Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Учебный курс "Введение в методы параллельного программирования"

Раздел "Параллельные методы матричного умножения "

Разработчик: А.А.Лабутина

Нижний Новгород
2007

Содержание

8. Параллельные методы матричного умножения

Операция умножения матриц является одной из основных задач матричных вычислений. В данном разделе 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.

8.1. 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:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l. \quad (8.1)$$

Как следует из (8.1) , каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

$$c_{ij} = \left(a_i, b_j^T\right), a_i = \left(a_{i0}, a_{i1}, ..., a_{in-1}\right), b_j^T = \left(b_{0j}, b_{1j}, ..., b_{n-1j}\right)^T. \quad (8.2)$$

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*)), но эти алгоритмы требуют определенных усилий для их освоения и, как результат, в данном разделе 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 $n \times n$.

8.2. Последовательный алгоритм

8.2.1. Описание алгоритма

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

```
// Алгоритм 8.2
// 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];
    }
  }
}
```

Алгоритм 8.1. Последовательный алгоритм умножения двух квадратных матриц

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. 8.1)

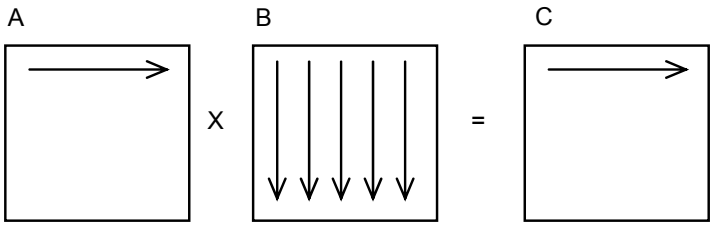

Рис. 8.1. На первой итерации цикла по переменной *i* используется первая строка матрицы *A* и все столбцы матрицы *B* для того, чтобы вычислить элементы первой строки результирующей матрицы *C*

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 $n^2 \cdot (2n-1)$ скалярных операций и затратить время

$$T_1 = n^2 \cdot (2n-1) \cdot \tau \tag{8.3}$$

где τ есть время выполнения одной элементарной скалярной операции.

8.2.2. Анализ эффективности

При анализе эффективности последовательного алгоритма умножения матриц будем опираться на положения, изложенные в разделе 7.5.4.

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 (8.3).

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*. Поскольку матрицы хранятся в памяти построчно, а чтение в кэш происходит линейками по 64 байта (8 элементов типа double), то для вычисления одного элемента результирующей матрицы необходимо прочитать $n+8n$ элементов. Для записи полученного результата дополнительно требуется чтение соответствующего элемента матрицы *C* из оперативной памяти (поскольку переписывается линейка кэша полностью, данная операция сводится к чтению 8 элементов типа double). Важно отметить, что приведенные оценки количества читаемых из памяти данных справедливы, если все эти данные отсутствуют в кэше. В реальности часть этих данных может 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 линеек 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 $n^3+8n^{3+}+8n^2$.

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

$$T_1 = n^2(2n-1)\cdot\tau + \frac{8\cdot(n^3+8n^3+8n^2)}{\beta} = n^2(2n-1)\cdot\tau + \frac{8\cdot(9n^3+8n^2)}{\beta} \tag{8.4}$$

где β есть пропускная способность канала доступа к оперативной памяти.

8.2.3. Программная реализация

Представим возможный вариант последовательной программы умножения матриц.

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

```
// Программа 8.1
// Последовательное умножение матриц
void main(int argc, char* argv[]) {
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result of matrix multiplication
```

```
  int Size;          // Sizes of matricies

  // Data initialization
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // Matrix-vector multiplication
  SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);

  // Program termination
  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.*

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  int i, j;  // Loop variables

  do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    if (Size <= 0) {
      printf("Size of the objects must be greater than 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.

```
// Function for calculating matrix multiplication
void SerialResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
  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.

8.2.4. Результаты вычислительных экспериментов

Эксперименты проводились на вычислительном узле на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2003, для компиляции использовался Intel C++ Compiler 9.0 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. Для того, 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 таблице 8.1 и на рис. 8.2.

