллелизма из-за интенсивной синхронизации параллельных участков программы обычно именуются *сериализацией* (*serialization*).

Как показывают выполненные рассуждения, путь для достижения эффективности параллельных вычислений лежит в уменьшении необходимых моментов синхронизации параллельных участков программы. Так, в нашем примере мы можем ограничиться распараллеливанием только одного внешнего цикла **for**. Кроме того, для снижения количества возможных блокировок применим для оценки максимальной погрешности многоуровневую схему расчета: пусть параллельно выполняемый поток первоначально формирует локальную оценку погрешности **dm** только для своих обрабатываемых данных (одной или нескольких строк сетки), затем при завершении вычислений поток сравнивает свою оценку **dm** с общей оценкой погрешности **dmax**.

Новый вариант программы решения задачи Дирихле имеет вид:

```
//Алгоритм 12.3
```

```
// Параллельный метод Гаусса-Зейделя (первый вариант)
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax)\
                         private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                 u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // конец параллельной области
} while ( dmax > eps );
```

Алгоритм 12.3. Второй вариант параллельного алгоритма Гаусса-Зейделя

Как результат выполненного изменения схемы вычислений, количество обращений к общей переменной **dmax** уменьшается с N^2 до N раз, что должно приводить к существенному снижению затрат на синхронизацию потоков и уменьшению проявления эффекта сериализации вычислений. Результаты экспериментов с данным вариантом параллельного алгоритма, приведенные в табл. 12.1, показывают существенное изменение ситуации – ускорение в ряде экспериментов оказывается даже большим, чем используемое количество процессоров (такой эффект *сверхлинейного ускорения* достигается за счет наличия у каждого из процессоров вычислительного сервера своей быстрой кэш-памяти). Следует также обратить внимание, что улучшение показателей параллельного алгоритма достигнуто при снижении максимально возможного параллелизма (для выполнения программы может использоваться не более N процессоров).

12.2.3. Возможность неоднозначности вычислений в параллельных программах

Последний рассмотренный вариант организации параллельных вычислений для метода сеток обеспечивает практически максимально возможное ускорение выполняемых расчетов – так, в экспериментах данное ускорение достигало величины 5.9 при использовании четырехпроцессорного вычислительного сервера. Вместе с этим необходимо отметить, что разработанная вычислительная схема расчетов имеет важную принципиальную особенность – порождаемая при вычислениях последовательность обработки данных может различаться при разных запусках программы даже при одних и тех же исходных параметрах решаемой задачи. Данный эффект может проявляться в силу изменения каких-либо условий выполнения программы (вычислительной нагрузки, алгоритмов синхронизации потоков и т.п.), что может повлиять на временные соотношения между потоками (см. рис. 12.5). Взаиморасположение потоков по области расчетов может быть различным: одни потоки могут опережать другие и, обратно, часть потоков могут отставать (при этом, характер взаиморасположения может меняться в ходе вычислений). Подобное поведение параллельных участков программы обычно именуется *состязанием потоков* (*race condition*).

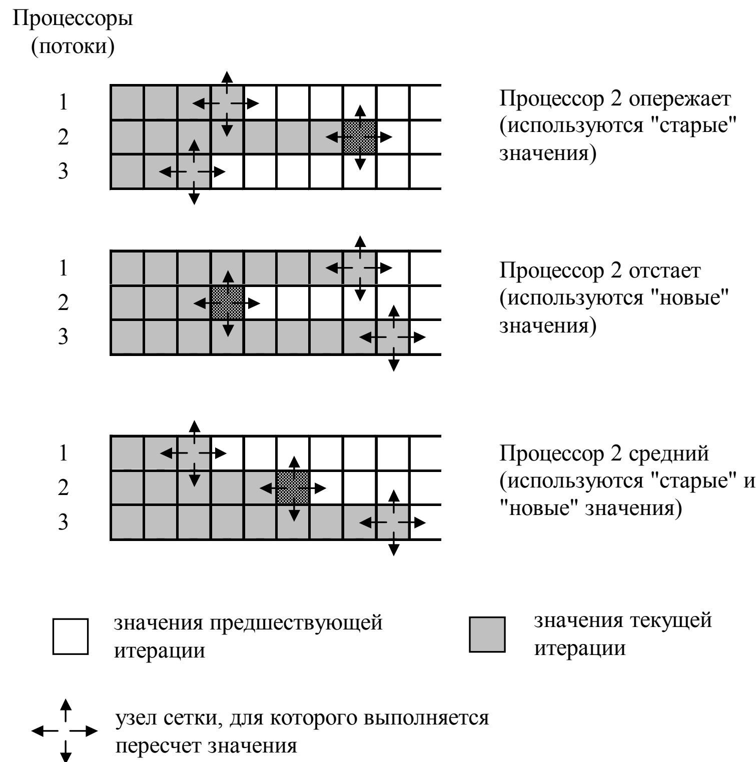

Рис. 12.5. Возможные различные варианты взаиморасположения параллельных потоков (состязание потоков)

В рассматриваемом примере при вычислении нового значения u_{ij} в зависимости от условий выполнения могут использоваться разные (от предыдущей или текущей итераций) оценки соседних значений по вертикали. Тем самым, количество итераций метода до выполнения условия остановки и, самое главное, конечное решение задачи может различаться при повторных запусках программы. Получаемые оценки величин u_{ij} будут соответствовать точному решению задачи в пределах задаваемой точности, но, тем не менее, могут быть различными. Использование вычислений такого типа для сеточных алгоритмов получило наименование *метода хаотической релаксации* (*chaotic relaxation*).

Следует отметить, что в общем случае неоднозначность результатов параллельных вычислений является нежелательной, поскольку повторяемость результатов работы программ является основой для возможности проверки правильности программного обеспечения. Стремление к однозначности получаемых результатов расчета приводит к необходимости соблюдения важного принципа параллельного программирования, в соответствии с которым временная динамика выполнения параллельных потоков не должна сказываться на выполнение параллельных программ.

12.2.4. Проблема взаимоблокировки

Возможный подход для получения однозначных результатов (уход от состязания потоков) может состоять в ограничении доступа к узлам сетки, которые обрабатываются в параллельных потоках программы. Для этого можно ввести набор семафоров row_lock[N], который позволит потокам закрывать доступ к "своим" строкам сетки:

```
// поток обрабатывает i строку сетки
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i-1]);
// обработка i строку сетки
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i-1]);
```

Закрыв доступ к своим данным, параллельный поток уже не будет зависеть от динамики выполнения других параллельных участков программы. Результат вычислений потока однозначно определяется значениями данных в момент начала расчетов.

Данный подход позволяет продемонстрировать еще одну проблему, которая может возникать в ходе параллельных вычислений. Эта проблема состоит в том, что при организации доступа к множественным общим переменным может возникать конфликт между параллельными потоками и этот конфликт не может быть разрешен успешно. Так, в приведенном фрагменте программного кода при обработке потоками двух последовательных строк (например, строк 1 и 2) может сложиться ситуация, когда потоки блокируют сначала свои строки 1 и 2 и только затем переходят к блокировке оставшихся строк (см. рис. 12.6). В этом случае доступ к необходимым строкам не может быть обеспечен ни для одного потока – возникает неразрешимая ситуация, обычно именуемая *тупиком*. Как можно показать, необходимым условием тупика является наличие цикла в графе распределения и запросов ресурсов. В рассматриваемом примере уход от цикла может состоять в строго последовательной схеме блокировки строк потока:

```
    // поток обрабатывает i строку сетки
    omp_set_lock(row_lock[i+1]);
    omp_set_lock(row_lock[i]);
    omp_set_lock(row_lock[i-1]);
    // <обработка i строку сетки>
    omp_unset_lock(row_lock[i+1]);
    omp_unset_lock(row_lock[i]);
    omp_unset_lock(row_lock[i-1]);
```

(следует отметить, что и эта схема блокировки строк может оказаться тупиковой, если рассматривать модифицированную задачу Дирихле, в которой горизонтальные границы являются "склеенными").

