ГЛАВА 7
ПАРАЛЛЕЛЬНЫЕ МЕТОДЫ МАТРИЧНОГО УМНОЖЕНИЯ

Операция умножения матриц является одной из основных задач матричных вычислений. В данной главе рассматриваются несколько разных параллельных алгоритмов для выполнения этой операции. Два из них основаны на ленточной схеме разделения данных. Другие два метода используют блочную схему разделения данных, при этом последний из них основывается на разбиении матриц на блоки такого размера, чтобы блоки можно было полностью поместить в кэш-память.

7.1. Постановка задачи

Умножение матрицы A размера $m \times n$ и матрицы B размера $n \times l$ приводит к получению матрицы C размера $m \times l$, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l. \qquad (7.1)$$

Как следует из (7.1), каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

$$c_{ij} = a_i, b_j^T, \; a_i = a_{i0,} a_{i1}, ..., a_{in-1}, \; b_j^T = b_{0j}, b_{1j}, ..., b_{n-1j}{}^T. \qquad (7.2)$$

Этот алгоритм предполагает выполнение $m \cdot n \cdot l$ операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера $n \times n$ количество выполненных операций имеет порядок $O(n^3)$. Известны последовательные алгоритмы умножения матриц, обладающие меньшей вычислительной сложностью (например, алгоритм Страссена (*Strassen's algorithm*) [17]), но эти алгоритмы требуют определенных усилий для их освоения и, как результат, в данной главе при разработке параллельных методов в качестве основы будет использоваться приведенный выше последовательный алгоритм. Также будем предполагать далее, что все матрицы являются квадратными и имеют размер $n \times n$.

7.2. Последовательный алгоритм

7.2.1. Описание алгоритма

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

```
// Программа 7.1
// Последовательный алгоритм умножения матриц
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
  for (j=0; j<Size; j++){
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++){
      MatrixC[i][j] = MatrixC[i][j] +
        MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы C. Действительно, при выполнении одной итерации внешнего цикла (цикла по переменной i) вычисляется одна строка результирующей матрицы (см. рис. 7.1)

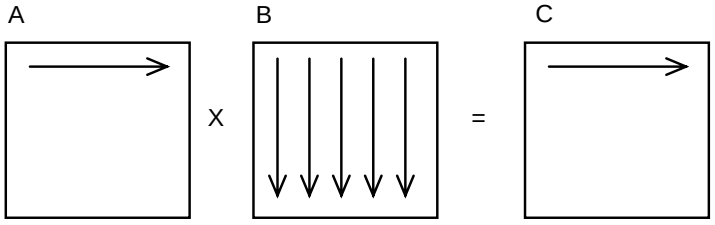

Рис. 7.1. На первой итерации цикла по переменной i используется первая строка матрицы A и все столбцы матрицы B для того, чтобы вычислить элементы первой строки результирующей матрицы C

Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы C размером $n \times n$ необходимо выполнить $n^2(2n\text{-}1)$ скалярных операций и затратить время

$$T_1 = n^2 \cdot \ (2n-1) \ \cdot \tau \qquad (7.3)$$

где τ есть время выполнения одной элементарной скалярной операции.

7.2.2. Анализ эффективности

При анализе эффективности последовательного алгоритма умножения матриц будем опираться на положения, изложенные в п. 6.5.4.

Итак, время выполнения алгоритма складывается из времени, которое тратится непосредственно на вычисления, и времени, необходимого на чтение данных из оперативной памяти в кэш процессора. Время вычислений может быть оценено с использованием формулы (7.3).

Теперь необходимо оценить объем данных, которые необходимо прочитать из оперативной памяти в кэш вычислительного элемента в случае, когда размер матриц настолько велик, что они одновременно не могут быть помещены в кэш. Для вычисления одного элемента результирующей матрицы необходимо прочитать в кэш элементы одной строки матрицы A и одного столбца матрицы B. Для записи полученного результата дополнительно требуется чтение соответствующего элемента матрицы C из оперативной памяти. Важно отметить, что приведенные оценки количества читаемых из памяти данных справедливы, если все эти данные отсутствуют в кэше. В реальности часть этих данных может уже присутствовать в кэше, и тогда объем переписываемых данных в кэш уменьшается.

Расположение данных в каждом конкретном случае зависит от многих величин (размер кэша, объема обрабатываемых данных, стратегии замещения строк кэша и т. п.). Детальный анализ всех этих моментов является достаточно затруднительным. Возможный выход в таком случае состоит в оценке максимально возможного объема данных, перемещаемых из памяти в кэш (построение оценки сверху). В нашем случае всего необходимо вычислить n^2 элементов результирующей матрицы – тогда, предполагая, что при вычислении каждого очередного элемента требуется прочитать в кэш все необходимые данные, следует, что общий объем данных, необходимых для чтения из оперативной памяти в кэш, не превышает величины $2n^3+n^2$.

Таким образом, оценка времени выполнения последовательного алгоритма умножения матриц может быть представлена следующим образом:

$$T_1 = n^2(2n-1)\cdot\tau + 64\cdot(2n^3+n^2)/\beta \qquad (7.4)$$

где β есть пропускная способность канала доступа к оперативной памяти (константа 64 введена для учета факта, что в случае кэш-промаха из ОП читается кэш-строка размером 64 байт).

Если, как и в предыдущей главе, помимо пропускной способности учесть латентность памяти, модель приобретет следующий вид:

$$T_1 = n^2(2n-1)\cdot\tau + (2n^3+n^2)\ \alpha + 64/\beta \qquad (7.5)$$

где α есть латентность оперативной памяти.

Обе предыдущие модели являются моделями на худший случай и дают сильно завышенную оценку времени выполнения алгоритма, так как доступ к оперативной памяти происходит не при каждом обращении к элементу (см. п. 6.5.4).

Как и ранее, введем в модель (7.5) величину γ, $0 \le \gamma \le 1$, для задания *частоты возникновения кэш-промахов*. Тогда оценка времени выполнения алгоритма матричного умножения принимает вид:

$$T_1 = n^2(2n-1)\cdot\tau + \gamma(2n^3+n^2)\ \alpha + 64/\beta \qquad (7.6)$$

7.2.3. Программная реализация

Представим возможный вариант последовательной программы умножения матриц.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 7.2
// Последовательный алгоритм умножения матриц
void main(int argc, char* argv[]) {
  double* pAMatrix;  // Матрица A
  double* pBMatrix;  // Матрица B
  double* pCMatrix;  // Матрица C
  int Size;          // Размер матриц

  // Инициализация данных
  ProcessInitialization(pAMatrix, pBMatrix,
    pCMatrix, Size);

  // Умножение матриц
  SerialResultCalculation(pAMatrix, pBMatrix,
    pCMatrix, Size);

  // Завершение вычислений
```

```
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix,
    Size);
}
```

2. Функция ProcessInitialization. Эта функция определяет размер матриц и элементы для матриц *A* и *B*, результирующая матрица *C* заполняется нулями. Значения элементов для матриц *A* и *B* определяются в функции *RandomDataInitialization.*

```
// Функция выделения памяти и инициализации данных
void ProcessInitialization (double* &pAMatrix,
  double* &pBMatrix, double* &pCMatrix, int &Size)
{
  int i, j;

  do {
    printf("\nВведите размер матриц: ");
    scanf("%d", &Size);
    if (Size <= 0) {
      printf("Размер должен быть больше 0! \n ");
    }
  } while (Size <= 0);

  pAMatrix = new double [Size*Size];
  pBMatrix = new double [Size*Size];
  pCMatrix = new double [Size*Size];
  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++)
      pCMatrix[i*Size+j] = 0;
RandomDataInitialization(pAMatrix, pBMatrix, Size);
}
```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

3. Функция SerialResultCaculation. Данная функция производит умножение матриц.

```
// Функция матричного умножения
void SerialResultCalculation(double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size) {
  int i, j, k;
  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++)
```

```
      for (k=0; k<Size; k++)
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*
          pBMatrix[k*Size+j];
}
```

Следует отметить, что приведенный программный код может быть оптимизирован (вычисление индексов, использование кэша и т. п.), однако такая оптимизация не является целью данного учебного материала и усложняет понимание программ.

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

7.2.4. Результаты вычислительных экспериментов

Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехъядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

Известно, что в современных компиляторах реализованы достаточно сложные алгоритмы оптимизации кода: в некоторых случаях может автоматически выполняться развертка циклов, осуществление предсказаний потребности данных и т. п. Чтобы не учитывать влияние этих средств и рассматривать код «как он есть», функция оптимизации кода компилятором была отключена.

Для того, чтобы оценить влияние оптимизации, производимой компилятором, на эффективность приложения, проведем простой эксперимент. Измерим время выполнения оптимизированной и неоптимизированной версий программы, выполняющей последовательный алгоритм умножения матриц для разных размеров матриц. Результаты проведенных экспериментов представлены в табл. 7.1 и на рис. 7.2.

Таблица 7.1.

Сравнение времени выполнения оптимизированной и неоптимизированной версий последовательного алгоритма умножения матриц

Размер матриц	Компиляторная оптимизация включена	Компиляторная оптимизация выключена
1000,0000	8,2219	24,8192
1500,0000	28,6027	85,8869
2000,0000	75,1572	176,5772

2500,0000	145,2053	403,2405
3000,0000	267,0592	707,1501

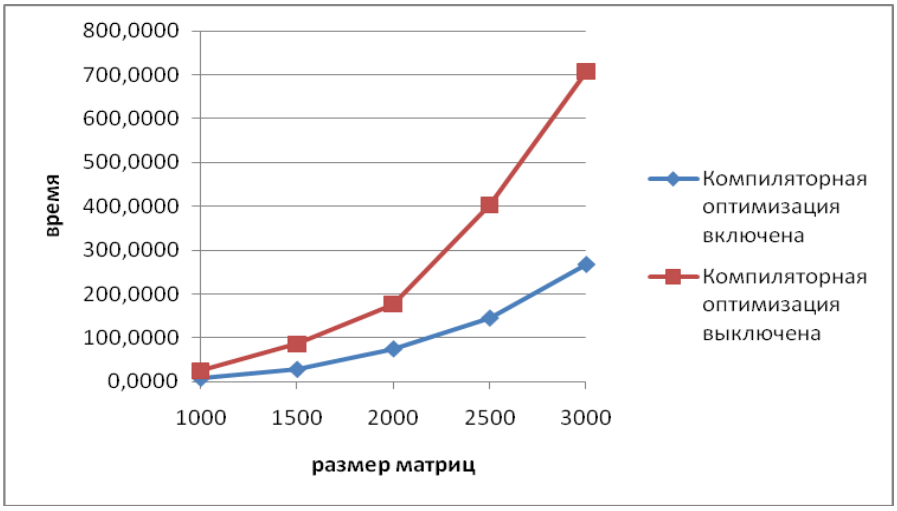

Рис. 7.2. Графики зависимости времени выполнения оптимизированной и неоптимизированной версий последовательного алгоритма

Как видно из представленных графиков, оптимизация кода при помощи компилятора позволяет добиться более чем двукратного ускорения без каких-либо усилий со стороны программиста.