Таблица 8.1. Сравнение времени выполнения оптимизированной и неоптимизированной версии последовательного алгоритма умножения матриц

Размер матриц	Компиляторная оптимизация включена	Компиляторная оптимизация выключена
1000	8,7647	23,0975
1500	30,1341	80,9673
2000	85,1422	172,8723
2500	160,3179	381,3749
3000	313,8816	672,9809

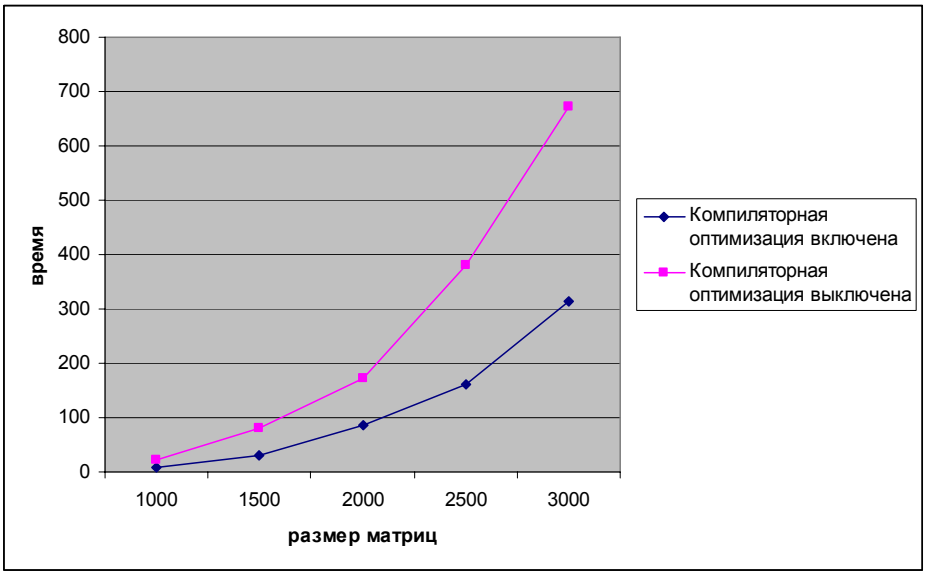

Рис. 8.2. Графики зависимости времени выполнения оптимизированной и неоптимизированной версий последовательного алгоритма

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, так как 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 6,550 нс.

Оценка величины пропускной способности канала доступа к оперативной памяти β проводилась в п. 7.5.4 и определена для используемого вычислительного узла как 5,5 Гб/с.

В таблице 8.2 и на рис. 8.3 представлены результаты сравнения времени выполнения последовательного алгоритма умножения матриц со временем, полученным при помощи модели (8.4). Как следует из приведенных данных, погрешность аналитической оценки трудоемкости алгоритма матричного умножения уменьшается при увеличении размера матриц (для n=3000 относительная погрешность составляет порядка 5%).

Таблица 8.2. Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма умножения матриц

Размер матриц	Эксперимент	Время счета	Время доступа к памяти	Модель
1000	23,0975	13,0937	13,1025	26,1963
1500	80,9673	44,1987	44,2080	88,4067
2000	172,8723	104,7760	104,7738	209,5499
2500	381,3749	204,6509	204,6182	409,2691
3000	672,9809	353,6486	353,5593	707,2079

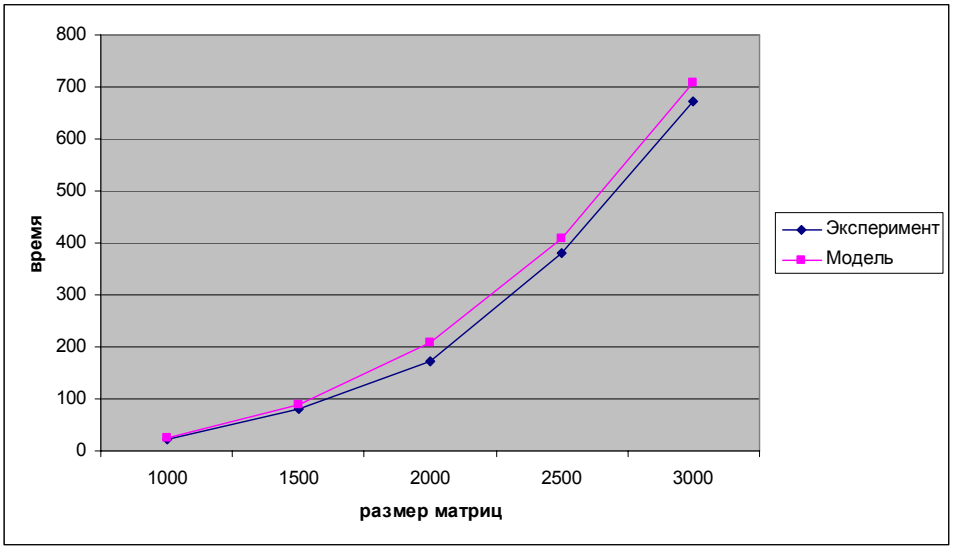

Рис. 8.3. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных

8.3. Базовый параллельный алгоритм умножения матриц

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

8.3.1. 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. Обычно при проведении практических расчетов количество сформированных подзадач превышает число имеющихся вычислительных элементов (процессоров и/или ядер) и, как результат, 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 подраздела 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 дублированию данных, поскольку разрабатываемый алгоритм ориентирован на применение для вычислительных систем с общей разделяемой памятью, к которой имеется доступ со всех используемых вычислительных элементов.