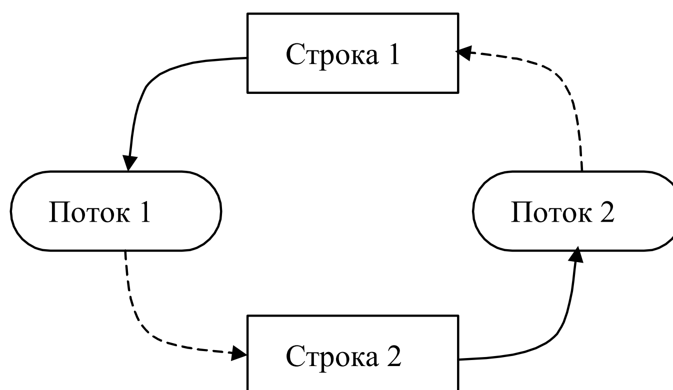

Рис. 12.6. Ситуация тупика при доступе к строкам сетки (поток 1 владеет строкой 1 и запрашивает строку 2, поток 2 владеет строкой 2 и запрашивает строку 1)

12.2.5. Исключение неоднозначности вычислений

Подход, рассмотренный в п. 12.2.4, уменьшает эффект состязания потоков, но не гарантирует единственности решения при повторении вычислений. Для достижения однозначности необходимо использование дополнительных вычислительных схем.

Возможный и широко применяемый в практике расчетов способ состоит в разделении места хранения результатов вычислений на предыдущей и текущей итерациях метода сеток. Схема такого подхода может быть представлена в следующем общем виде:

```
//Алгоритм 12.4
// Параллельный метод Гаусса-Якоби
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax) \
                         private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      un[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
```

```
                 u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-un[i][j])
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // конец параллельной области
  for ( i=1; i<N+1; i++ ) // обновление данных
    for ( j=1; j<N+1; j++ )
      u[i][j] = un[i][j];
} while ( dmax > eps );
```

Алгоритм 12.4. Параллельная реализация сеточного метода Гаусса-Якоби

Как следует из приведенного алгоритма, результаты предыдущей итерации запоминаются в массиве **u**, новые вычисления значения запоминаются в дополнительном массиве **un**. Как результат, независимо от порядка выполнения вычислений для проведения расчетов всегда используются значения величин u_{ij} от предыдущей итерации метода. Такая схема реализации сеточных алгоритмов обычно именуется *методом Гаусса-Якоби*. Этот метод гарантирует однозначность результатов независимо от способа распараллеливания, но требует использования большого дополнительного объема памяти и обладает меньшей (по сравнению с алгоритмом Гаусса-Зейделя) скоростью сходимости. Результаты расчетов с последовательным и параллельным вариантами метода приведены в табл. 12.2.

Таблица 12.2. Результаты вычислительных экспериментов для алгоритма Гаусса-Якоби (***p***=4)

Размер сетки	Последовательный метод Гаусса-Якоби (алгоритм 12.4)		Параллельный метод, разработанный по аналогии с алгоритмом 12.3		
	k	***T***	***k***	***T***	***S***
100	5257	1,39	5257	0,73	1,90
200	23067	23,84	23067	11,00	2,17
300	26961	226,23	26961	29,00	7,80
400	34377	562,94	34377	66,25	8,50
500	56941	1330,39	56941	191,95	6,93
600	114342	3815,36	114342	2247,95	1,70
700	64433	2927,88	64433	1699,19	1,72
800	87099	5467,64	87099	2751,73	1,99
900	286188	22759,36	286188	11776,09	1,93
1000	152657	14258,38	152657	7397,60	1,93
2000	337809	134140,64	337809	70312,45	1,91
3000	655210	247726,69	655210	129752,13	1,91

(***k*** – количество итераций, ***T*** – время в сек., ***S*** – ускорение)

Иной возможный подход для устранения взаимозависимости параллельных потоков состоит в применении схемы чередования обработки четных и нечетных строк (red/black row alternation scheme), когда выполнение итерации метода сеток подразделяется на два последовательных этапа, на первом из которых обрабатываются строки только с четными номерами, а затем на втором этапе - строки с нечетными номерами (см. рис. 12.7). Данная схема может быть обобщена на применение одновременно и к строкам, и к столбцам (блочное разбиение) области расчетов.

Рассмотренная схема чередования строк не требует по сравнению с методом Гаусса-Якоби какой-либо дополнительной памяти и обеспечивает однозначность решения при многократных запусках программы. Но следует заметить, что оба рассмотренных в данном пункте подхода могут получать результаты, не совпадающие с решением задачи Дирихле, найденном при помощи последовательного алгоритма. Кроме того, эти вычислительные схемы имеют меньшую область и худшую скорость сходимости, чем исходный вариант метода Гаусса-Зейделя.

Этап 1

Этап 2

граничные значения

значения предшествующей итерации

значения после этапа 1 текущей итерации

значения после этапа 2 текущей итерации

Рис. 12.7. Схема чередования обработки четных и нечетных строк

12.2.6. Волновые схемы параллельных вычислений

Рассмотрим теперь возможность построения параллельного алгоритма, который выполнял бы только те вычислительные действия, что и последовательный метод (может быть только в некотором ином порядке) и, как результат, обеспечивал бы получение точно таких же решений исходной вычислительной задачи. Как уже было отмечено выше, в последовательном алгоритме каждое очередное *k*-ое приближение значения u_{ij} вычисляется по последнему *k*-ому приближению значений $u_{i-1,j}$ и $u_{i,j-1}$ и предпоследнему (*k-1*)-ому приближению значений $u_{i+1,j}$ и $u_{i,j+1}$. Как результат, при требовании совпадения результатов вычислений последовательных и параллельных вычислительных схем в начале каждой итерации метода только одно значение u_{11} может быть пересчитано (возможности для распараллеливания нет). Но далее после пересчета u_{11} вычисления могут выполняться уже в двух узлах сетки u_{12} и u_{21} (в этих узлах выполняются условия последовательной схемы), затем после пересчета узлов u_{12} и u_{21} - в узлах u_{13}, u_{22} и u_{31} и т.д. Обобщая сказанное, можно увидеть, что выполнение итерации метода сеток можно разбить на последовательность шагов, на каждом из которых к вычислениям окажутся подготовленными узлы вспомогательной диагонали сетки с номером, определяемом номером этапа – см. рис. 12.8. Получаемая в результате вычислительная схема получила наименование *волны* или *фронта вычислений*, а алгоритмы, получаемые на ее основе, - *методами волновой обработки данных* (*wavefront* or *hyperplane methods*). Следует отметить, что в нашем случае размер волны (степень возможного параллелизма) динамически изменяется в ходе вычислений – волна нарастает до своего пика, а затем затухает при приближении к правому нижнему узлу сетки.

Таблица 12.3. Результаты экспериментов для параллельных вариантов алгоритма Гаусса-Зейделя с волновой схемой расчета (***p***=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 12.1)		Параллельный алгоритм 12.5			Параллельный алгоритм 12.6		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	0,30	0,21	210	0,16	0,40
200	273	0,34	273	0,86	0,40	273	0,59	0,58
300	305	0,88	305	1,63	0,54	305	1,53	0,57
400	318	3,78	318	2,50	1,51	318	2,36	1,60
500	343	6,00	343	3,53	1,70	343	4,03	1,49

600	336	8,81	336	5,20	1,69	336	5,34	1,65
700	344	12,11	344	8,13	1,49	344	10,00	1,21
800	343	16,41	343	12,08	1,36	343	12,64	1,30
900	358	20,61	358	14,98	1,38	358	15,59	1,32
1000	351	25,59	351	18,27	1,40	351	19,30	1,33
2000	367	106,75	367	69,08	1,55	367	65,72	1,62
3000	370	243,00	370	149,36	1,63	370	140,89	1,72

(***k*** – количество итераций, ***T*** – время в сек., ***S*** – ускорение)

Рис. 12.8. Движение фронта волны вычислений

Возможная схема параллельного метода, основанного на эффекте волны вычислений, может быть представлена в следующей форме:

```
//Алгоритм 12.5
// Параллельный метод Гаусса-Зейделя (волновая схема)
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
  // нарастание волны (nx – размер волны)
  for ( nx=1; nx<N+1; nx++ ) {
    dm[nx] = 0;
#pragma omp parallel for shared(u,nx,dm) \
                         private(i,j,temp,d)
    for ( i=1; i<nx+1; i++ ) {
      j    = nx + 1 – i;
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                 u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm[i] < d ) dm[i] = d;
    } // конец параллельной области
  }
  // затухание волны
  for ( nx=N-1; nx>0; nx-- ) {
#pragma omp parallel for shared(u,nx,dm) \
                         private(i,j,temp,d)
    for ( i=N-nx+1; i<N+1; i++ ) {
      j    = 2*N - nx – I + 1;
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                 u[i][j-1]+u[i][j+1]–h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm[i] < d ) dm[i] = d;
    } // конец параллельной области
  }
#pragma omp parallel for shared(n,dm,dmax) \
                         private(i)
  for ( i=1; i<nx+1; i++ ) {
    omp_set_lock(dmax_lock);
      if ( dmax < dm[i] ) dmax = dm[i];
    omp_unset_lock(dmax_lock);
  } // конец параллельной области
} while ( dmax > eps );
```