Для того, чтобы оценить время одной операции τ, измерим время выполнения последовательного алгоритма умножения матриц при малых объемах данных, таких, чтобы три матрицы, участвующие в умножении, полностью поместились в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицы-аргументы случайными числами, а матрицу-результат – нулями. Выполнение этих действий гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, т. к. нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение τ, равное 6,402 нс.

Оценка времени латентности α и величины пропускной способности канала доступа к оперативной памяти β проводилась в п. 6.5.4 и

определена для используемого вычислительного узла как β = 12,44 Гб/с и α = 8,31 нс.

Таблица 7.2.

Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма умножения матриц

Размер матриц	Эксперимент	Время счета	Время доступа к памяти	Модель
1000	24,8192	12,7976	13,2390	26,0366
1500	85,8869	43,1991	44,6742	87,8733
2000	176,5772	102,4064	105,8855	208,2919
2500	403,2405	200,0225	206,7973	406,8198
3000	707,1501	345,6504	357,3339	702,9843

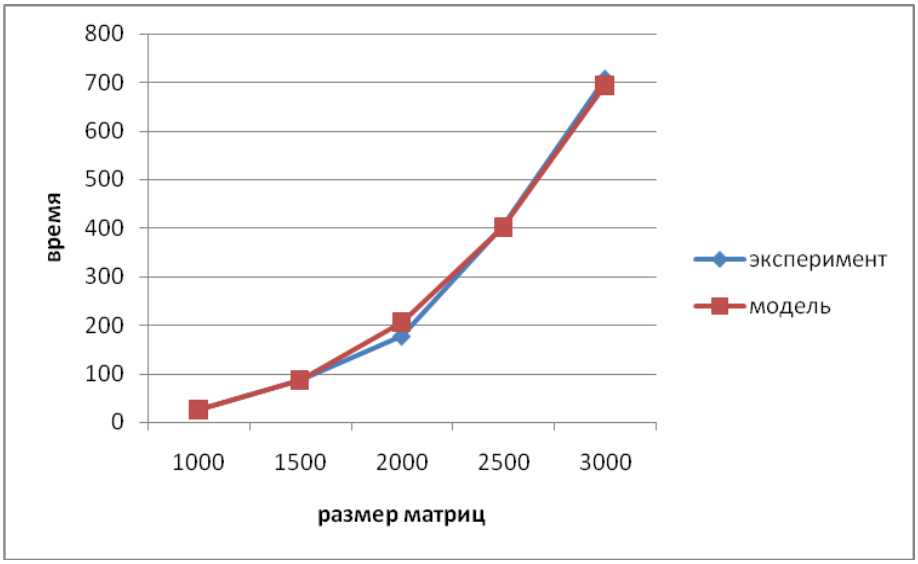

Рис. 7.3. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных

В табл. 7.2 и на рис. 7.3 представлены результаты сравнения времени выполнения последовательного алгоритма умножения матриц со временем, полученным при помощи модели (7.6). Как следует из приведенных данных, погрешность аналитической оценки трудоемкости алгоритма матричного умножения уменьшается при увеличении размера матриц (для n = 3000 относительная погрешность составляет менее 1%). Частота кэш-промахов, измеренная с помощью системы VPS, оказалась равной 0,505.

7.3. Базовый параллельный алгоритм умножения матриц

Рассмотрим параллельный алгоритм умножения матриц, в основу которого будет положено разбиение матрицы A на непрерывные последовательности строк (*горизонтальные полосы*).

7.3.1. Определение подзадач

Из определения операции матричного умножения следует, что вычисление всех элементов матрицы C может быть выполнено независимо друг от друга. Как результат, возможный подход для организации параллельных вычислений состоит в использовании в качестве базовой подзадачи процедуры определения одного элемента результирующей матрицы C. Для проведения всех необходимых вычислений каждая подзадача должна производить вычисления над элементами одной строки матрицы A и одного столбца матрицы B. Общее количество получаемых при таком подходе подзадач оказывается равным n^2 (по числу элементов матрицы C).

Рассмотрев предложенный подход, можно отметить, что достигнутый уровень параллелизма является в некоторой степени избыточным. При проведении практических расчетов количество сформированных подзадач, как правило, будет превышать число имеющихся вычислительных элементов (процессоров и/или ядер) и, тем самым, неизбежным является этап укрупнения базовых задач. В этом плане может оказаться полезным агрегация вычислений уже на шаге выделения базовых подзадач. Возможное решение может состоять в объединении в рамках одной подзадачи всех вычислений, связанных не с одним, а с несколькими элементами результирующей матрицы C. Для дальнейшего рассмотрения в рамках данного раздела определим базовую задачу как процедуру вычисления всех элементов одной из строк матрицы C. Такой подход приводит к снижению общего количества подзадач до величины n.

Для выполнения всех необходимых вычислений базовой подзадаче должны быть доступны одна из строк матрицы A и все столбцы матрицы B. Простое решение этой проблемы – дублирование матрицы B во всех подзадачах. Следует отметить, что такой подход не приводит к реальному дублированию данных, поскольку разрабатываемый алгоритм ориентирован на применение для вычислительных систем с общей разделяемой памятью, к которой имеется доступ из всех используемых вычислительных элементов.

7.3.2. Выделение информационных зависимостей

Для вычисления одной строки матрицы C необходимо, чтобы в каждой подзадаче содержалась строка матрицы A и был обеспечен доступ ко всем столбцам матрицы B. Способ организации параллельных вычислений представлен на рис. 7.4.

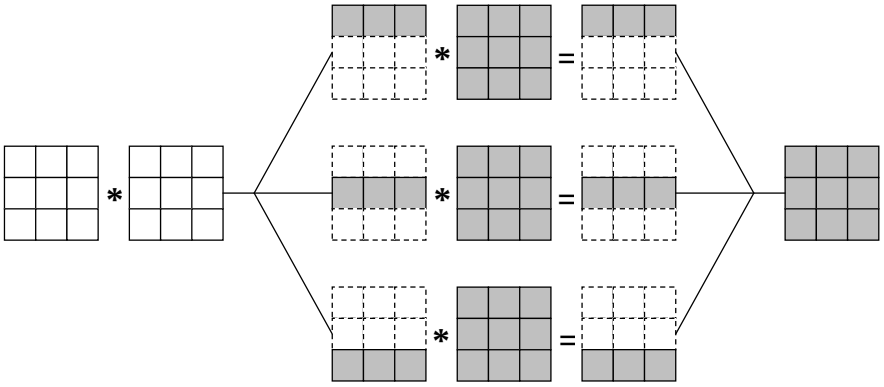

Рис. 7.4. Организация вычислений при выполнении параллельного алгоритма умножения матриц, основанного на разделении матриц по строкам

7.3.3. Масштабирование и распределение подзадач

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае, когда размер n матриц оказывается больше, чем число p вычислительных элементов (процессоров и/или ядер), базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних строк матрицы. В этом случае исходная матрица A и матрица-результат C разбиваются на ряд горизонтальных полос. Размер полос при этом следует выбрать равным $k = n/p$ (в предположении, что n кратно p), что позволит по-прежнему обеспечить равномерность распределения вычислительной нагрузки по вычислительным элементам.

7.3.4. Анализ эффективности

Будем проводить анализ эффективности реализации базового параллельного алгоритма умножения матриц по схеме, изложенной в п. 6.5.4.

Данный параллельный алгоритм обладает хорошей «локальностью вычислений». Это означает, что данные, которые обрабатывает один из потоков параллельной программы, не изменяются другим потоком. Нет

взаимодействия между потоками, нет необходимости в синхронизации. Значит, для того, чтобы определить время выполнения параллельного алгоритма, необходимо знать, сколько вычислительных операций выполняет каждый поток параллельной программы (вычисления выполняются потоками параллельно) и сколько данных необходимо прочитать из оперативной памяти в кэш процессора (доступ к памяти осуществляется строго последовательно).

Для вычисления одного элемента результирующей матрицы необходимо выполнить скалярное умножение строки матрицы A на столбец матрицы B. Выполнение скалярного умножения включает $(2n–1)$ вычислительных операций. Каждый поток вычисляет элементы горизонтальной полосы результирующей матрицы, число элементов в полосе составляет n^2/p. Таким образом, время, которое тратится на вычисления, может быть определено по формуле:

$$T_{calc} = (n^2 / p)(2n-1) \cdot \tau . \qquad (7.7)$$

Для оценки объема данных, которые необходимо прочитать из оперативной памяти в кэш, снова применим подход, изложенный в п. 7.2.3. Для вычисления одного элемента результирующей матрицы C необходимо прочитать в кэш $n+8n+8$ элементов данных. Каждый поток вычисляет n/p элементов матрицы C, однако для определения полного объема переписываемых в кэш данных следует учитывать, что чтение значений из оперативной памяти может выполняться только последовательно (см. п. 6.5.4 и рис. 6.6). Как результат, сокращение объема переписываемых в кэш данных достигается только для матрицы B (прочитанный однократно в кэш столбец матрицы B может использоваться всеми потоками без повторного чтения из оперативной памяти). Чтение же строк матрицы A и элементов матрицы C в предельном случае должно быть выполнено полностью и последовательно. В результате время работы с оперативной памятью при выполнении описанного параллельного алгоритма умножения матриц может быть определено в соответствии со следующим соотношением:

$$T_{mem} = \; n^3 + n^3/p + n^2 \quad \alpha + 64/\beta \;\; , \qquad (7.8)$$

где, как и ранее, β есть пропускная способность канала доступа к оперативной памяти, а α – латентность оперативной памяти. Следовательно, время выполнения параллельного алгоритма составляет:

$$T_p = (n^2 / p) \;\; 2n-1 \;\cdot \tau + \;\; n^3 + n^3/p + n^2 \quad \alpha + 64/\beta \qquad (7.9)$$