8.3.2. 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 рисунке 8.4.

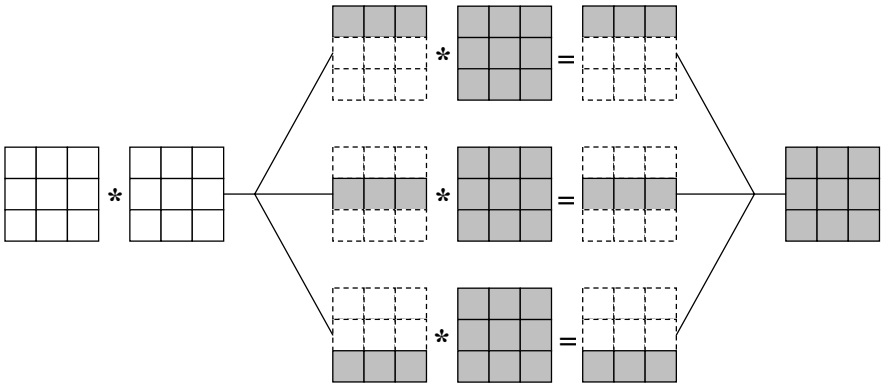

Рис. 8.4. Организация вычислений при выполнении параллельного алгоритма умножения матриц, основанного на разделении матриц по строкам

8.3.3. Масштабирование и распределение подзадач

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае, когда размер матриц n оказывается больше, чем число вычислительных элементов (процессоров и/или ядер) p, 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.

8.3.4. Анализ эффективности

Будем проводить анализ эффективности реализации базового параллельного алгоритма умножения матриц по схеме, изложенной в разделе 7.5.4.

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:

$$T_{calc} = \frac{n^2(2n-1)}{p} \cdot \tau . \qquad (8.5)$$

Для оценки объема данных, которые необходимо прочитать из оперативной памяти в кэш, снова применим подход, изложенный в п. 8.2.3. 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. 7.5.4 и рис. 7.4). 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. Как результат, время работы с оперативной памятью при выполнении описанного параллельного алгоритма умножения матриц может быть определено в соответствии со следующим соотношением:

$$T_{mem} = \frac{8 \cdot \left(n^3 + 8n^3/p + 8n^2\right)}{\beta}, \tag{8.6}$$

где, как и ранее, β есть пропускная способность канала доступа к оперативной памяти.

Следовательно, время выполнения параллельного алгоритма составляет:

$$T_p = \frac{n^2(2n-1)}{p} \cdot \tau + \frac{8 \cdot \left(n^3 + 8n^3/p + 8n^2\right)}{\beta} \tag{8.7}$$

Следует отметить, что при выполнении задачи несколькими потоками часть обращений к памяти может выполняться одновременно с вычислениями (см. раздел 7.5.4 и рис. 7.6). При выполнении матричного умножения большая доля времени тратится на обращение к памяти, а так как процесс вычислений и чтение необходимых данных из оперативной памяти в кэш процессора могут происходить одновременно, формула (8.7) может быть модифицирована следующим образом:

$$\frac{8 \cdot \left(n^3 + 8n^3/p + 8n^2\right)}{\beta} \le T_p \le \frac{n^2(2n-1)}{p} \cdot \tau + \frac{8 \cdot \left(n^3 + 8n^3/p + 8n^2\right)}{\beta}. \tag{8.8}$$