Алгоритм 12.5. Параллельный алгоритм, реализующий волновую схему вычислений

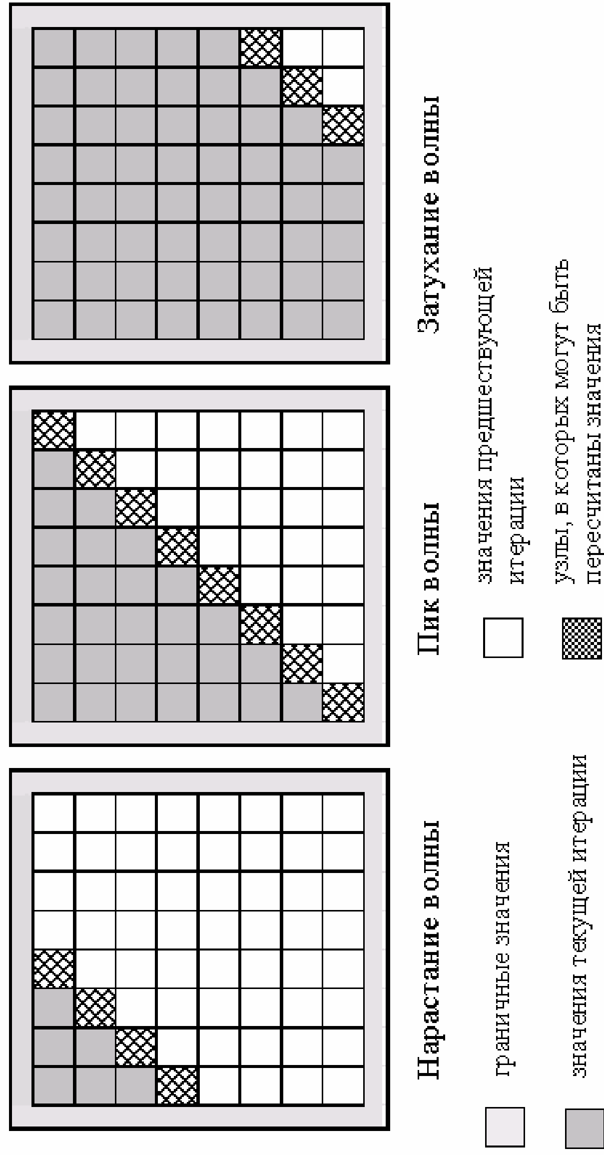

Рис. 6.8. Движение фронта волны вычислений

При разработке алгоритма, реализующего волновую схему вычислений, оценку погрешности решения можно осуществлять для каждой строки в отдельности (массив значений **dm**). Этот массив является общим для всех выполняемых потоков, однако, синхронизации доступа к элементам не требуется, так как потоки используют всегда разные элементы массива (фронт волны вычислений содержит только по одному узлу строк сетки).

После обработки всех элементов волны в составе массива **dm** находится максимальная погрешность выполненной итерации вычислений. Однако именно эта последняя часть расчетов может оказаться наиболее неэффективной из-за высоких дополнительных затрат на синхронизацию. Улучшение ситуации, как и ранее, может быть достигнуто за счет увеличения размера последовательных участков и сокращения, тем самым, количества необходимых взаимодействий параллельных участков вычислений. Возможный вариант реализации такого подхода может состоять в следующем:

```
chunk = 200; // размер последовательного участка
#pragma omp parallel for shared(n,dm,dmax) \
                         private(i,d)
  for ( i=1; i<nx+1; i+=chunk ) {
    d = 0;
    for ( j=i; j<i+chunk; j++ )
      if ( d < dm[j] ) d = dm[j];
    omp_set_lock(dmax_lock);
      if ( dmax < d ) dmax = d;
    omp_unset_lock(dmax_lock);
  } // конец параллельной области
```

Подобный прием укрупнения последовательных участков вычислений для снижения затрат на синхронизацию именуется *фрагментированием* (*chunking*). Результаты экспериментов для данного варианта параллельных вычислений приведены в табл. 12.3.

Следует обратить внимание еще на один момент при анализе эффективности разработанного параллельного алгоритма. Фронт волны вычислений плохо соответствует правилам использования *кэша* - быстродействующей дополнительной памяти компьютера, используемой для хранения копии наиболее часто используемых областей оперативной памяти. Эффективное использование кэша может существенно повысить (в десятки раз) быстродействие вычислений. Размещение данных в кэше может происходить или предварительно (при использовании тех или иных алгоритмов предсказания потребности в данных) или в момент извлечения значений из основной оперативной памяти. При этом подкачка данных в кэш осуществляется не одиночными значениями, а небольшими группами – *строками кэша* (*cache line*). Загрузка значений в строку кэша осуществляется из последовательных элементов памяти; типовые размеры строки кэша обычно равны 32, 64, 128, 256 байтам (дополнительная информация по организации памяти может быть получена, например, в (см., например, Хамахер и Вранешич (2003)). Как результат, эффект наличия кэша будет наблюдаться, если выполняемые вычисления используют одни и те же данные многократно (*локальность обработки* данных) и осуществляют доступ к элементам памяти с последовательно возрастающими адресами (*последовательность доступа*).

В рассматриваемом нами алгоритме размещение данных в памяти осуществляется по строкам, а фронт волны вычислений располагается по диагонали сетки, и это приводит к низкой эффективности использования кэша. Возможный способ улучшения ситуации – опять же укрупнение вычислительных операций и рассмотрение в качестве распределяемых между процессорами действий процедуру обработки некоторой прямоугольной подобласти (*блока*) сетки области расчетов - см. рис. 12.9.

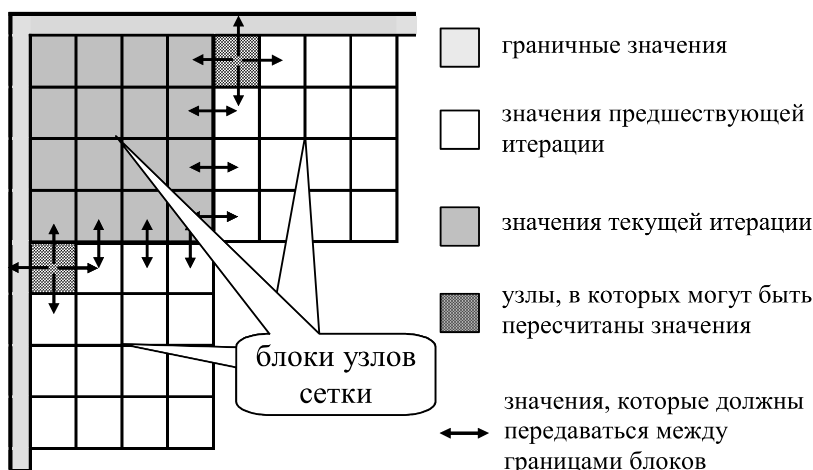

Рис. 12.9. Блочное представление сетки области расчетов

Порождаемый на основе такого подхода метод вычислений в самом общем виде может быть описан следующим образом (блоки образуют в области расчётов прямоугольную решётку размера NBxNB):

```
//Алгоритм 12.6
// Параллельный метод Гаусса-Зейделя (блочная волновая схема)
do {
  // нарастание волны (размер волны равен nx+1)
  for ( nx=0; nx<NB; nx++ ) { // NB количество блоков
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = nx - i;
      // <обработка блока с координатами (i,j)>
    } // конец параллельной области
  }
  // затухание волны
  for ( nx=NB-2; nx>-1; nx-- ) {
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = 2*(NB-1) - nx - i;
      // <обработка блока с координатами (i,j)>
    } // конец параллельной области
  }
  // <определение погрешности вычислений>
} while ( dmax > eps );
```

Алгоритм 12.6. Блочный подход к методу волновой обработки данных

Вычисления в предлагаемом алгоритме происходят в соответствии с волновой схемой обработки данных – вначале вычисления выполняются только в левом верхнем блоке с координатами (0,0), далее для обработки становятся доступными блоки с координатами (0,1) и (1,0) и т.д. – см. результаты экспериментов в табл. 12.3.