Как и ранее, следует учесть, что часть необходимых данных может быть перемещена в кэш заблаговременно при помощи тех или иных

механизмов предсказания. Кроме того, обращение к данным не обязательно приводит к кэш-промаху и, соответственно, к чтению данных из оперативной памяти (необходимые данные могут находиться и в кэш памяти) Данные факторы можно, как и в предыдущих случаях, учесть при помощи введения в модель показатель частоты кэш промахов:

$$T_p = (n^2/p)\ \ 2n-1\ \cdot\tau+\gamma\ \ n^3+n^3/p+n^2\quad \alpha+64/\beta\ . \qquad (7.10)$$

7.3.5. Программная реализация

Для того, чтобы разработать параллельную программу, реализующую описанный подход при помощи технологии OpenMP, необходимо внести минимальные изменения в функцию умножения матриц. Достаточно добавить одну директиву *parallel for* в функции *SerialResultCaclualtion* (назовем новый вариант функции *ParallelResultCaclualtion*):

```
// Программа 7.3
// Функция параллельного матричного умножения
void ParallelResultCalculation(double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size) {
  int i, j, k;
#pragma omp paralell for private (j, k)
  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++)
      for (k=0; k<Size; k++)
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*
          pBMatrix[k*Size+j];
}
```

Данная функция производит умножение строк матрицы A на столбцы матрицы B с использованием нескольких параллельных потоков. Каждый поток выполняет вычисления над несколькими соседними строками матрицы A и, таким образом, получает несколько соседних строк результирующей матрицы C.

7.3.6. Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности параллельного алгоритма матричного умножения. Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехъядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась

в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

Результаты вычислительных экспериментов приведены в табл. 7.3. Времена выполнения алгоритмов указаны в секундах.

Таблица 7.3.

Результаты вычислительных экспериментов для параллельного алгоритма умножения матриц при ленточной схеме разделении данных по строкам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		Время	Ускорение	Время	Ускорение
1000	24,8192	12,3295	2,0130	6,1702	4,0224
1500	85,8869	42,6829	2,0122	21,3086	4,0306
2000	176,5772	87,4123	2,0200	45,4788	3,8826
2500	403,2405	200,8551	2,0076	103,9742	3,8783
3000	707,1501	350,3547	2,0184	176,9224	3,9970

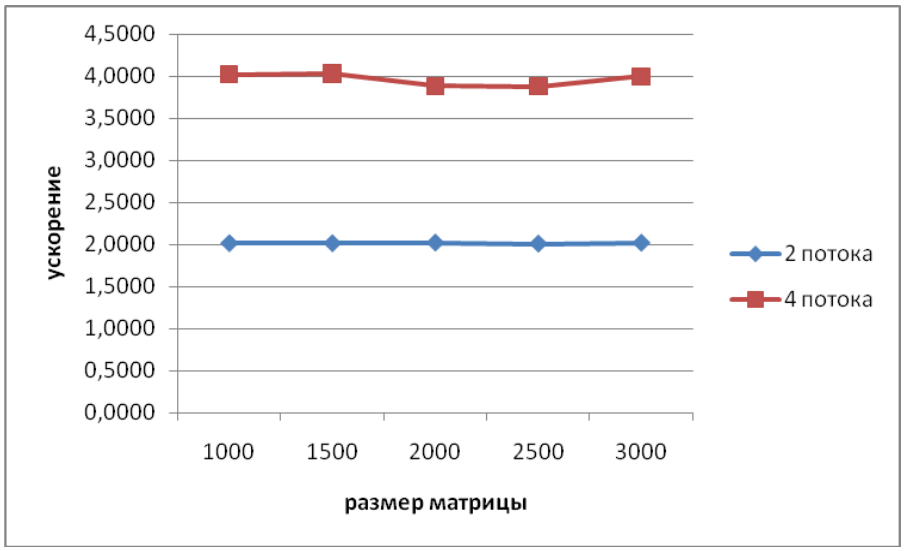

Рис. 7.5. Зависимость ускорения от количества исходных данных при выполнении базового параллельного алгоритма умножения матриц

Можно отметить, что выполненные эксперименты показывают почти идеальное ускорение вычислений для разработанного параллельного алгоритма умножения матриц (и данный результат достигнут в результате незначительной корректировки исходной последовательной программы).

В табл. 7.4, 7.5 и на рис. 7.6, 7.7 представлены результаты сравнения времени выполнения T_p параллельного алгоритма умножения матриц с использованием двух и четырех потоков со временем T_p^*, полученным при помощи модели (7.4). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,3371, а для четырех потоков значение этой величины было оценено как 0,1832.

Таблица 7.4.

Сравнение экспериментального и теоретического времени выполнения базового параллельного алгоритма умножения матриц с использованием двух потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 7.4 – оценка сверху		Модель 7.5 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	12,3295	6,3988	19,6652	26,0640	6,6291	13,0279
1500	42,6829	21,5995	66,3552	87,9547	22,3683	43,9679
2000	87,4123	51,2032	157,2688	208,4720	53,0153	104,2185
2500	200,8551	100,0112	307,1452	407,1565	103,5387	203,5499
3000	350,3547	172,8252	530,7234	703,5486	178,9069	351,7321

Таблица 7.5.

Сравнение экспериментального и теоретического времени выполнения базового параллельного алгоритма умножения матриц с использованием четырех потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 7.4 – оценка сверху		Модель 7.5 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	6,1702	3,1994	16,3898	19,5892	3,0026	6,2020
1500	21,3086	10,7998	55,3009	66,1007	10,1311	20,9309
2000	45,4788	25,6016	131,0661	156,6677	24,0113	49,6129
2500	103,9742	50,0056	255,9680	305,9736	46,8933	96,8990
3000	176,9224	86,4126	442,2892	528,7018	81,0274	167,4400

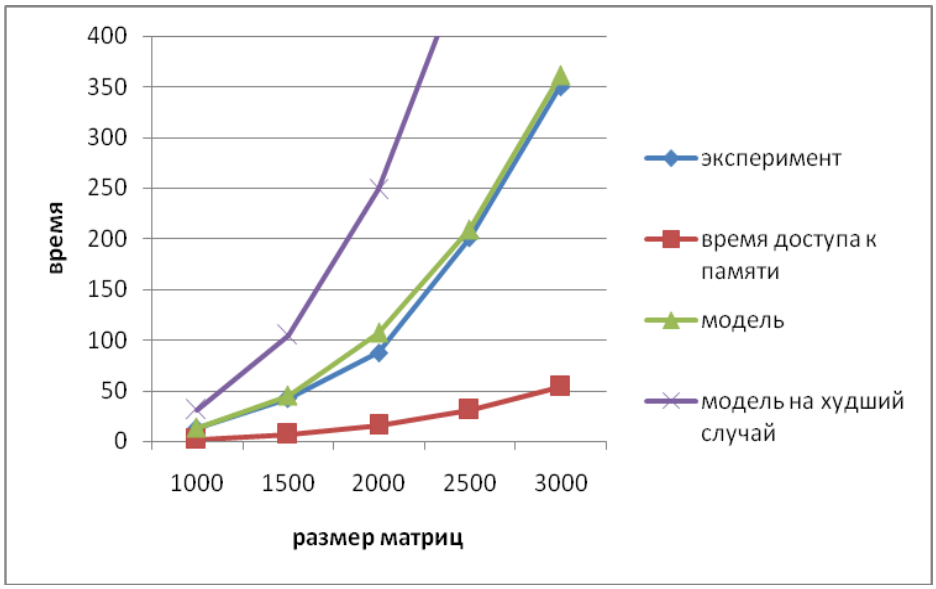

Рис. 7.6. График зависимости экспериментального и теоретического времен выполнения базового параллельного алгоритма от объема исходных данных при использовании двух потоков

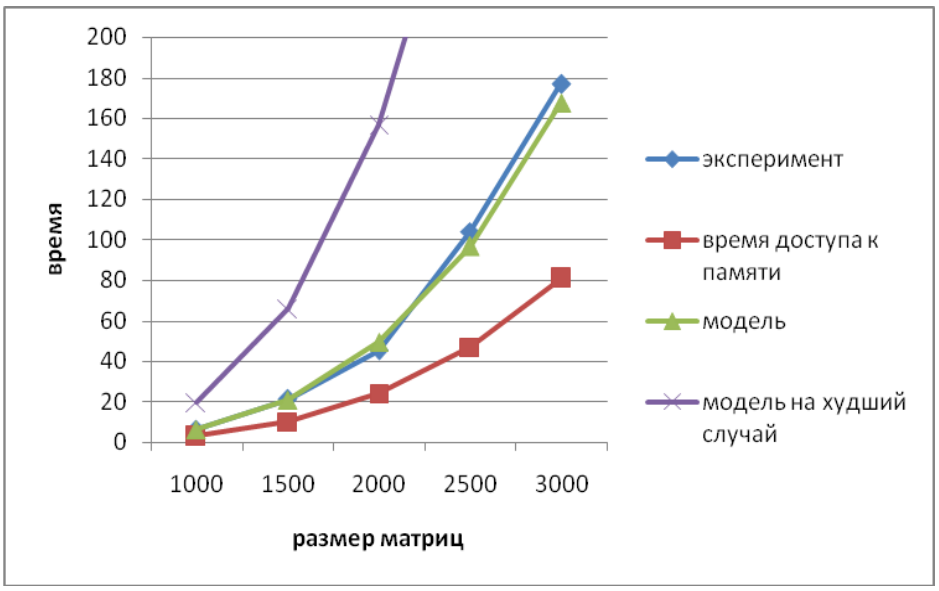

Рис. 7.7. График зависимости экспериментального и теоретического времени выполнения базового параллельного алгоритма от объема исходных данных при использовании четырех потоков

7.4. Алгоритм умножения матриц, основанный на ленточном разделении данных

В рассмотренном в разделе 7.3 только одна из перемножаемых матриц – матрица A – распределялась между параллельно выполняемыми потоками. В данном разделе излагается алгоритм, в котором ленточная схема разделения данных применяется и для второй перемножаемой матрицы B – такой подход, в частности, позволяет улучшить локализацию данных в потоках и повысить эффективность использования кэша.