8.3.5. 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*):

```
// Function for calculating matrix multiplication
void ParallelResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
#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*.

8.3.6. Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности параллельного алгоритма умножения матрицы на вектор. Эксперименты проводились на вычислительном узле на базе процессора Intel Core 2 6300, 1.87 ГГц, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2003, для компиляции использовался Intel C++ Compiler 9.0 for Windows.

Результаты вычислительных экспериментов приведены в таблице 8.3. Времена выполнения алгоритмов указаны в секундах.

Таблица 8.3. Результаты вычислительных экспериментов для параллельного алгоритма умножения матриц при ленточной схеме разделении данных по строкам

Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	23,0975	11,5856	1,9937
1500	80,9673	40,2758	2,0103
2000	172,8723	87,0490	1,9859
2500	381,3749	195,6257	1,9495
3000	672,9809	338,2172	1,9898

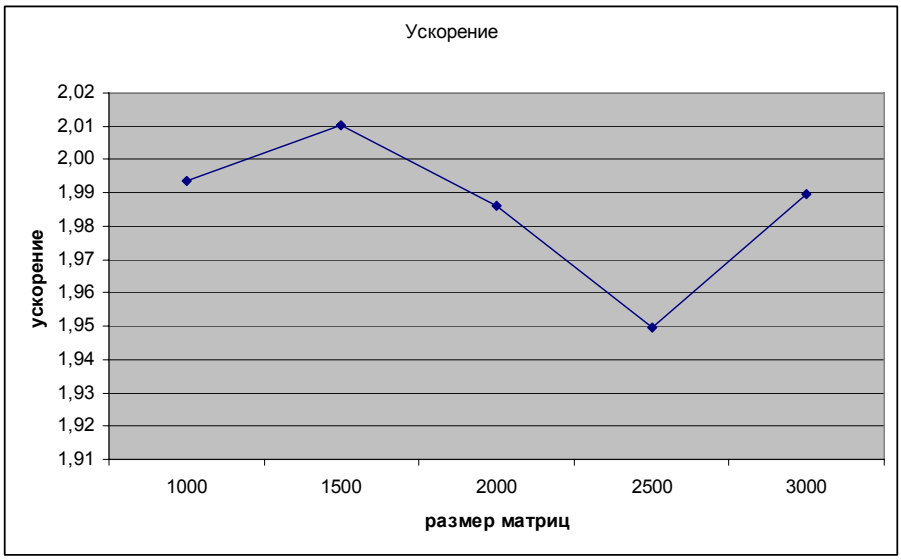

Рис. 8.5. Зависимость ускорения от количества исходных данных при выполнении базового параллельного алгоритма умножения матриц

В таблице 8.4 и на рис. 8.6 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матриц с использованием двух потоков со временем, полученным при помощи модели (8.7).

Таблица 8.4. Сравнение экспериментального и теоретического времени выполнения базового параллельного алгоритма умножения матриц с использованием двух потоков

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Модель
1000	11,585557	6,5469	7,2844	13,8312
1500	40,275836	22,0994	24,5716	46,6710
2000	87,049049	52,3880	58,2284	110,6164
2500	195,625736	102,3255	113,7091	216,0346
3000	338,217195	176,8243	196,4684	373,2927

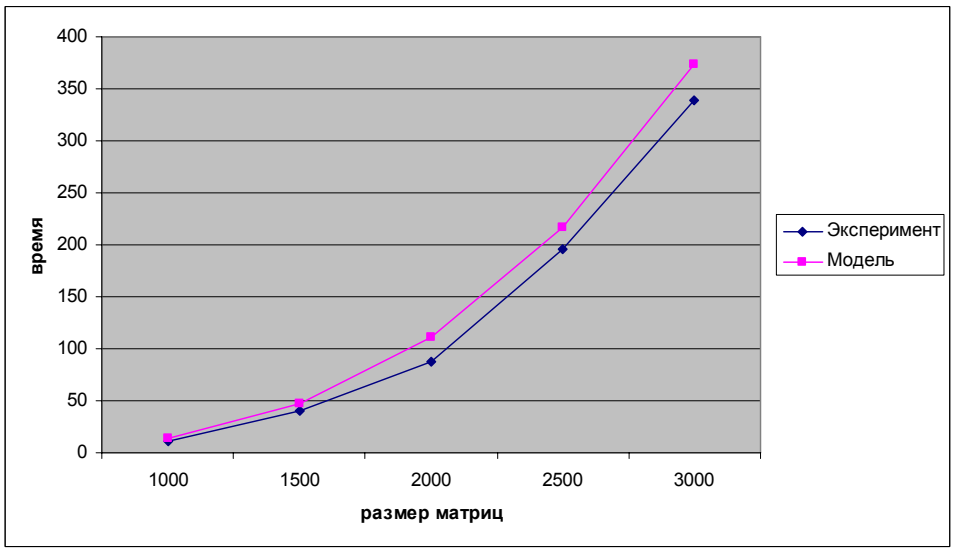

Рис. 8.6. График зависимости экспериментального и теоретического времени выполнения базового параллельного алгоритма от объема исходных данных при использовании двух потоков

8.4. Алгоритм умножения матриц, основанный на ленточном разделении данных