Блочный подход к методу волновой обработки данных существенным образом меняет состояние дел – обработку узлов можно организовать построчно, доступ к данным осуществляется последовательно по элементам памяти, перемещенные в кэш значения используются многократно. Кроме того, поскольку обработка блоков будет выполняться на разных процессорах и блоки не пересекаются по данным, при таком подходе будут отсутствовать и накладные расходы для обеспечения однозначности (*когерентности*) кэшей разных процессоров.

Наилучшие показатели использования кэша будут достигаться, если в кэше будет достаточно места для размещения не менее трех строк блока (при обработке строки блока используются данные трех строк блока одновременно). Тем самым, исходя из размера кэша, можно определить рекомендуемый максимально-возможный размер блока. Так, например, при кэше 8 Кб и 8-байтовых значениях данных этот размер составит приближенно 300 (8Кб/3/8). Можно определить и минимально-допустимый размер

блока из условия совпадения размеров строк кэша и блока. Так, при размере строки кэша 256 байт и 8-байтовых значениях данных размер блока должен быть кратен 32.

Последнее замечание необходимо сделать о взаимодействии граничных узлов блоков. Учитывая граничное взаимодействие, соседние блоки целесообразно обрабатывать на одних и тех же процессорах. В противном случае можно попытаться так определить размеры блоков, чтобы объем пересылаемых между процессорами граничных данных был минимален. Так, при размере строки кэша в 256 байт, 8-байтовых значениях данных и размере блока 64x64 объем пересылаемых данных 132 строки кэша, при размере блока 128x32 – всего 72 строки. Такая оптимизация имеет наиболее принципиальное значение при медленных операциях пересылки данных между кэшами процессоров, т.е. для систем с неоднородным доступом к памяти (nonuniform memory access - NUMA).

12.2.7. Балансировка вычислительной нагрузки процессоров

Как уже отмечалось ранее, вычислительная нагрузка при волновой обработке данных изменяется динамически в ходе вычислений. Данный момент следует учитывать при распределении вычислительной нагрузки между процессорами. Так, например, при фронте волны из 5 блоков и при использовании 4 процессоров обработка волны потребует двух параллельных итераций, во время второй из которых будет задействован только один процессор, а все остальные процессоры будут простаивать, дожидаясь завершения вычислений. Достигнутое ускорение расчетов в этом случае окажется равным 2.5 вместо потенциально возможного значения 4. Подобное снижение эффективности использования процессоров становится менее заметным при большой длине волны, размер которой может регулироваться размером блока. Как общий результат, можно отметить, что размер блока определяет *степень разбиения* (*granularity*) вычислений для распараллеливания и является параметром, подбирая значения которого, можно управлять эффективностью параллельных вычислений.

Для обеспечения равномерности (*балансировки*) загрузки процессоров можно задействовать еще один подход, широко используемый для организации параллельных вычислений. Этот подход состоит в том, что все готовые к выполнению в системе вычислительные действия организуются в виде *очереди заданий*. В ходе вычислений освободившийся процессор может запросить для себя работу из этой очереди; появляющиеся по мере обработки данных дополнительные задания пополняют задания очереди. Такая схема балансировки вычислительной нагрузки между процессорами является простой, наглядной и эффективной. Это позволяет говорить об использовании очереди заданий как об *общей модели организации параллельных вычислений* для систем с общей памятью.

Рассмотренная схема балансировки может быть задействована и для рассматриваемого примера. В самом деле, в ходе обработки фронта текущей волны происходит постепенное формирование блоков следующей волны вычислений. Эти блоки могут быть задействованы для обработки при нехватке достаточной вычислительной нагрузки для процессоров

Общая схема вычислений с использованием очереди заданий может быть представлена в следующем виде:

```
//Алгоритм 12.7
// Параллельный метод Гаусса-Зейделя (блочная волновая схема, очередь заданий)
// <инициализация служебных данных>
// <загрузка в очередь указателя на начальный блок>
// взять блок из очереди (если очередь не пуста)
while ( (pBlock=GetBlock()) != NULL ) {
  // <обработка блока>
  // отметка готовности соседних блоков
  omp_set_lock(pBlock->pNext.Lock); // сосед справа
    pBlock->pNext.Count++;
    if ( pBlock->pNext.Count == 2 )
      PutBlock(pBlock->pNext);
  omp_unset_lock(pBlock->pNext.Lock);
  omp_set_lock(pBlock->pDown.Lock); // сосед снизу
    pBlock->pDown.Count++;
    if ( pBlock->pDown.Count == 2 )
      PutBlock(pBlock->pDown);
  omp_unset_lock(pBlock->pDown.Lock);
} // завершение вычислений, т.к. очередь пуста
```

Алгоритм 12.7. Общая схема вычислений с использованием очереди

Для описания имеющихся в задаче блоков узлов сетки в алгоритме используется структура со следующим набором параметров:

- **Lock** – семафор, синхронизирующий доступ к описанию блока,
- **pNext** – указатель на соседний справа блок,
- **pDown** – указатель на соседний снизу блок,
- **Count** – счетчик готовности блока к вычислениям (количество готовых границ блока).

Операции для выборки из очереди и вставки в очередь указателя на готовый к обработке блок узлов сетки обеспечивают соответственно функции *GetBlock* и *PutBlock*.

Как следует из приведенной схемы, процессор извлекает блок для обработки из очереди, выполняет необходимые вычисления для блока и отмечает готовность своих границ для соседних справа и снизу блоков. Если при этом оказывается, что у соседних блоков являются подготовленными обе границы, процессор передает эти блоки для запоминания в очередь заданий.

Использование очереди заданий позволяет решить практически все оставшиеся вопросы организации параллельных вычислений для систем с общей памятью. Развитие рассмотренного подхода может предусматривать уточнение правил выделения заданий из очереди для согласования с состояниями процессоров (близкие блоки целесообразно обрабатывать на одних и тех же процессорах), расширение числа имеющихся очередей заданий и т.п. Дополнительная информация по этим вопросам может быть получена, например, в Xu and Hwang (1998).

12.3. Организация параллельных вычислений для систем с распределенной памятью

Использование процессоров с распределенной памятью является другим общим способом построения многопроцессорных вычислительных систем. Актуальность таких систем становится все более высокой в последнее время в связи с широким развитием высокопроизводительных кластерных вычислительных систем (см. раздел 1).

Многие проблемы параллельного программирования (состязание вычислений, тупики, сериализация) являются общими для систем с общей и распределенной памятью. Момент, который отличает параллельные вычисления с распределенной памятью, состоит в том, что взаимодействие параллельных участков программы на разных процессорах может быть обеспечено только при помощи *передачи сообщений* (*message passing*).

Следует отметить, что процессор с распределенной памятью является, как правило, более сложным вычислительным устройством, чем процессор в многопроцессорной системе с общей памятью. Для учета этих различий в дальнейшем процессор с распределенной памятью будет именоваться как *вычислительный сервер* (сервером может быть, в частности, многопроцессорная система с общей памятью). При проведении всех ниже рассмотренных экспериментов использовались 4 компьютера с процессорами Pentium IV, 1300 Mhz, 256 RAM, 100 Mbit Fast Ethernet.

12.3.1. Разделение данных

Первая проблема, которую приходится решать при организации параллельных вычислений на системах с распределенной памяти, обычно состоит в выборе способа разделения обрабатываемых данных между вычислительными серверами. Успешность такого разделения определяется достигнутой степенью локализации вычислений на серверах (в силу больших временных задержек при передаче сообщений интенсивность взаимодействия серверов должна быть минимальной).

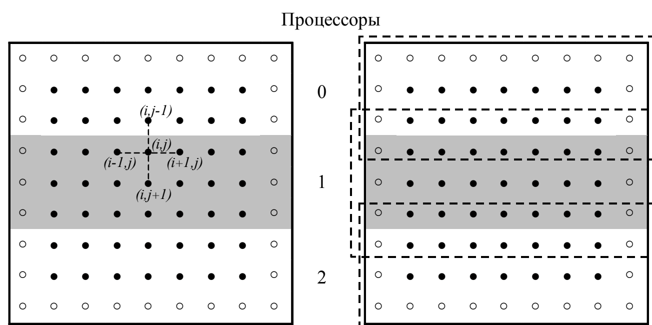

Рис. 12.10. Ленточное разделение области расчетов между процессорами (кружки представляют граничные узлы сетки)

В рассматриваемой учебной задаче по решению задачи Дирихле возможны два различных способа разделения данных – *одномерная* или *ленточная* схема (см. рис. 12.10) или *двухмерное* или *блочное* разбиение (см. рис. 12.9) вычислительной сетки. Дальнейшее изложение учебного материала будет проводиться на примере первого подхода; блочная схема будет рассмотрена позднее в более кратком виде.