7.4.1. Определение подзадач

Как и ранее, в качестве базовой подзадачи будем рассматривать процедуру определения одного элемента результирующей матрицы C. Общее количество получаемых при таком подходе подзадач оказывается равным n^2 (по числу элементов матрицы C). Выше было отмечено, что достигнутый таким образом уровень параллелизма зачастую является избыточным – количество базовых подзадач существенно превышает число доступных вычислительных элементов. В этом случае необходимо выполнить укрупнение подзадач. В рамках данного подраздела определим базовую подзадачу как процедуру вычисления всех элементов прямоугольного блока матрицы C (блочная схема разбиения матриц подробно рассмотрена в разделе 6.2).

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем полагать, что число блоков в матрице C по горизонтали и по вертикали совпадает. Для эффективного выполнения параллельного алгоритма умножения матриц целесообразно выделить число параллельных потоков, совпадающим с количеством блоков матрицы C, т. е. такое количество потоков, которое является полным квадратом q^2 ($\pi = q^2$). Дополнительно можно отметить заранее, что для эффективного выполнения вычислений количество потоков π должно быть, по крайней мере, кратным числу вычислительных элементов (процессоров и/или ядер) p.

7.4.2. Выделение информационных зависимостей

Для вычисления одного элемента c_{ij} результирующей матрицы необходимо выполнить скалярное умножение i-й строки матрицы A и j-го столбца матрицы B. Следовательно, для вычисления всех элементов прямоугольного блока результирующей матрицы

$$C_{i_1-i_2,j_1-j_2} = \{c_{ij} : i_1 \le i \le i_2, j_1 \le j \le j_2\}$$

необходимо выполнить скалярное умножение строк матрицы A с индексами i ($i_1 \le i \le i_2$) на столбцы матрицы B с индексами j ($j_1 \le j \le j_2$). То есть необходимо разделить между потоками параллельной программы как строки матрицы A, так и столбцы матрицы B. Для этого воспользуемся механизмом вложенного параллелизма, который был подробно рассмотрен в п. 6.7.2.

Пусть каждое новое объявление параллельной секции разделяет поток выполнения на q потоков. В этом случае разделение итераций внешнего цикла матричного умножения между потоками параллельной программы разделит матрицу A на q горизонтальных полос. При последующем разделении итераций внутреннего цикла с помощью механизма вложенного параллелизма матрица B окажется разделенной на q вертикальных полос (рис. 7.8).

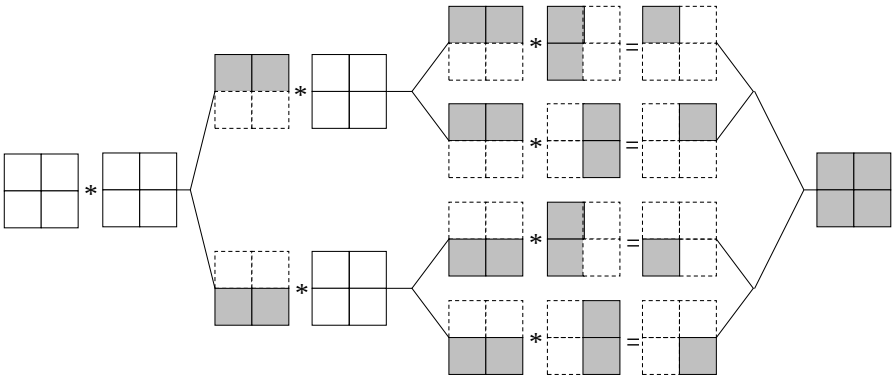

Рис. 7.8. Организация параллельных вычислений при выполнении параллельного алгоритма умножения матриц, основанного на ленточном разделении данных, и использованием четырех потоков

После выполнения вычислений над определенными полосами каждый поток параллельной программы вычислит все элементы блока результирующей матрицы.

7.4.3. Масштабирование и распределение подзадач

Размер блоков матрицы C может быть подобран таким образом, чтобы общее количество базовых подзадач совпадало с числом выделенных потоков π. Если, например, определить размер блочной решетки матрицы C как $\pi = q \cdot q$, то

$$k = m/q,\ l = n/q,$$

где k и l есть количество строк и столбцов в блоках матрицы C. Такой способ определения размера блоков приводит к тому, что объем вычислений в каждой подзадаче является равным и, тем самым, достигается полная балансировка вычислительной нагрузки между вычислительными элементами.

7.4.4. Анализ эффективности

Пусть для параллельного выполнения операции матричного умножения используется π параллельных потоков ($\pi = q \cdot q$). Каждый поток вычисляет элементы прямоугольного блока результирующей матрицы, для вычисления каждого элемента необходимо выполнить скалярное произведение строки матрицы A на столбец матрицы B. Следовательно, количество операций, которые выполняет каждый поток, составляет $n^2 \cdot (2n-1)/\pi$.

Для выполнения программы используется p вычислительных элементов. Если потоки параллельной программы могут быть равномерно распределены по вычислительным элементам, то время выполнения вычислений составляет:

$$T_{calc} = (n^2 / p)(2n-1) \cdot \tau \ .$$

Далее оценим объем данных, которые необходимо прочитать из оперативной памяти в кэш процессора. Каждый поток выполняет умножение горизонтальной полосы матрицы A на вертикальную полосу матрицы B для того, чтобы получить прямоугольный блок результирующей матрицы. Как и ранее, для вычисления одного элемента результирующей матрицы C необходимо прочитать в кэш $2n + 2$ элементов данных. Каждый поток вычисляет n^2/q^2 элементов матрицы C, однако для определения полного объема переписываемых в кэш данных следует учитывать, что чтение значений из оперативной памяти может выполняться только последовательно (см. п. 6.5.4 и рис. 6.6). Элементы матриц A и C, обрабатываемые в разных потоках, не пересекаются и, в предельном случае, должны читаться в кэш для каждой итерации алгоритма повторно (т. е. n^2 раз). С другой стороны, если кэш память является общей для потоков, то столбцы матрицы B могут быть использованы без повторного чтения из оперативной памяти (каждый столбец матрицы B обрабатывается q потоками, отвечающими за вычисление одного и того же вертикального ряда блоков матрицы C). Таким образом, время, необходимое на чтение необходимых данных из оперативной памяти составляет:

$$T_{mem} = n^3 + n^3/q + n^2 \quad \alpha + 64/\beta \ ,$$

где β есть пропускная способность канала доступа к оперативной памяти, а α – латентность оперативной памяти.

Следует обратить внимание на то, что при выполнении представленного алгоритма, реализованного с помощью вложенного параллелизма, «внутренние» параллельные секции создаются и закрываются *n/q* раз. На выполнение функций библиотеки OpenMP, поддерживающих вложенный параллелизм, тратится дополнительное время. Кроме того, поскольку для работы этих функций необходимо читать в кэш служебные данные, «полезные» данные будут вытесняться, а затем повторно загружаться в кэш, что также ведет к росту накладных расходов. Как и ранее, величину накладных расходов на организацию и закрытие одной параллельной секции обозначим через δ.

Таким образом, оценка времени выполнения параллельного алгоритма матричного умножения в худшем случае может быть определена следующим образом:

$$T_p = (n^2/p)\cdot(2n-1)\tau + n^3 + n^3/q + n^2 \quad \alpha + 64/\beta + (n/q)\cdot\delta \qquad (7.11)$$

Если же учесть частоту кэш промахов γ, то выражение (7.11) принимает вид:

$$T_p = (n^2/p)\cdot(2n-1)\tau + \gamma \ n^3 + n^3/q + n^2 \quad \alpha + 64/\beta + (n/q)\cdot\delta \qquad (7.12)$$

7.4.5. Программная реализация

Использование механизма вложенного параллелизма OpenMP позволяет существенно упростить реализацию алгоритма. Однако следует отметить, что на данный момент не все компиляторы, реализующие стандарт OpenMP, поддерживают вложенный параллелизм. Для компиляции представленного ниже кода использовался компилятор Intel C++ Compiler 10.0 for Windows, поддерживающий вложенный параллелизм.

Для того, чтобы разработать параллельную программу, реализующую описанный подход, при помощи технологии OpenMP, прежде всего необходимо включить поддержку вложенного параллелизма при помощи вызова функции *omp_set_nested*.

Для разделения матрицы *A* на горизонтальные полосы нужно разделить итерации внешнего цикла при помощи директивы *omp parallel for*. Далее при помощи той же директивы необходимо разделить итера-

ции внутреннего цикла для разделения матрицы B на вертикальные полосы.

```
// Программа 7.4
// Функция параллельного матричного умножения
void ParallelResultCalculation(double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size) {
  int i, j, k;
  int NestedThreadsNum = 2;
  omp_set_nested(true);
  omp_set_num_threads (NestedThreadsNum);
#pragma omp parallel for private (j, k)
  for (i=0; i<Size; i++)
#pragma omp parallel for private (k)
    for (j=0; j<Size; j++)
      for (k=0; k<Size; k++)
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*
          pBMatrix[k*Size+j];
}
```

Данная функция производит умножение строк матрицы A на столбцы матрицы B с использованием нескольких параллельных потоков. Каждый поток выполняет вычисления над элементами горизонтальной полосы матрицы A и элементами вертикальной полосы матрицы B, таким образом, получает значения элементов прямоугольного блока результирующей матрицы C.

Отметим, что в приведенной программе для задания количество потоков, создаваемых на каждом уровне вложенности параллельных областей, используется переменная *NestedThreadsNum* (в данном варианте программы ее значение устанавливается равным 2 – данное значение должно переустанавливаться для определения необходимого числа потоков).

7.4.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при ленточном разделении данных проводились при условиях, указанных в п. 7.3.6. Результаты вычислительных экспериментов приведены в табл. 7.6. Времена выполнения алгоритма указаны в секундах.

Таблица 7.6.