В рассмотренном в подразделе 8.3 только одна из перемножаемых матриц – матрица A – распределялась между параллельно выполняемыми потоками. В данном подразделе 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.

8.4.1. 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 подразделе 7.2).

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.

8.4.2. Выделение информационных зависимостей

Для вычисления одного элемента c_{ij} результирующей матрицы необходимо выполнить скалярное умножение i-ой строки матрицы A и j-ого 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 разделе 7.7.2.

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. 8.7).

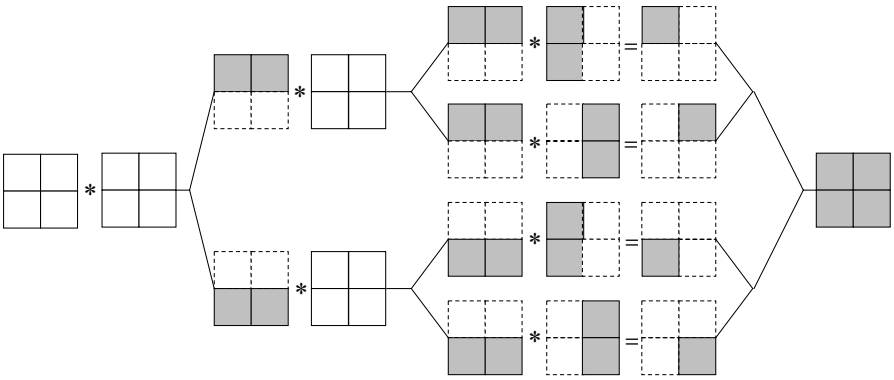

Рис. 8.7. Организация параллельных вычислений при выполнении параллельного алгоритма умножения матриц, основанного на ленточном разделении данных, и использованием четырех потоков

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

8.4.3. Масштабирование и распределение подзадач

Размер блоков матрицы C может быть подобран таким образом, чтобы общее количество базовых подзадач совпадало с числом выделенных потоков π. Так, например, если 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.

8.4.4. 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, тогда время выполнения вычислений составляет:

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

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 $n+8n+8$ 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. 7.5.4 и рис. 7.4). 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, 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 вертикального ряда блоков матрицы C). Таким образом, время, необходимое на чтение необходимых данных из оперативной памяти составляет:

$$T_{mem} = \frac{8 \cdot \left(n^3 + 8n^3/q + 8n^2\right)}{\beta},$$

где β есть пропускная способность канала доступа к 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 может быть определена следующим образом:

$$T_p = \frac{n^2 \cdot (2n-1)}{p} + \frac{8 \cdot \left(n^3 + 8n^3/q + 8n^2\right)}{\beta} + n \cdot \delta \quad (8.9)$$

8.4.5. 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 9.0 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 итерации внутреннего цикла для разделения матрицы *B* на вертикальные полосы.