При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы (не уменьшая общности, далее будем рассматривать только горизонтальные полосы). Число полос определяется количеством процессоров, размер полос обычно является одинаковым, узлы горизонтальных границ (первая и последняя строки) включаются в первую и последнюю полосы соответственно. Полосы для обработки распределяются между процессорами.

Основной момент при организации вычислений с подобным разделением данных состоит в том, что на процессор, выполняющий обработку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки (получаемые в результате расширенные полосы показаны на рис. 12.10 справа пунктирными рамками). Продублированные граничные строки полос используются только при проведении расчетов, пересчет же этих строк происходит в полосах своего исходного месторасположения. Тем самым дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток.

12.3.2. Обмен информацией между процессорами

Параллельный вариант метода сеток при ленточном разделении данных состоит в обработке полос на всех имеющихся серверах одновременно в соответствии со следующей схемой работы:

```
//Алгоритм 12.8
// Параллельный метод Гаусса-Зейделя (распределенная память, ленточная схема)
// действия, выполняемые на каждом процессоре
do {
  // <обмен граничных строк полос с соседями>
  // <обработка полосы>
  // <вычисление общей погрешности вычислений dmax>}
 while ( dmax > eps ); // eps - точность решения
```

Алгоритм 12.8. Параллельный алгоритм, реализующий метод сеток при ленточном разделении данных

Для конкретизации представленных в алгоритме действий введем обозначения:

- **ProcNum** – номер процессора, на котором выполняются описываемые действия,
- **PrevProc, NextProc** – номера соседних процессоров, содержащих предшествующую и следующую полосы,
- **NP** – количество процессоров,
- **M** – количество строк в полосе (без учета продублированных граничных строк),
- **N** – количество внутренних узлов в строке сетки (т.е. всего в строке N+2 узла).

Для нумерации строк полосы будем использовать нумерацию, при которой строки *0* и *M+1* есть продублированные из соседних полос граничные строки, а строки собственной полосы процессора имеют номера от *1* до *M*.

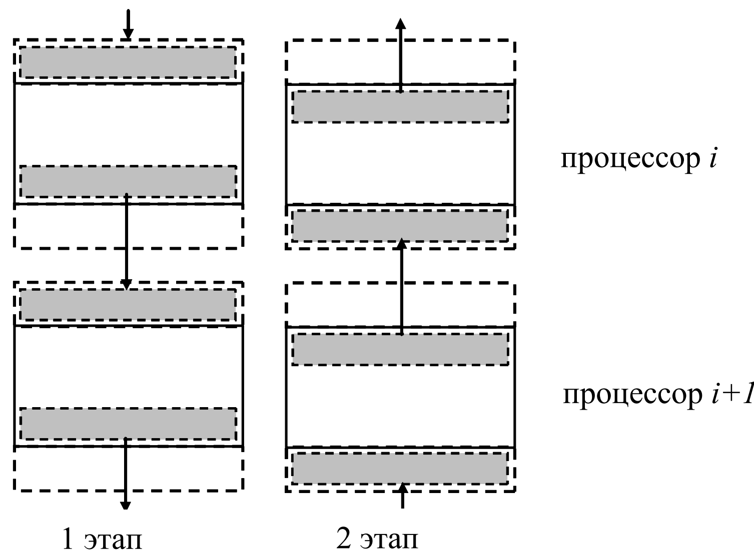

Рис. 12.11. Схема передачи граничных строк между соседними процессорами

Процедура обмена граничных строк между соседними процессорами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора (см. рис. 12.11). Вторая часть передачи строк выполняется в обратном направлении: процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров.

Выполнение подобных операций передачи данных в общем виде может быть представлено следующим образом (для краткости рассмотрим только первую часть процедуры обмена):

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
```

(для записи процедур приема-передачи используется близкий к стандарту MPI (см., например, Group, et al. (1994)) формат, где первый и второй параметры представляют пересылаемые данные и их объем, а третий параметр определяет адресат (для операции *Send*) или источник (для операции *Receive*) пересылки данных).

Для передачи данных могут быть задействованы два различных механизма. При первом из них выполнение программ, инициировавших операцию передачи, приостанавливается до полного завершения всех действий по пересылке данных (т.е. до момента получения процессором-адресатом всех передаваемых ему данных). Операции приема-передачи, реализуемые подобным образом, обычно называются *синхронными* или *блокирующими*. Иной подход – а*синхронная* или *неблокирующая* передача - может состоять в том, что операции приема-передачи только инициируют процесс пересылки и на этом завершают свое выполнение. В результате программы, не дожидаясь завершения длительных коммуникационных операций, могут продолжать свои вычислительные действия, проверяя по мере необходимости готовность передаваемых данных. Оба эти варианта операций передачи широко используются при организации параллельных вычислений и имеют свои достоинства и свои недостатки. Синхронные процедуры передачи, как правило, более просты для использования и более надежны; неблокирующие операции могут позволить совместить процессы передачи данных и вычислений, но обычно приводят к повышению сложности программирования. С учетом всего выше сказанного для организации пересылки данных во всех последующих примерf[будут использоваться операции приема-передачи блокирующего типа.

Приведенная выше последовательность блокирующих операций приема-передачи данных (вначале *Send*, затем *Receive*) приводит к строго последовательной схеме выполнения процесса пересылок строк, т.к. все процессоры одновременно обращаются к операции *Send* и переходят в режим ожидания. Первым процессором, который окажется готовым к приему пересылаемых данных, окажется сервер с номером *NP-1*. В результате процессор *NP-2* выполнит операцию передачи своей граничной строки и перейдет к приему строки от процессора *NP-3* и т.д. Общее количество повторений таких операций равно *NP-1*.

Аналогично происходит выполнение и второй части процедуры пересылки граничных строк перед началом обработки строк (см. рис. 12.11).

Последовательный характер рассмотренных операций пересылок данных определяется выбранным способом очередности выполнения. Изменим этот порядок очередности при помощи чередования приема и передачи для процессоров с четными и нечетными номерами:

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum % 2 == 1 ) { //  нечетный процессор
  if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
  if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
}
else { //  процессор с четным номером
  if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
  if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
}
```

Данный прием позволяет выполнить все необходимые операции передачи всего за два последовательных шага. На первом шаге все процессоры с нечетными номерами отправляют данные, а процессоры с четными номерами осуществляют прием этих данных. На втором шаге роли процессоров меняются – четные процессоры выполняют *Send*, нечетные процессоры исполняют операцию приема *Receive*.

Рассмотренные последовательности операций приема-передачи для взаимодействия соседних процессоров широко используются в практике параллельных вычислений. Как результат, во многих базовых библиотеках параллельных программ имеются процедуры для поддержки подобных действий. Так, в стандарте MPI (см., например, Group, et al. (1994)) предусмотрена операция *Sendrecv*, с использованием которой предыдущий фрагмент программного кода может быть записан более кратко:

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
```

Реализация подобной объединенной функции Sendrecv обычно осуществляется таким образом, чтобы обеспечить и корректную работу на крайних процессорах, когда не нужно выполнять одну из операций передачи или приема, и организацию чередования процедур передачи на процессорах для ухода от тупиковых ситуаций, и возможности параллельного выполнения всех необходимых пересылок данных.

12.3.3. Коллективные операции обмена информацией

Для завершения круга вопросов, связанных с параллельной реализацией метода сеток на системах с распределенной памятью, осталось рассмотреть способы вычисления общей для всех процессоров погрешности вычислений. Возможный очевидный подход состоит в передаче всех локальных оценок погрешности, полученных на отдельных полосах сетки, на один какой-либо процессор, вычисления на нем максимального значения и последующей рассылки полученного значения всем процессорам системы. Однако такая схема является крайне неэффективной – количество необходимых операций передачи данных определяется числом процессоров и выполнение этих операций может происходить только в последовательном режиме. Между тем, как показывает анализ требуемых коммуникационных действий, выполнение операций сборки и рассылки данных может быть реализовано с использованием рассмотренной в п. 2.5 пособия *каскадной схемы* обработки данных. На самом деле, получение максимального значения локальных погрешностей, вычисленных на каждом процессоре, может быть обеспечено, например, путем предварительного нахождения максимальных значений для отдельных пар процессоров (данные вычисления могут выполняться параллельно), затем может быть снова осуществлен попарный поиск максимума среди полученных результатов и т.д. Всего, как полагается по каскадной схеме, необходимо выполнить log_2NP параллельных итераций для получения конечного значения (NP – количество процессоров).

Учитывая большую эффективность каскадной схемы для выполнения коллективных операций передачи данных, большинство базовых библиотек параллельных программ содержит процедуры для поддержки подобных действий. Так, в стандарте MPI (см., например, Group, et al. (1994)) предусмотрены операции:

- **Reduce(dm,dmax,op,proc)** – процедура сборки на процессоре *proc* итогового результата *dmax* среди локальных на каждом процессоре значений *dm* с применением операции *op*,

- **Broadcast(dmax,proc)** – процедура рассылки с процессора *proc* значения *dmax* всем имеющимся процессорам системы.

С учетом перечисленных процедур общая схема вычислений на каждом процессоре может быть представлена в следующем виде:

```
//Алгоритм 12.8 – уточненный вариант
// Параллельный метод Гаусса-Зейделя (распределенная память, ленточная схема)
// действия, выполняемые на каждом процессоре
do {
  // обмен граничных строк полос с соседями
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
  // <обработка полосы с оценкой погрешности dm>
  // вычисление общей погрешности вычислений dmax
  Reduce(dm,dmax,MAX,0);
  Broadcast(dmax,0);
} while ( dmax > eps ); // eps - точность решения
```

(в приведенном алгоритме переменная *dm* представляет локальную погрешность вычислений на отдельном процессоре, параметр *MAX* задает операцию поиска максимального значения для операции сборки). Следует отметить, что в составе MPI имеется процедура *Allreduce*, которая совмещает действия редукции и рассылки данных. Результаты экспериментов для данного варианта параллельных вычислений для метода Гаусса-Зейделя приведены в табл. 12.4.

12.3.4. Организация волны вычислений

Представленные в пп. 1-3 алгоритмы определяют общую схему параллельных вычислений для метода сеток в многопроцессорных системах с распределенной памятью. Далее эта схема может быть конкретизирована реализацией практически всех вариантов методов, рассмотренных для систем с общей памятью (использование дополнительной памяти для схемы Гаусса-Якоби, чередование обработки полос и т.п.). Проработка таких вариантов не привносит каких-либо новых эффектов с точки зрения параллельных вычислений, и их разбор может использоваться как темы заданий для самостоятельных упражнений.

Таблица 12.4. Результаты экспериментов для систем с распределенной памятью, ленточная схема разделения данных (***p***=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 6.1)		Параллельный алгоритм 6.8			Параллельный алгоритм с волновой схемой расчета (см. п. 6.3.4)		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	0,54	0,11	210	1,27	0,05
200	273	0,35	273	0,86	0,41	273	1,37	0,26
300	305	0,92	305	0,92	1,00	305	1,83	0,50
400	318	1,69	318	1,27	1,33	318	2,53	0,67
500	343	2,88	343	1,72	1,68	343	3,26	0,88
600	336	4,04	336	2,16	1,87	336	3,66	1,10
700	344	5,68	344	2,52	2,25	344	4,64	1,22
800	343	7,37	343	3,32	2,22	343	5,65	1,30
900	358	9,94	358	4,12	2,41	358	7,53	1,32
1000	351	11,87	351	4,43	2,68	351	8,10	1,46
2000	367	50,19	367	15,13	3,32	367	27,00	1,86
3000	364	113,17	364	37,96	2,98	364	55,76	2,03

(***k*** – количество итераций, ***T*** – время в сек., ***S*** – ускорение)

В завершение рассмотрим возможность организации параллельных вычислений, при которых обеспечивалось бы нахождение таких же решений задачи Дирихле, что и при использовании исходного последовательного метода Гаусса-Зейделя. Как отмечалось ранее, такой результат может быть получен за счет организации волновой схемы расчетов. Для образования волны вычислений представим

логически каждую полосу узлов области расчетов в виде набора блоков (размер блоков можно положить, в частности, равным ширине полосы) и организуем обработку полос поблочно в последовательном порядке (см. рис. 12.12). Тогда для полного повторения действий последовательного алгоритма вычисления могут быть начаты только для первого блока первой полосы узлов; после того, как этот блок будет обработан, для вычислений будут готовы уже два блока – блок 2 первой полосы и блок 1 второй полосы (для обработки блока полосы 2 необходимо передать граничную строку узлов первого блока полосы 1). После обработки указанных блоков к вычислениям будут готовы уже 3 блока, и мы получаем знакомый уже процесс волновой обработки данных (результаты экспериментов приведены в табл. 12.4).

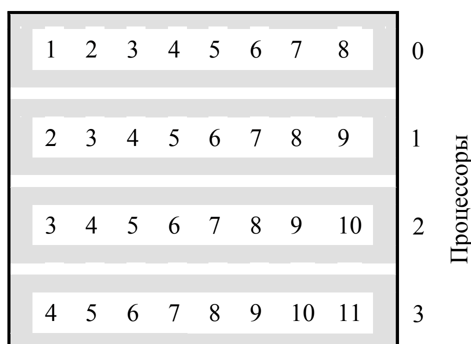

Рис. 12.12. Организация волны вычислений при ленточной схеме разделения данных

Интересный момент при организации подобный схемы параллельных вычислений может состоять в попытке совмещения операций пересылки граничных строк и действий по обработке блоков данных.

12.3.5. Блочная схема разделения данных

Ленточная схема разделения данных может быть естественным образом обобщена на блочный способ представления сетки области расчетов (см. рис. 12.9). При этом столь радикальное изменение способа разбиения сетки практически не потребует каких-либо существенных корректировок рассмотренной схемы параллельных вычислений. Основной новый момент при блочном представлении данных состоит в увеличении количества граничных строк на каждом процессоре (для блока их количество становится равным 4), что приводит, соответственно, к большему числу операций передачи данных при обмене граничных строк. Сравнивая затраты на организацию передачи граничных строк, можно отметить, что при ленточной схеме для каждого процессора выполняется 4 операции приема-передачи данных, в каждой из которых пересылается ($N+2$) значения. Для блочного же способа происходит 8 операций пересылки и объем каждого сообщения равен ($N/\sqrt{NP}+2$) (N – количество внутренних узлов сетки, NP – число процессоров, размер всех блоков предполагается одинаковым). Тем самым, блочная схема представления области расчетов становится оправданной при большом количестве узлов сетки области расчетов, когда увеличение количества коммуникационных операций приводит к снижению затрат на пересылку данных в силу сокращения размеров передаваемых сообщений. Результаты экспериментов при блочной схеме разделения данных приведены в табл. 12.5.

Таблица 12.5. Результаты экспериментов для систем с распределенной памятью, блочная схема разделения данных (***p***=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 6.1)		Параллельный алгоритм с блочной схемой расчета (см. п. 6.3.5)			Параллельный алгоритм 6.9		
	k	***T***	***k***	***T***	***S***	***k***	***T***	***S***
100	210	0,06	210	0,71	0,08	210	0,60	0,10
200	273	0,35	273	0,74	0,47	273	1,06	0,33
300	305	0,92	305	1,04	0,88	305	2,01	0,46
400	318	1,69	318	1,44	1,18	318	2,63	0,64
500	343	2,88	343	1,91	1,51	343	3,60	0,80
600	336	4,04	336	2,39	1,69	336	4,63	0,87
700	344	5,68	344	2,96	1,92	344	5,81	0,98
800	343	7,37	343	3,58	2,06	343	7,65	0,96
900	358	9,94	358	4,50	2,21	358	9,57	1,04

1000	351	11,87	351	4,90	2,42	351	11,16	1,06
2000	367	50,19	367	16,07	3,12	367	39,49	1,27
3000	364	113,17	364	39,25	2,88	364	85,72	1,32

(***k*** – количество итераций, ***T*** – время в сек., ***S*** – ускорение)

При блочном представлении сетки может быть реализован также и волновой метод выполнения расчетов (см. рис. 12.13). Пусть процессоры образуют прямоугольную решетку размером *NBxNB* ($NB = \sqrt{NP}$) и процессоры пронумерованы от *0* слева направо по строкам решетки.

Общая схема параллельных вычислений в этом случае имеет вид:

```
//Алгоритм 12.9
// Параллельный метод Гаусса-Зейделя (распределенная память, блочная схема)
// волновая схема расчетов
// действия, выполняемые на каждом процессоре
do {
  // получение граничных узлов
  if ( ProcNum / NB != 0 ) { // строка не нулевая
    // получение данных от верхнего процессора
    Receive(u[0][*],M+2,TopProc); // верхняя строка
    Receive(dmax,1,TopProc);      // погрешность
  }
  if ( ProcNum % NB != 0 ) { // столбец не нулевой
    // получение данных от левого процессора
    Receive(u[*][0],M+2,LeftProc); // левый столбец
    Receive(dm,1,LeftProc);        // погрешность
    If ( dm > dmax ) dmax = dm;
  }
  // <обработка блока с оценкой погрешности dmax>
  // пересылка граничных узлов
  if ( ProcNum / NB != NB-1 ) { // строка решетки не
                                // последняя
    // пересылка данных нижнему процессору
    Send(u[M+1][*],M+2,DownProc); // нижняя строка
    Send(dmax,1,DownProc);        // погрешность
  }
  if ( ProcNum % NB != NB-1 ) { // столбец решетки
                                // не последний
    // пересылка данных правому процессору
    Send(u[*][M+1],M+2,RightProc); // правый столбец
    Send(dmax,1, RightProc);       // погрешность
  }
  // синхронизация и рассылка погрешности dmax
  Barrier();
  Broadcast(dmax,NP-1);
} while ( dmax > eps ); // eps - точность решения
```

Алгоритм 12.9. Блочная схема разделения данных

(в приведенном алгоритме функция *Barrier*() представляет операцию коллективной синхронизации, которая завершает свое выполнение только в тот момент, когда все процессоры осуществят вызов этой процедуры).

Следует обратить внимание, что при реализации алгоритма нужно обеспечить, чтобы в начальный момент времени все процессоры (кроме процессора с нулевым номером) оказались в состоянии ожидания своих граничных узлов (верхней строки и левого столбца). Вычисления должен начинать процессор с левым верхним блоком, после завершения обработки которого обновленные значения правого столбца и нижней строки блока необходимо переправить правому и нижнему процессорам решетки соответственно. Данные действия обеспечат снятие блокировки процессоров второй диагонали процессорной решётки (ситуация слева на рис. 12.13) и т.д.

Анализ эффективности организации волновых вычислений в системах с распределенной памятью (см. табл. 12.5) показывает значительное снижение полезной вычислительной нагрузки для процессоров, которые занимаются обработкой данных только в моменты, когда их блоки попадают во фронт волны вычислений. При этом балансировка (перераспределение) нагрузки является крайне затруднительной,

поскольку связана с пересылкой между процессорами блоков данных большого объема. Возможный интересный способ улучшения ситуации состоит в организации *множественной волны вычислений*, в соответствии с которой процессоры после отработки волны текущей итерации расчетов могут приступить к выполнению волны следующей итерации метода сеток. Так, например, процессор 0 (см. рис. 12.13), передав после обработки своего блока граничные данные и запустив, тем самым, вычисления на процессорах 1 и 4, оказывается готовым к исполнению следующей итерации метода Гаусса-Зейделя. После обработки блоков первой (процессорах 1 и 4) и второй (процессор 0) волн, к вычислениям окажутся готовыми следующие группы процессоров (для первой волны - процессоры 2, 5 и 8, для второй волны - процессоры 1 и 4). Кроме того, процессор 0 опять окажется готовым к запуску очередной волны обработки данных. После выполнения ***NB*** подобных шагов в обработке будет находиться одновременно ***NB*** итераций и все процессоры окажутся задействованными. Подобная схема организации расчетов позволяет рассматривать имеющуюся процессорную решетку как *вычислительный конвейер* поэтапного выполнения итераций метода сеток. Останов конвейера может осуществляться, как и ранее, по максимальной погрешности вычислений (проверку условия остановки следует начинать только при достижении полной загрузки конвейера после запуска ***NB*** итераций расчетов). Необходимо отметить также, что получаемое после выполнения условия остановки решение задачи Дирихле будет содержать значения узлов сетки от разных итераций метода и не будет, тем самым, совпадать с решением, получаемым при помощи исходного последовательного алгоритма.

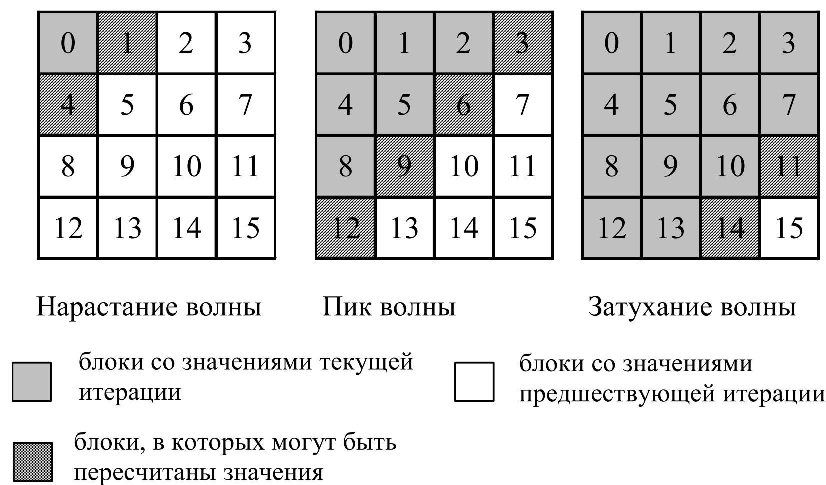

Рис. 12.13. Организация волны вычислений при блочной схеме разделения данных

12.3.6. Оценка трудоемкости операций передачи данных

Время выполнения коммуникационных операций значительно превышает длительность вычислительных команд. Оценка трудоемкости операций приема-передачи может быть осуществлена с использованием двух основных характеристик сети передачи: *латентности* (*latency*), определяющей время подготовки данных к передаче по сети, и *пропускной способности* сети (*bandwidth*), задающей объем передаваемых по сети за 1 сек. данных – более полное изложение вопроса содержится в разделе 3.

Пропускная способность наиболее распространенной на данный момент сети Fast Ethernet – 100 Мбит/с, для более современной сети Gigabit Ethernet – 1000 Мбит/с. В то же время, скорость передачи данных в системах с общей памятью обычно составляет сотни и тысячи миллионов байт в секунду. Тем самым, использование систем с распределенной памятью приводит к снижению скорости передачи данных не менее чем в 100 раз.

Еще хуже дело обстоит с латентностью. Для сети Fast Ethernet эта характеристика имеет значений порядка 150 мкс, для сети Gigabit Ethernet – около 100 мкс. Для современных компьютеров с тактовой частотой свыше 2 ГГц/с различие в производительности достигает не менее, чем 10000-100000 раз. При указанных характеристиках вычислительной системы для достижения 90% эффективности в рассматриваемом примере решения задачи Дирихле (т.е. чтобы в ходе расчетов обработка данных занимала не менее 90% времени от общей длительности вычислений и только 10% времени тратилось бы на операции передачи данных) размер блоков вычислительной сетки должен быть не менее N=7500 узлов по вертикали и горизонтали (объем вычислений в блоке составляет $5N^2$ операций с плавающей запятой).

Как результат, можно заключить, что эффективность параллельных вычислений при использовании распределенной памяти определяется в основном интенсивностью и видом выполняемых коммуникационных операций при взаимодействии процессоров. Необходимый при этом анализ параллельных методов и программ может быть выполнен значительно быстрее за счет выделения типовых операций передачи данных – см. раздел 3. Так, например, в рассматриваемой учебной задаче решения задачи Дирихле практически все пересылки значений сводятся к стандартным коммуникационным действиям, имеющим адекватную поддержку в стандарте MPI (см. рис. 12.14):

- рассылка количества узлов сетки всем процессорам – типовая операция передачи данных от одного процессора всем процессорам сети (функция *MPI_Bcast*);