Результаты вычислительных экспериментов для параллельного алгоритма умножения матриц при ленточной схеме разделения данных

Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	24,82	5,6766	4,37
1500	85,89	20,4516	4,20
2000	176,58	44,3016	3,99
2500	403,24	98,5983	4,09
3000	707,15	173,3591	4,08

Рис. 7.9. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма умножения матриц, основанного на ленточном разделении матриц

Чтобы оценить величину накладных расходов на организацию и закрытие параллельных секций на каждой итерации внешнего цикла, разработаем еще одну реализацию параллельного алгоритма умножения матриц, также основанного на разделении матриц на полосы. Теперь реализуем это разделение явным образом без использования механизма вложенного параллелизма.

```
// Программа 7.5
// Функция параллельного матричного умножения
void ParallelResultCalculation (double* pAMatrix,
```

```
  double* pBMatrix, double* pCMatrix, int Size) {
  int BlocksNum = 2;
  omp_set_num_threads(BlocksNum*BlocksNum);
#pragma omp parallel
  {
    int ThreadID = omp_get_thread_num();
    int RowIndex = ThreadID/BlocksNum;
    int ColIndex = ThreadID%BlocksNum;
    int BlockSize = Size/BlocksNum;
    for (int i=0; i<BlockSize; i++)
      for (int j=0; j<BlockSize; j++)
        for (int k=0; k<Size; k++)
         pCMatrix[(RowIndex*BlockSize+i)*Size +
          (ColIndex*BlockSize+j)] +=
          pAMatrix[(RowIndex*BlockSize+i)*Size+k] *
          pBMatrix[k*Size+(ColIndex*BlockSize+j)];
  }
}
```

Как и ранее в главе 6, время δ, необходимое на организацию и закрытие параллельной секции, принято равным 0,25 мкс. В табл. 7.7 и на рис. 7.10 представлены результаты сравнения времени выполнения T_p параллельного алгоритма умножения матриц с использованием четырех потоков со временем T_p^*, полученным при помощи модели (7.12). Частота кэш-промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,2.

Таблица 7.7.

Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матриц, основанного на ленточном разделении данных, с использованием четырех потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 7.11 – оценка сверху		Модель 7.12 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	5,6766	3,1995	16,3898	19,5893	3,2780	6,4774
1500	20,4516	10,7999	55,3009	66,1008	11,0602	21,8600
2000	44,3016	25,6017	131,0661	156,6678	26,2132	51,8149
2500	98,5983	50,0058	255,9680	305,9738	51,1936	101,1994
3000	173,3591	86,4128	442,2892	528,7019	88,4578	174,8706

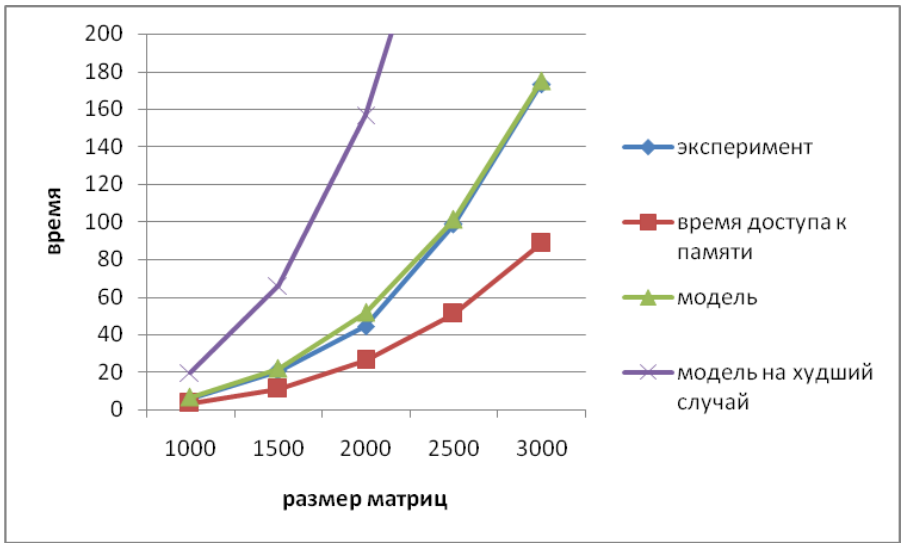

Рис. 7.10. График зависимости экспериментального и теоретического времен выполнения параллельного алгоритма, основанного на ленточном разделении данных, от объема исходных данных при использовании четырех потоков

7.5. Блочный алгоритм умножения матриц

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Рассмотрим более подробно данный способ организации вычислений. В этом случае не только результирующая матрица, но и матрицы-аргументы матричного умножения разделяются между потоками параллельной программы на прямоугольные блоки. Такой подход позволяет добиться большей локализации данных и повысить эффективность использования кэш памяти.

7.5.1. Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в разделе 6.2. При таком способе разделения данных исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера $n \times n$, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т. е. размер всех блоков равен $k \times k$, $k = n/q$). При таком

представлении данных операция матричного умножения матриц A и B в блочном виде может быть представлена в виде:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0q-1} \\ \dots & \dots & \dots & \dots \\ A_{q-10} & A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0q-1} \\ \dots & \dots & \dots & \dots \\ B_{q-10} & B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix} =$$

$$= \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0q-1} \\ \dots & \dots & \dots & \dots \\ C_{q-10} & C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix}, \qquad (7.13)$$

где каждый блок C_{ij} матрицы C определяется в соответствии с выражением

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj} . \qquad (7.14)$$

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы C.

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы A и столбцов матрицы B.

Широко известны параллельные алгоритмы умножения матриц, основанные на блочном разделении данных, ориентированные на многопроцессорные вычислительные системы с распределенной памятью [10]. При разработке алгоритмов, ориентированных на использование параллельных вычислительных систем с распределенной памятью, следует учитывать, что размещение всех требуемых данных в каждой подзадаче (в данном случае – размещение в подзадачах необходимых наборов строк матрицы A и столбцов матрицы B) неизбежно приведет к дублированию и к значительному росту объема используемой памяти. Как результат, вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы при помощи передачи сообщений. К числу алгоритмов, реализующих описанный подход, относятся *алгоритм Фокса* (*Fox*) и *алгоритм Кэннона* (*Cannon*). Отличие этих алгоритмов состоит в последовательности передачи матричных блоков между процессорами вычислительной системы.

При выполнении параллельных алгоритмов на системах с общей памятью передача данных между процессорами уже не требуется. Различия между параллельными алгоритмами в этом случае состоят в порядке организации вычислений над матричными блоками, удовлетворяющие соотношению (7.11). Возможная естественным образом определяемая вычислительная схема рассматривается в п. 7.5.2.

7.5.2. Выделение информационных зависимостей

За основу параллельных вычислений для матричного умножения при блочном разделении данных принимается подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом подзадачи на каждой итерации расчетов обрабатывают только по одному блоку исходных матриц A и B. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C, т. е. подзадача (i,j) отвечает за вычисление блока C_{ij} – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

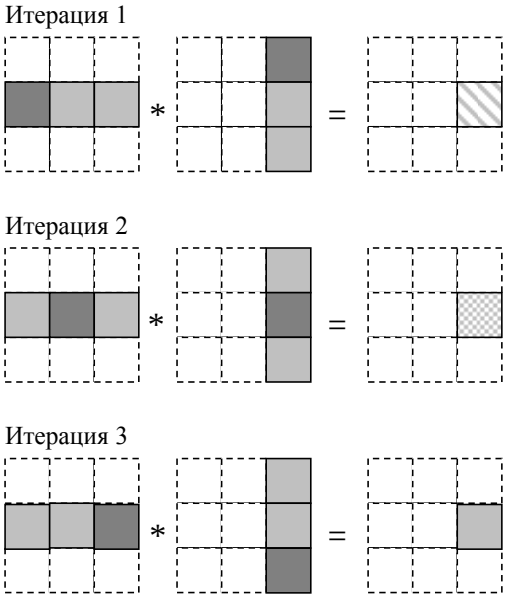

Рис. 7.11. Схема организации блочного умножения полос

Как уже отмечалось выше, для вычисления блока результирующей матрицы поток должен выполнить умножение горизонтальной полосы матрицы A на вертикальную полосу матрицы B. Организуем поблочное умножение полос. На каждой итерации i алгоритма i-й блок полосы

матрицы A умножается на i-й блок полосы матрицы B; результат умножения блоков прибавляется к блоку результирующей матрицы (рис. 7.11). Количество итераций определяется размером блочной решетки.

7.5.3. Масштабирование и распределение подзадач

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом вычислительных элементов p. Так, в наиболее простом случае, когда число вычислительных элементов представимо в виде $p = \sigma^2$ (т. е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали, равным σ (т. е. $q = \sigma$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и, тем самым, достигается полная балансировка вычислительной нагрузки между вычислительными элементами. В случае, когда число вычислительных элементов не является полным квадратом, число базовых подзадач $\pi = q \cdot q$ должно быть по крайней мере кратно числу вычислительных элементов.

7.5.4. Анализ эффективности

При анализе эффективности параллельного алгоритма умножения матриц, основанного на разделении данных на блоки, можно использовать оценки, полученные при анализе предыдущего алгоритма. Действительно, каждый поток выполняет умножение полос матриц-аргументов. Блочное разбиение полос вносит изменение лишь в порядок выполнения вычислений, общий же объем вычислительных операций при этом не изменяется. Поэтому вычислительная сложность блочного алгоритма составляет:

$$T_{calc} = (n^2 / p)(2n-1) \cdot \tau .$$

Оценим объем данных, которые необходимо прочитать из оперативной памяти в кэш процессора. На каждой итерации алгоритма каждый поток выполняет умножение двух матричных блоков размером $(n/q)\times(n/q)$. Для оценки объема данных, который при выполнении этой операции должен быть прочитан из оперативной памяти в кэш, можно воспользоваться формулой (7.5), полученной при анализе эффективности последовательного алгоритма, с поправкой на размер матричного блока.