```
// Function for calculating matrix multiplication
void ParallelResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
  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 при изменении необходимого числа потоков).

8.4.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при ленточном разделении данных проводились при условиях, указанных в п. 8.3.6. Результаты вычислительных экспериментов приведены в таблице 8.5. Времена выполнения алгоритма указаны в секундах.

Таблица 8.5. Результаты вычислительных экспериментов для параллельного алгоритма умножения матриц при ленточной схеме разделения данных

Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	23,0975	11,6076	1,9899
1500	80,9673	42,3614	1,9113
2000	172,8723	87,4104	1,9777
2500	381,3749	196,9081	1,9368
3000	672,9809	345,1124	1,9500

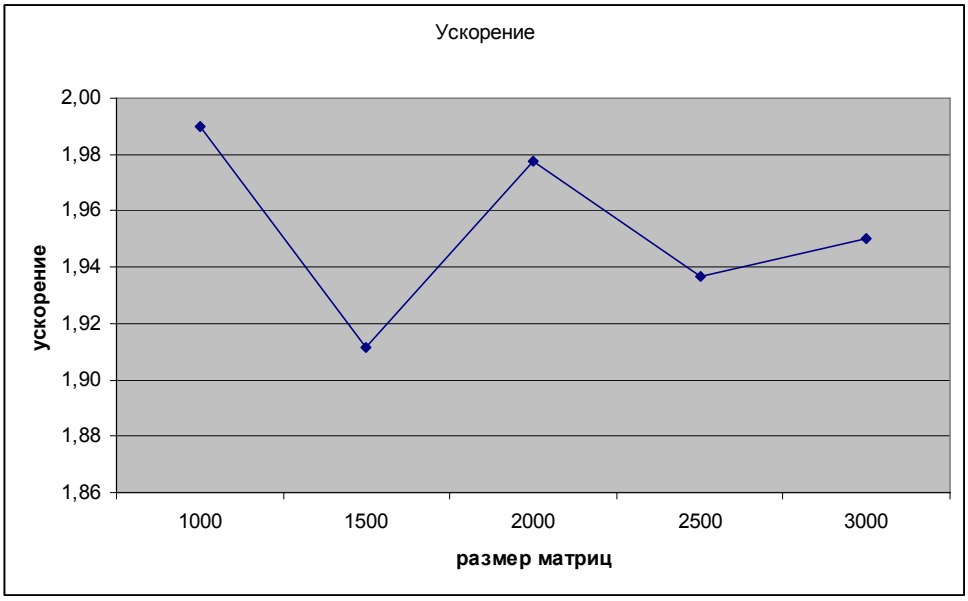

Рис. 8.8. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма умножения матриц, основанного на ленточном разделении матриц

Для того, 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.

```
void ResultCalculation (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)];
  }
}
```

Для того, чтобы оценить накладные расходы δ на организацию параллельности на каждой итерации алгоритма, необходимо из времени выполнения исходного алгоритма умножения матриц вычесть время выполнения вновь разработанного параллельного алгоритма и поделить полученную разницу на количество созданий вложенных параллельных секций (т.е. на n/q). Эксперименты показывают, что величина δ равна 500 микросекунд ($5 \cdot 10^{-4}$ с).

В таблице 8.6 и на рис. 8.9 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матриц, основанного на ленточном разделении данных, с использованием четырех потоков со временем, полученным при помощи модели (8.9).

Таблица 8.6. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матриц, основанного на ленточном разделении данных, с использованием четырех потоков

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Модель
1000	11,6076	6,5469	7,2844	14,3312

1500	42,3614	22,0994	24,5716	47,4210
2000	87,4104	52,3880	58,2284	111,6164
2500	196,9081	102,3255	113,7091	217,2846
3000	345,1124	176,8243	196,4684	374,7927

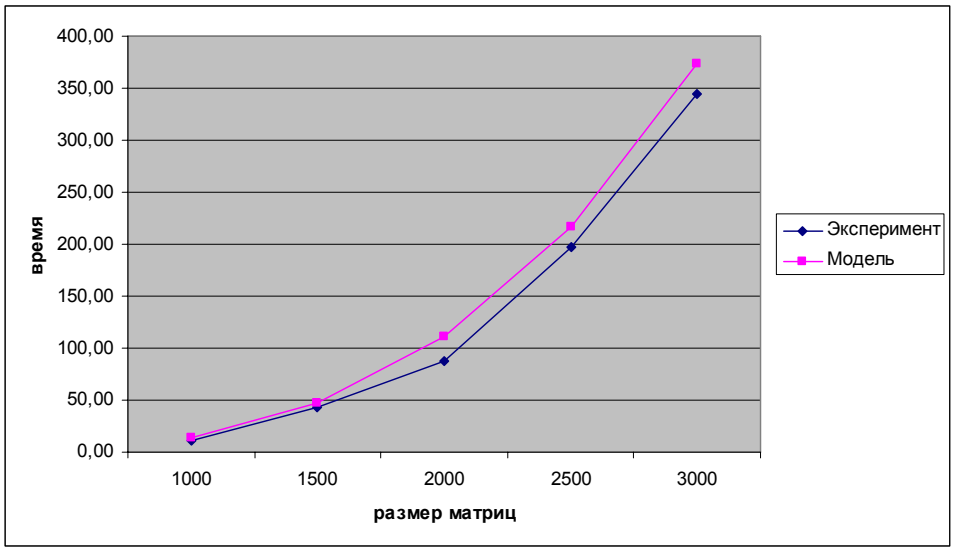

Рис. 8.9. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма, основанного на ленточном разделении данных, от объема исходных данных при использовании четырех потоков

8.5. 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. Выполним более подробное рассмотрение данного способа организации вычислений. При этом 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.

8.5.1. Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 7.2. 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:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0q-1} \\ \dots & \dots & \dots & \dots \\ A_{q-10} & A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0q-1} \\ \dots & \dots & \dots & \dots \\ B_{q-10} & B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0q-1} \\ \dots & \dots & \dots & \dots \\ C_{q-10} & C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix}, \quad (8.10)$$

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

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj} \ . \quad (8.11)$$

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. 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 (8.11). Возможная естественным образом определяемая вычислительная схема рассматривается в п. 8.5.2.

8.5.2. 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 в 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.