- рассылка полос или блоков узлов сетки всем процессорам – типовая операция передачи разных данных от одного процессора всем процессорам сети (функция *MPI_Scatter*);

- обмен граничных строк или столбцов сетки между соседними процессорами – типовая операция передачи данных между соседними процессорами сети (функция *MPI_Sendrecv*);

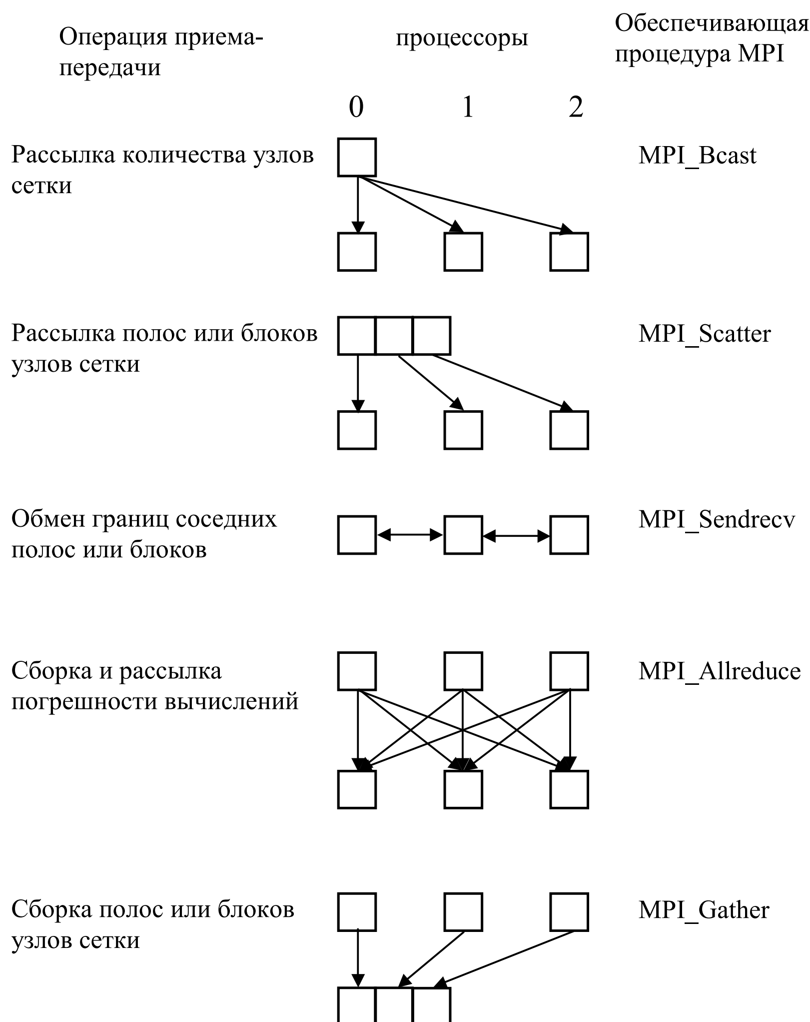

Рис. 12.14. Операции передачи данных при выполнении метода сеток в системе с распределенной памятью

- сборка и рассылка погрешности вычислений всем процессорам – типовая операция передачи данных от всех процессоров всем процессорам сети (функция *MPI_Allreduce*);

- сборка на одном процессоре решения задачи (всех полос или блоков сетки) – типовая операция передачи данных от всех процессоров сети одному процессору (функция *MPI_Gather*).

12.4. Краткий обзор раздела

В разделе рассматриваются параллельные методы для решения дифференциальных уравнений в частных производных. В качестве учебного примера используется проблема численного решения задачи Дирихле для уравнения Пуассона. Наиболее распространенный подход численного решения

дифференциальных уравнений является метод конечных разностей. В разделе рассматриваются возможные способы организации параллельных вычислений для сеточных методов на многопроцессорных вычислительных системах с общей и распределенной памятью.

В подразделе 14.1 дается постановка задачи Дирихле для уравнения Пуассона, приводится описание метода конечных разностей и излагается алгоритм Гаусса-Зейделя для решения поставленной задачи.

В подразделе 14.2 при изложении вопросов организации параллельных вычислений для систем с общей памятью основное внимание уделяется технологиям OpenMP. Приводятся проблемы, возникающие при применении этой технологии, и даются пути решения данных проблем. Среди рассмотренных вопросов – синхронизация параллельных вычислений, способы выявления взаимоблокировок и исключение неоднозначности вычислений с использованием чередования обработки четных и нечетных строк и волновых схем расчетов. В завершении темы рассматривается схема динамической балансировки вычислительной нагрузки процессов при помощи организации очереди заданий.

В подразделе 14.3 обсуждаются вопросы организации параллельных вычислений для систем с распределенной памятью. Основное внимание уделяется проблемам разделения данных и организации обменов информацией между процессорами. Среди способов разделения области расчетов более подробно рассматривается ленточная схема, для блочного представления обрабатываемых данных дается только самая общая характеристика. Обсуждается возможность синхронного и асинхронного выполнения операций передачи данных. Для повышения эффективности параллельных вычислений описывается возможность организации множественной волновой схемы расчетов. В завершении приводятся оценки трудоемкости операций передачи данных.

12.5. Обзор литературы

Дополнительная информация по исследованию проблематики численного решения дифференциальных уравнений в частных производных и по методу конечных разностей может быть получена в Березин и Жидков (1966), Тихонов и Самарский (1977), полезная информация содержится также в Fox, et al. (1988).

Рассмотрение вопроса по организации памяти и использованию кэша приводится в Хамахер и Вранешич (2003).

Информация по вопросам балансировки вычислительной нагрузки между процессорами может быть получена в Xu and Hwang (1998).

12.6. Контрольные вопросы

1. Как определяется задача Дирихле для уравнения Пуассона?
2. Как метод конечных разностей применяется для решения задачи Дирихле?
3. Какие способы определяют организацию параллельных вычислений для сеточных методов на многопроцессорных вычислительных системах с общей памятью?
4. В чем состоит проблема синхронизации параллельных вычислений?
5. Как характеризуется поведение параллельных участков программы, которое именуется состязанием потоков (*race condition*)?
6. В чем состоит проблема взаимоблокировки?
7. Какой метод гарантирует однозначность результатов сеточных методов независимо от способа распараллеливания, но требует использования большого дополнительного объема памяти?
8. Как изменяется объем вычислений при применении методов волновой обработки данных?
9. Что такое фрагментирование (*chunking*)?
10. Как повысить эффективность методов волновой обработки данных?
11. Как очередь заданий позволяет балансировать нагрузку процессорам?
12. Какие проблемы приходится решать при организации параллельных вычислений на системах с распределенной памятью?
13. Какие механизмы могут быть задействованы для передачи данных?
14. Каким образом организация множественной волны вычислений позволяет повысить эффективность волновых вычислений в системах с распределенной памятью?

12.7. Задачи и упражнения

1. Выполните реализацию первого и второго вариантов параллельного алгоритма Гаусса-Зейделя. Сравните время выполнения разработанных программ.
2. Выполните реализации параллельного алгоритма на основе волновой схемы вычислений и параллельного алгоритма, в котором реализуется блочный подход к методу волновой обработки данных. Сравните время выполнения разработанных программ.
3. Выполните реализацию очереди заданий параллельных вычислений для систем с общей памятью. При реализации необходимо обеспечить возможность обработки близких блоков на одних и тех же процессорах.

Литература

Березин И.С., Жидков И.П. (1966). Методы вычислений.-М.:Наука

Гергель, В.П., Стронгин, Р.Г. (2001). Основы параллельных вычислений для многопроцессорных вычислительных систем. - Н.Новгород, ННГУ (2 изд., 2003).

Корнеев В.В. (2003) Параллельное программирование в MPI. Москва-Ижевск: Институт компьютерных исследований,2003

Немнюгин С., Стесик О. (2002). Параллельное программирование для многопроцессорных вычислительных систем – СПб.: БХВ-Петербург.

Таненбаум Э. (2002) . Архитектура компьютера. – СПб.: Питер.

Тихонов А.Н., Самарский А.А. (1977). Уравнения математической физики. – М.: Наука.

Хамахер К., Вранешич З., Заки С. (2003). Организация ЭВМ. –СПб:Питер.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Melon, R. (2000). Parallel Programming in OpenMP. Morgan Kaufmann Publishers.

Fox, G.C. et al.(1988). Solving Problems on Concurrent Processors.- Prentice Hall, Englewood Cliffs, NJ.

Group, W., Lusk, E., Skjellum, A. (1994). Using MPI. Portable Parallel Programming with the Message-Passing Interface. –MIT Press.

Pfister, G. P. (1995). In Search of Clusters. - Prentice Hall PTR, Upper Saddle River, NJ (2nd edn., 1998).

Roosta, S.H. (2000). Parallel Processing and Parallel Algorithms: Theory and Computation. Springer-Verlag,NY.

Tanenbaum, A. (2001). Modern Operating System. 2nd edn. – Prentice Hall (русский перевод Таненбаум Э. Современные операционные системы. – СПб.: Питер, 2002)

Xu, Z., Hwang, K. (1998). Scalable Parallel Computing Technology, Architecture, Programming. – Boston: McGraw-Hill.