$$T^1_{mem} = \quad 2 \cdot \ n/q \ ^3 + \ n/q \ ^2 \quad \alpha + 64/\beta$$

Поскольку для вычисления блока результирующей матрицы каждый поток выполняет q итераций, то для определения времени, которое каждый поток тратит на чтение необходимых данных в кэш, необходимо умножить величину T_{mem}^1 на число итераций q. Обращение нескольких потоков к памяти происходит строго последовательно. Общее число потоков – q^2. Таким образом, время считывания данных из оперативной памяти в кэш для блочного алгоритма умножения матриц составляет:

$$T_{mem} = 2 \cdot n/q^3 + n/q^2 \alpha + 64/\beta \cdot q \cdot q^2.$$

Однако при более подробном анализе алгоритма можно заметить, что блок, который считывается в кэш одним из потоков, одновременно используется и другими $(q - 1)$ потоками. Так, например, при использовании решетки потоков 2×2, при выполнении первой итерации алгоритма блок A_{11} используется одновременно нулевым и первым потоком, блок A_{21} – одновременно вторым и третьим потоком, блок B_{11} – нулевым и вторым, а блок B_{12} – первым и третьим потоками. Следовательно, время, необходимое на чтение данных из оперативной памяти в кэш составляет:

$$T_{mem} = 2 \cdot n/q^3 + n/q^2 \alpha + 64/\beta \cdot q \cdot q = 2 \cdot n^3/q + n^2 \alpha + 64/\beta .$$

Как итог, общее время выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении данных, в худшем случае определяется соотношением:

$$T_p = (n^2/p)(2n-1) \cdot \tau + 2 \cdot n^3/q + n^2 \alpha + 64/\beta . \quad (7.15)$$

Если же учесть частоту кэш-промахов, то выражение (7.15) принимает вид:

$$T_p = (n^2/p)(2n-1) \cdot \tau + \gamma \; 2 \cdot n^3/q + n^2 \alpha + 64/\beta \quad (7.16)$$

7.5.5. Программная реализация

Согласно вычислительной схеме блочного алгоритма умножения матриц, описанной в п. 7.5.2, на каждой итерации алгоритма каждый поток параллельной программы выполняет вычисления над матричными блоками. Номер блока, который должен обрабатываться потоком в данный момент, вычисляется на основании положения потока в «решетке потоков» и номера текущей итерации.

Для определения числа потоков, которые будут использоваться при выполнении операции матричного умножения, введем переменную *ThreadNum*. Установим число потоков в значение *ThreadNum* при помощи функции *omp_set_num_threads*. Как следует из описания параллельного алгоритма, число потоков должно являться полным квадратом, для того, чтобы потоки можно было представить в виде двумерной квадратной решетки. Определим размер «решетки потоков» *GridSize* и размер матричного блока *BlockSize*.

Для определения идентификатора потока воспользуемся функцией *omp_get_thread_num*, сохраним результат в переменной *ThreadID*. Чтобы определить положение потока в решетке потоков, введем переменные *RowIndex* и *ColIndex*. Номер строки потоков, в котором расположен данный поток, есть результат целочисленного деления идентификатора потока на размер решетки потоков. Номер столбца, в котором расположен поток, есть остаток от деления идентификатора потока на размер решетки потоков.

Для выполнения операции матричного умножения каждый поток вычисляет элементы блока результирующей матрицы. Для этого каждый поток должен выполнить *GridSize* итераций алгоритма, каждая такая итерация есть умножение матричных блоков; итерации выполняются внешним циклом *for* по переменной *iter*.

```
// Программа 7.6
// Функция параллельного матричного умножения
void ParallelResultCalculation (double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size) {
  int ThreadNum = 4;
  int GridSize  = int (sqrt((double)ThreadNum));
  int BlockSize = Size/GridSize;
  omp_set_num_threads(ThreadNum);
#pragma omp parallel
  {
    int ThreadID = omp_get_thread_num();
    int RowIndex = ThreadID/GridSize;
    int ColIndex = ThreadID%GridSize;
    for (int iter=0; iter<GridSize; iter++) {
      for (int i=RowIndex*BlockSize;
           i<(RowIndex+1)*BlockSize; i++)
        for (int j=ColIndex*BlockSize;
             j<(ColIndex+1)*BlockSize; j++)
          for (int k=iter*BlockSize;
               k<(iter+1)*BlockSize; k++)
            pCMatrix[i*Size+j] +=
          pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
```

```
    }
  } // pragma omp parallel
}
```

7.5.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при блочном разделении данных проводились при условиях, указанных в п. 7.3.6 – их результаты приведены в табл. 7.8. Времена выполнения алгоритма указаны в секундах.

Таблица 7.8.

Результаты вычислительных экспериментов для параллельного блочного алгоритма умножения матриц

Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	24,82	4,40	5,64
1500	85,89	19,01	4,52
2000	176,58	44,90	3,93
2500	403,24	100,93	4,00
3000	707,15	172,88	4,09

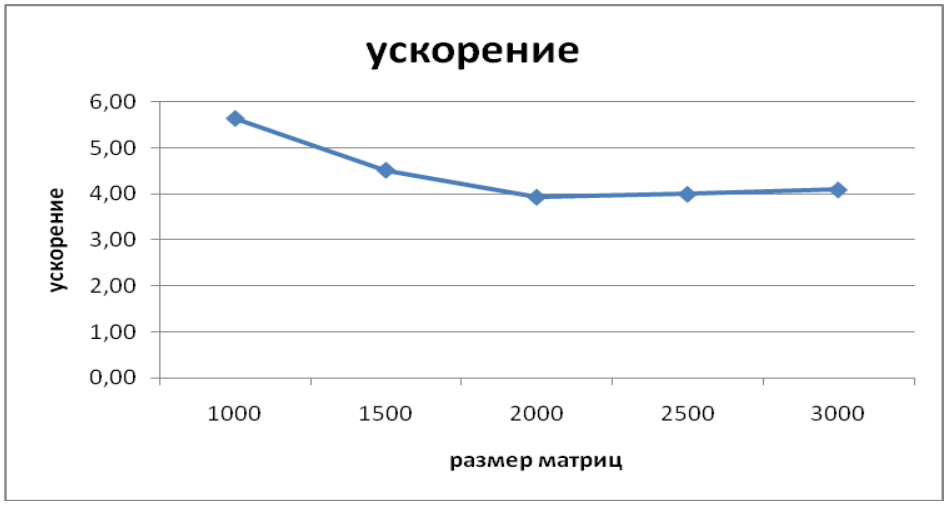

Рис. 7.12. Зависимость ускорения от количества исходных данных при выполнении параллельного блочного алгоритма умножения матриц

В табл. 7.9 и на рис. 7.13 представлены результаты сравнения времени выполнения T_p параллельного алгоритма умножения матриц с

использованием четырех потоков со временем T_p^*, полученным при помощи модели (7.16). Частота кэш-промахов, измеренная с помощью системы VPS для четырех потоков, была оценена как 0,24.

Таблица 7.9.

Сравнение экспериментального и теоретического времен выполнения параллельного блочного алгоритма умножения матриц с использованием четырех потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 7.15 – оценка сверху		Модель 7.16 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	4,3990	3,1994	13,1145	16,3139	3,1475	6,3469
1500	19,0077	10,7998	44,2466	55,0464	10,6192	21,4190
2000	44,8985	25,6016	104,8634	130,4650	25,1672	50,7688
2500	100,9302	50,0056	204,7908	254,7964	49,1498	99,1554
3000	172,8754	86,4126	353,8549	440,2675	84,9252	171,3378

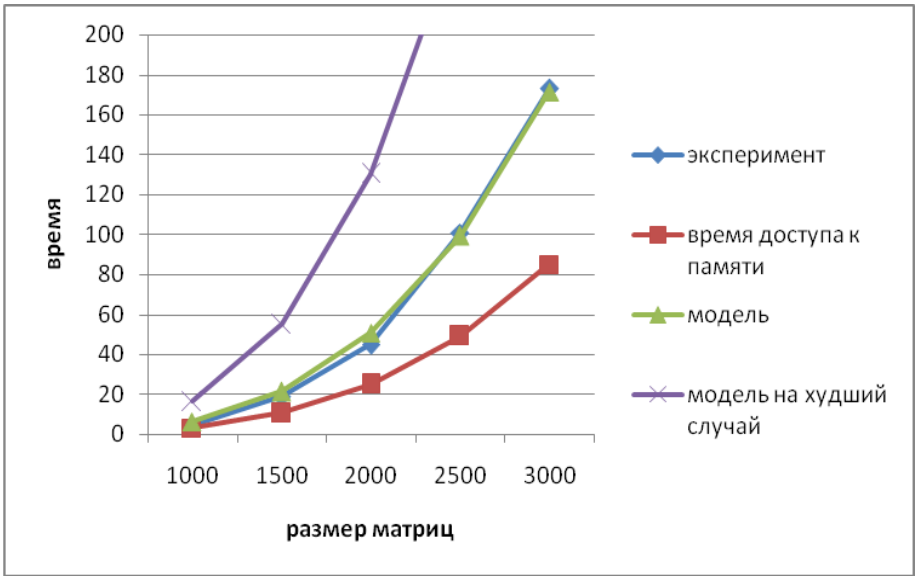

Рис. 7.13. График зависимости экспериментального и теоретического времен выполнения параллельного блочного алгоритма от объема исходных данных при использовании четырех потоков

7.6. Блочный алгоритм, эффективно использующий кэш-память

Как видно из результатов анализа эффективности рассмотренных параллельных алгоритмов, значительная доля времени умножения матриц тратится на многократное чтение элементов матриц A и B из оперативной памяти в кэш. Действительно, при вычислении элементов одной строки результирующей матрицы используются элементы одной строки матрицы A и все элементы матрицы B (рис. 7.1). Для вычисления первого элемента строки результирующей матрицы необходимо прочитать в кэш все элементы первой строки матрицы A и первого столбца матрицы B, при этом, поскольку матрица B хранится в памяти построчно, то элементы одного столбца располагаются в оперативной памяти с некоторым интервалом; чтение одного элемента столбца из оперативной памяти в кэш приводит к считыванию целой линейки данных размером 64 байта.

Таким образом, можно сказать, что в кэш считывается не один столбец матрицы B, а сразу восемь столбцов. Полоса матрицы B может быть настолько велика, что чтение «последних» элементов этой полосы ведет к вытеснению из кэш «первых» элементов строки матрицы A. При вычислении второго элемента первой строки результирующей матрицы может произойти повторное чтение строки матрицы A и чтение второго столбца матрицы B. Так как размер матрицы B может быть настолько большим, что матрица не может быть помещена в кэш процессора полностью, то при вычислении последних элементов строки результирующей матрицы и чтении в кэш элементов последних столбцов, элементы первых столбцов матрицы B будут вытеснены из кэша. Далее, при вычислении элементов следующей строки результирующей матрицы элементы первых столбцов матрицы B необходимо будет снова загрузить в кэш. Итак, если размер матриц составляет $n \times n$ элементов, то матрицы A и B могут переписываться n раз. Эта ситуация имеет место и в случае ленточного и блочного разбиения, если части матриц A и B не могут быть помещены в кэш полностью. Такая организация работы с кэш не может быть признана эффективной.