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-ый блок полосы матрицы A умножается на i-ый блок полосы матрицы B, результат умножения блоков прибавляется к блоку результирующей матрицы (рис. 8.10). Количество итераций определяется размером блочной решетки.

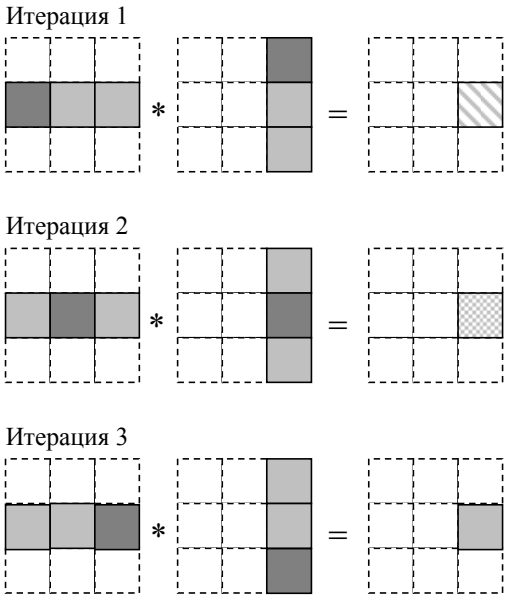

Рис. 8.10. Схема организации блочного умножения полос

8.5.3. 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, например, 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.

8.5.4. 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:

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

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)×(n/q)*. Для оценки объема данных, который при выполнении этой операции должен быть прочитан из оперативной памяти в кэш, можно воспользоваться формулой (8.4), полученной при анализе эффективности последовательного алгоритма, с поправкой на размер матричного блока.

$$T_{mem}^1 = \frac{8 \cdot \left(9 \cdot (n/q)^3 + 8 \cdot (n/q)^2\right)}{\beta}$$

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:

$$T_{mem} = \frac{8 \cdot \left(9 \cdot (n/q)^3 + 8 \cdot (n/q)^2\right)}{\beta} \cdot q \cdot q^2 \ .$$

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:

$$T_{mem} = \frac{8 \cdot \left(9 \cdot (n/q)^3 + 8 \cdot (n/q)^2\right)}{\beta} \cdot q \cdot q = \frac{8 \cdot \left(9 \cdot n^3/q + 8 \cdot n^2\right)}{\beta} .$$

Как итог, общее время выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении данных, может быть описано соотношением:

$$\frac{8 \cdot \left(9 \cdot n^3/q + 8 \cdot n^2\right)}{\beta} \le T_p \le \frac{n^2(2n-1)}{p} \cdot \tau + \frac{8 \cdot \left(9 \cdot n^3/q + 8 \cdot n^2\right)}{\beta} \qquad (8.12)$$

8.5.5. Программная реализация

Согласно вычислительной схеме блочного алгоритма умножения матриц, описанной в разделе 8.5.2, 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 переменную *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*.

```
void ResultCalculation (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
}
```

8.5.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при блочном разделении данных проводились при условиях, указанных в п. 8.3.6. Результаты вычислительных экспериментов приведены в таблице 8.7. Времена выполнения алгоритма указаны в секундах.

Таблица 8.7. Результаты вычислительных экспериментов для параллельного блочного алгоритма умножения матриц

Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	23,0975	11,7066	1,9730
1500	80,9673	40,7939	1,9848
2000	172,8723	87,7952	1,9690
2500	381,3749	195,2347	1,9534
3000	672,9809	343,2033	1,9609