7.6.1. Последовательный алгоритм

Для организации алгоритма умножения матриц, который более эффективно использует кэш-память, воспользуемся разделением матриц на блоки. В случае, когда необходимо посчитать результат матричного умножения $C = A \times B$, и матрицы, участвующие в умножении, на-

столько велики, что не могут быть помещены в кэш полностью, то становится возможным разделить матрицы на несколько матричных блоков меньшего размера и воспользоваться идеей блочного умножения матриц для того, чтобы получить результат матричного умножения. Разделение на блоки следует проводить таким образом, чтобы размер блока был настолько мал, что три блока, участвующие в умножении на данной итерации, возможно было поместить в кэш. Если блоки матриц A и B могут быть помещены в кэш полностью, то при вычислении результата умножения матричных блоков не происходит многократного чтения элементов блока в кэш, и, следовательно, затраты на загрузку данных из оперативной памяти существенно сокращаются.

Программная реализация. Для реализации последовательного алгоритма умножения матриц, основанного на блочном разбиении матриц и ориентированного на максимально эффективное использование кэш, воспользуемся алгоритмом, описанным в разделе 7.5. Отличие состоит лишь в том, что количество разбиений матриц теперь определяется не количеством потоков, а объемом кэш памяти.

Количество разбиений матриц должно быть таким, чтобы в кэш одновременно могли быть помещены три матричных блока – блоки матриц *A, B* и *C*. Пусть V_{cache} – объем кэш в байтах. Тогда количество элементов типа double (числа с двойной точностью), которые могут быть помещены в кэш, составляет *V/8* (для хранения одного элемента типа double используется 8 байт). Таким образом, максимальный размер квадратного $k \times k$ матричного блока составляет:

$$k_{\max} = \left\lfloor \sqrt{V_{cache}/(3 \cdot 8)} \right\rfloor .$$

Следовательно, минимально необходимое число разбиений *GridSize* при разделении матриц размером $n \times n$ составляет:

$$GridSize = \left\lceil n / k_{\max} \right\rceil .$$

Для снижения сложности программной реализации алгоритма количество блоков по горизонтали и вертикали *GridSize* должно быть таким, чтобы размер матриц n мог быть поделен на *GridSize* без остатка.

В представленной программной реализации положим размер матричного блока равным 250. При таком размере блока три матричных блока могут быть одновременно помещены в кэш (для проведения экспериментов используется процессор Intel Core 2 Duo, объем кэш второго уровня составляет 2 Мб). Кроме того, такие размеры блоков в наших экспериментах гарантируют, что матрицы могут быть поделены на блоки поровну.

Рассмотрим программную реализацию последовательного блочного алгоритма матричного умножения.

```
// Программа 7.7
// Функция блочного матричного умножения
void SerialResultCalculation (double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size) {
  int BlockSize = 250;
  int GridSize = int (Size/double(BlockSize));
  for (int n=0; n<GridSize; n++)
   for (int m=0; m<GridSize; m++)
    for (int iter=0; iter<GridSize; iter++)
     for (int i=n*BlockSize;i<(n+1)*BlockSize; i++)
      for (int j=m*BlockSize;j<(m+1)*BlockSize; j++)
       for (int k=iter*BlockSize;
            k<(iter+1)*BlockSize; k++)
              pCMatrix[i*Size+j] +=
          pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}
```

Следует отметить, что при использовании различных вычислительных систем с размером кэша, отличным от 2 Мб, следует варьировать параметр *BlockSize* таким образом, чтобы три матричных блока могли быть одновременно помещены в кэш.

7.6.2. Параллельный алгоритм

Для распараллеливания представленного блочного алгоритма умножения матриц воспользуемся подходом, который основывается на предложениях, изложенных при рассмотрении ленточного и блочного алгоритмов. Пусть, как и ранее, базовая подзадача (поток) отвечает за вычисление блока результирующей матрицы. Однако теперь, когда количество блоков определяется не количеством потоков, а объемом кэш памяти, число блоков может существенно превосходить число доступных потоков. Проведем агрегацию вычислений – пусть каждый поток обеспечивает получение несколько матричных блоков результирующей матрицы *C*.

1. Анализ эффективности. При анализе эффективности параллельного алгоритма умножения матриц, основанного на разделении данных на блоки определенного размера с тем, чтобы максимально эффективно использовать кэш, можно использовать оценки, полученные при анализе предыдущего алгоритма. Действительно, объем вычислительных операций не изменяется. Поэтому вычислительная сложность блочного алгоритма составляет:

$$T_{calc} = (n^2 / p)(2n-1)\cdot\tau .$$

При умножении матриц с помощью рассмотренного алгоритма выполняется q^3 операций умножения матричных блоков (количество q блоков по вертикали и горизонтали определяется как результат деления порядка n матриц на количество k строк и столбцов матричного блока). При этом размер блоков определяется таким образом, чтобы на каждой итерации они могли быть одновременно помещены в кэш память. Значит, время, необходимое на чтение данных из оперативной памяти в кэш, может быть вычислено по формуле:

$$T_{mem} = \left\lceil n/k \right\rceil^3 \cdot 3\cdot k^2 \left(\alpha + 64/\beta\right) .$$

Следовательно, общее время выполнения параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш, может быть вычислено по формуле:

$$T_p = (n^2/p)(2n-1)\cdot\tau + \left\lceil n/k \right\rceil^3 \cdot 3\cdot k^2 \left(\alpha + 64/\beta\right) . \quad (7.17)$$

Если дополнительно учесть частоту кэш промахов, то выражение (7.17) приобретает следующий вид:

$$T_p = (n^2/p)(2n-1)\cdot\tau + \gamma \left\lceil n/k \right\rceil^3 \cdot 3\cdot k^2 \left(\alpha + 64/\beta\right) . \quad (7.18)$$

2. Программная реализация. Распределим между потоками параллельной программы итерации внешнего цикла (цикла по переменной n). При таком распределении нагрузки на каждой итерации внешнего цикла поток последовательно выполняет поблочное умножение горизонтальной полосы матрицы A на вертикальные полосы матрицы B.

Программная реализация описанного подхода может быть получена следующим образом:

```
// Программа 7.8
// Функция параллельного блочного умножения матриц
void ParallelResultCalculation (double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size) {
  int BlockSize = 250;
  int GridSize = int (Size/double(BlockSize));
#pragma omp parallel for
  for (int n=0; n<GridSize; n++)
   for (int m=0; m<GridSize; m++)
    for (int iter=0; iter<GridSize; iter++)
     for (int i=n*BlockSize;i<(n+1)*BlockSize; i++)
      for (int j=m*BlockSize;j<(m+1)*BlockSize; j++)
       for (int k=iter*BlockSize;
```

```
            k<(iter+1)*BlockSize; k++)
              pCMatrix[i*Size+j] +=
            pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}
```

Можно отметить, что, несмотря на определенную логическую сложность рассмотренного алгоритма, его программная реализация не требует каких-либо значительных усилий.

7.6.3. Результаты вычислительных экспериментов

Вначале необходимо оценить, какое влияние оказывает выбор размера матричного блока (параметр k) на эффективность параллельного алгоритма. Результаты вычислительных экспериментов, проведенных для параллельного алгоритма умножения матриц, эффективно использующего кэш память, при разном размере матричных блоков приведены в табл. 7.10 и на рис. 7.14 (параметр k выбирался таким образом, чтобы матрицы могли быть поделены на равные блоки указанного размера).

Таблица 7.10.

Время выполнения параллельного блочного алгоритма умножения матриц при разных размерах матричных блоков

Размер матриц	Параллельный блочный алгоритм умножения матриц, эффективно использующий кэш-память		
	k=125	k=250	k=500
1000	3,2505	3,3834	8,8142
1500	10,9575	15,4551	20,0040
2000	26,1422	31,0478	35,0169
2500	50,8252	62,8199	116,5353
3000	88,0399	92,4756	169,8120

Как видно из представленных результатов, если размер матричных блоков достаточно мал (k = 125 и k =2 50) и все блоки, участвующие в умножении, могут быть одновременно помещены в кэш, то время выполнения алгоритма практически одинаково. Отсюда можно сделать вывод о том, что накладные расходы на чтение данных (последнее слагаемое в модели (7.18) тем больше, чем меньше k) не оказывают существенного влияния на эффективность параллельного алгоритма. В случае, когда размеры матричных блоков настолько велики, что все блоки, участвующие в умножении, не могут быть одновременно помещены в кэш, эффективность алгоритма существенно снижается; время выполнения алгоритма становится равным времени выполнения параллельных алгоритмов, описанных в предыдущих разделах.

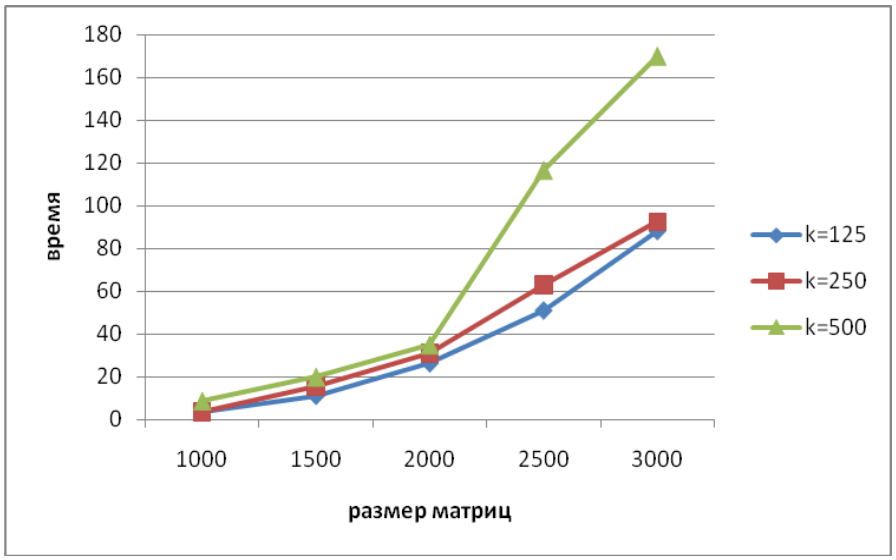

Рис. 7.14. Время выполнения параллельного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, при разных размерах матричных блоков

Таблица 7.11
Результаты вычислительных экспериментов для параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти

Размер матриц	Базовый последовательный алгоритм (T_1)	Блочный последовательный алгоритм (T_1')	Параллельный алгоритм		
			T_p	Ускорение	
				T_1'/T_p	T_1/T_p
1000	24,8192	13,7694	3,3834	4,0696	7,3355
1500	85,8869	47,1803	15,4551	3,0527	5,5572
2000	176,5772	126,6689	31,0478	4,0798	5,6873
2500	403,2405	213,0448	62,8199	3,3914	6,4190
3000	707,1501	377,2822	92,4756	4,0798	7,6469

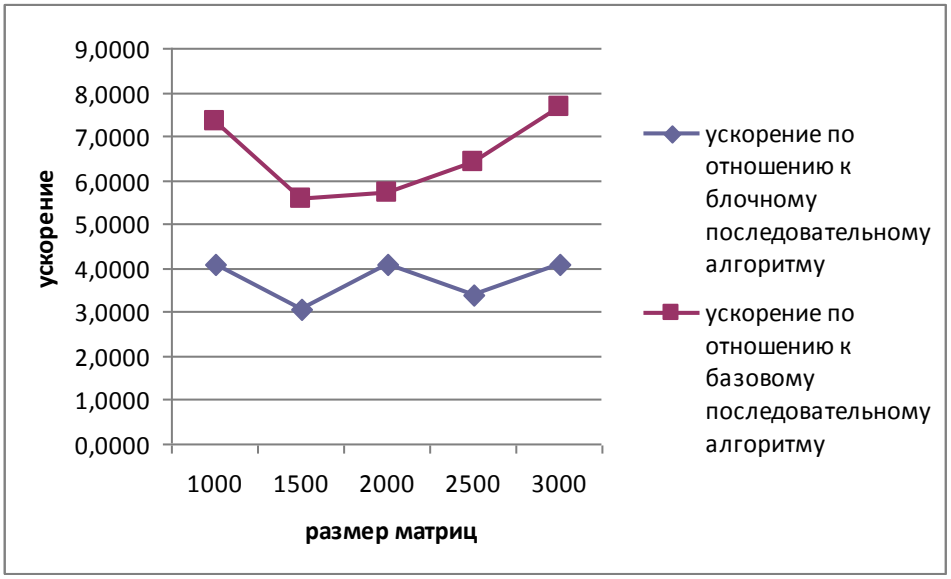

Рис. 7.15. Ускорение параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, в зависимости от размера матриц

В табл. 7.12 и на рис. 7.16 представлены результаты сравнения времени выполнения T_p параллельного алгоритма умножения матриц с использованием четырех потоков со временем T_p^*, полученным при помощи модели (7.18). Частота кэш-промахов, измеренная с помощью системы VPS, для четырех потоков была оценена как 0,0026.

Таблица 7.12.

Сравнение экспериментального и теоретического времени выполнения параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, с использованием четырех потоков

Размер матриц	T_p	T_p^* (*calc*) (модель)	Модель 7.15 – оценка сверху		Модель 7.16 – уточненная оценка	
			T_p^* (*mem*)	T_p^*	T_p^* (*mem*)	T_p^*
1000	3,3834	3,1994	0,1572	3,3566	0,0004	3,1998
1500	15,4551	10,7998	0,5306	11,3304	0,0014	10,8012
2000	31,0478	25,6016	1,2577	26,8593	0,0033	25,6049
2500	62,8199	50,0056	2,4565	52,4621	0,0064	50,0120
3000	92,4756	86,4126	4,2448	90,6574	0,0110	86,4236

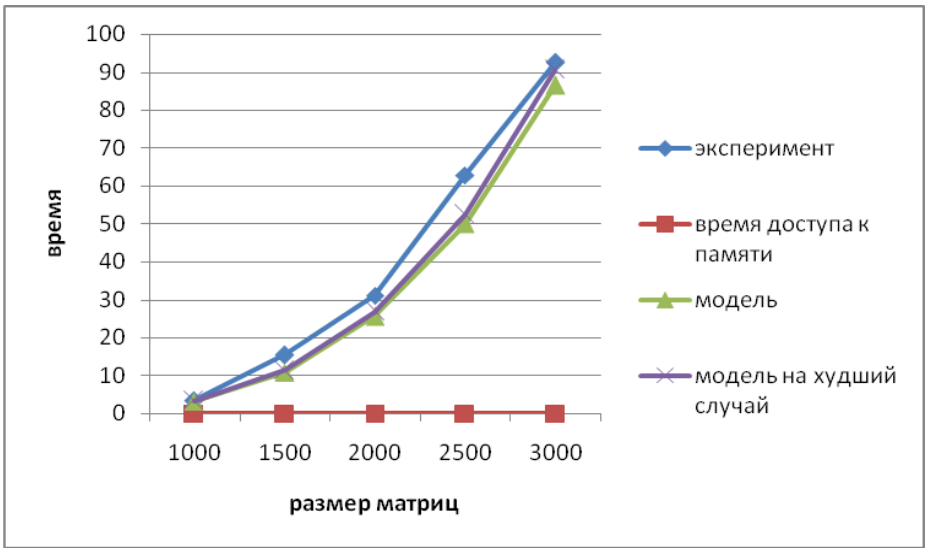

Рис. 7.16. График зависимости экспериментального и теоретического времени выполнения параллельного блочного алгоритма, ориентированного на эффективное использование кэш-памяти, от объема исходных данных при использовании четырех потоков

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при блочном разделении данных, ориентированного на эффективное использование кэш, проводились при условиях, указанных в п. 7.3.6. Результаты вычислительных экспериментов приведены в табл. 7.11 и на рис. 7.15. Времена выполнения алгоритма указаны в секундах.

7.7. Краткий обзор главы

В данной главе рассмотрены четыре параллельных метода для выполнения операции матричного умножения. Первый и второй алгоритмы основаны на ленточном разделении матриц между процессорами. Первый вариант алгоритма основан на разделении между потоками параллельной программы одной из матриц-аргументов (матрицы A) и матрицы-результата. Второй алгоритм основан на разделении первой матрицы на горизонтальные полосы, а второй матрицы – на вертикальные полосы, каждый поток параллельной программы в этом случае вычисляет один блок результирующей матрицы C. При реализации данного подхода используется механизм вложенного параллелизма. Также в главе рассмотрены два блочных алгоритма умножения матриц, по-

следний из которых основывается на разбиении матриц на блоки такого размера, чтобы блоки можно было полностью поместить в кэш-память.

На рис. 7.17 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов (ускорение параллельных алгоритмов показано относительно исходного последовательного метода умножения матриц).

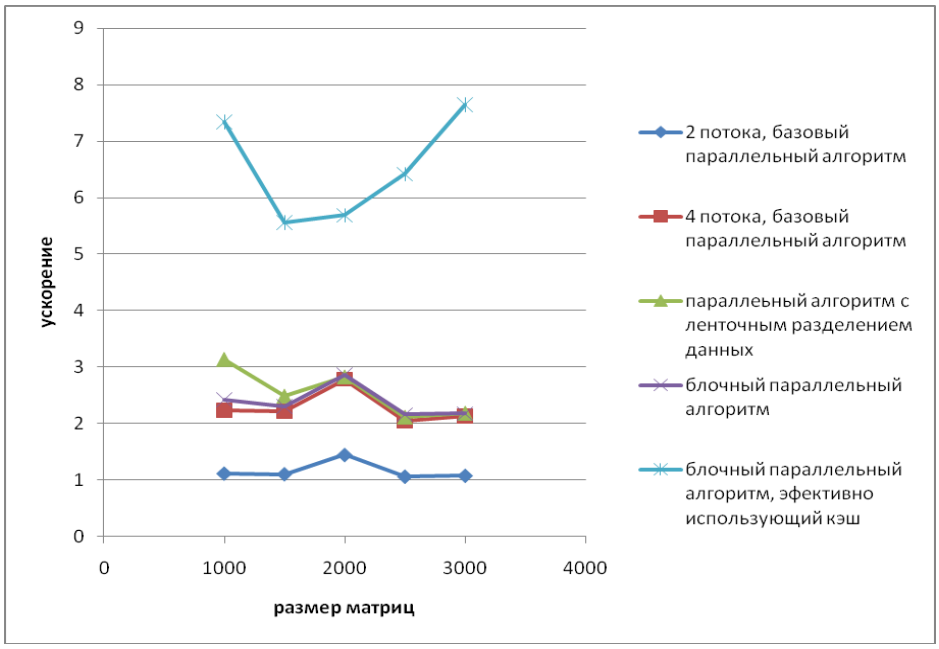

Рис. 7.17. Ускорение вычислений при матричном умножении для всех четырех рассмотренных в разделе параллельных алгоритмов

7.8. Обзор литературы

Задача умножения матриц широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы [8,10,72,85]. Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе [50].

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа [45]. В данной работе рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

7.9. Контрольные вопросы

1. В чем состоит постановка задачи умножения матриц?
2. Приведите примеры задач, в которых используется операция умножения матриц.
3. Приведите примеры различных последовательных алгоритмов выполнения операции умножения матриц. Отличается ли их вычислительная трудоемкость?
4. Какие способы разделения данных используются при разработке параллельных алгоритмов матричного умножения?
5. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матриц.
6. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных?
7. Какие информационные взаимодействия выполняются для блочных алгоритмов умножения матриц?
8. Какой из рассмотренных алгоритмов обладает наилучшими показателями ускорения и эффективности?
9. Оцените возможность выполнения матричного умножения как последовательности операций умножения матрицы на вектор.
10. Какие функции библиотеки OpenMP оказались необходимыми при программной реализации параллельных алгоритмов?

7.10. Задачи и упражнения

1. Выполните реализацию блочных алгоритмов умножения матриц, которые могли бы быть выполнены для прямоугольных решеток потоков общего вида.

2. Выполните реализацию матричного умножения с использованием ранее разработанных программ умножения матрицы на вектор.