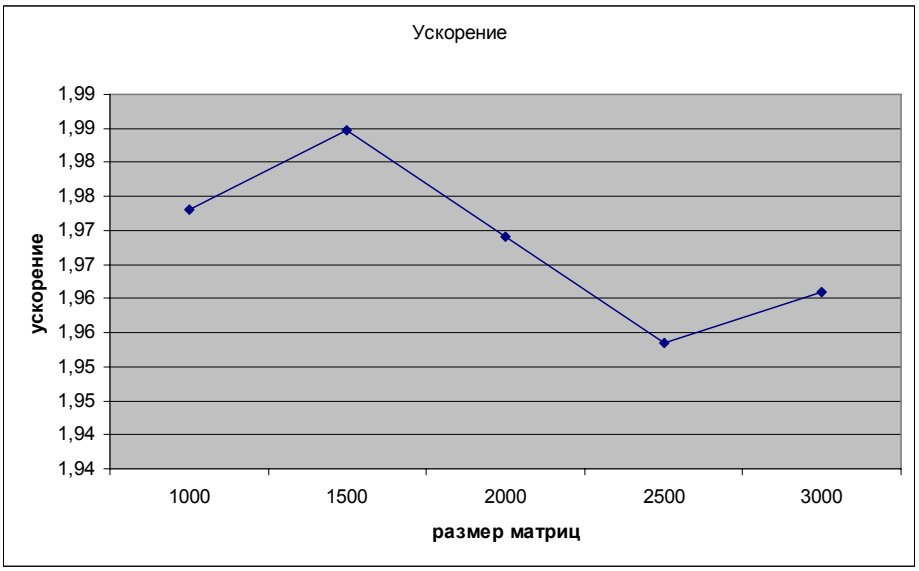

Рис. 8.11. Зависимость ускорения от количества исходных данных при выполнении параллельного блочного алгоритма умножения матриц

В таблице 8.8 и на рис. 8.12 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении данных, с использованием четырех потоков со временем, полученным при помощи модели (8.12). Можно отметить, что погрешность аналитической оценки трудоемкости алгоритма матричного умножения уменьшается при увеличении размера матриц (для n=3000 относительная .погрешность составляет порядка 3%).

Таблица 8.8. Сравнение экспериментального и теоретического времени выполнения параллельного блочного алгоритма умножения матриц с использованием четырех потоков

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Модель
1000	11,7066	6,5469	6,5571	13,1040
1500	40,7939	22,0994	22,1171	44,2164
2000	87,7952	52,3880	52,4102	104,7982
2500	195,2347	102,3255	102,3455	204,6709
3000	343,2033	176,8243	176,8320	353,6563

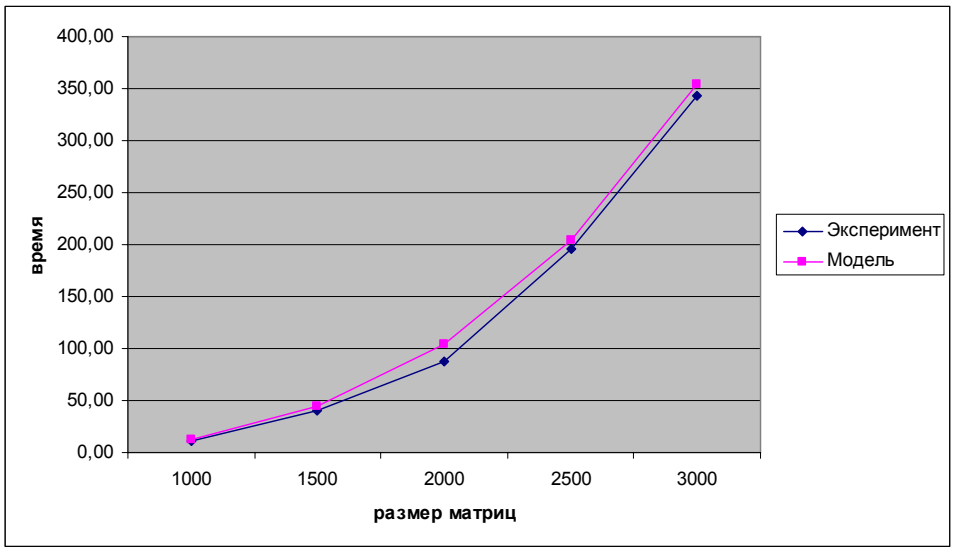

Рис. 8.12. График зависимости экспериментального и теоретического времени выполнения параллельного блочного алгоритма от объема исходных данных при использовании четырех потоков

8.6. 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. 8.1). 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, даже если части матриц A и B не могут быть помещены в кэш полностью. Такая организация работы с кэш не может быть признана эффективной.

8.6.1. 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, настолько 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 посчитать 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.

8.6.1.1 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 8.5. Отличие состоит лишь в том, что количество разбиений матриц теперь определяется не количеством потоков, а объемом кэш.

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:

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

Для упрощения 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.

```
// Serial block matrix mutiplication
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.

8.6.2. 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 кэша,

число блоков может существенно превосходить число доступных потоков. Проведем агрегацию вычислений – пусть каждый поток обеспечивает получение несколько матричных блоков результирующей матрицы *C*.

8.6.2.1 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:

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

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

$$T_{mem} = \frac{8 \cdot (n/k)^3 \cdot 3 \cdot k^2}{\beta} = \frac{24 \cdot n^3 / k}{\beta} .$$

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

$$T_p = \frac{n^2(2n-1)}{p} \cdot \tau + \frac{24 \cdot n^3 / k}{\beta} . \qquad (8.13)$$

8.6.2.2 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:

```
// Parallel block matrix mutiplication
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.

8.6.3. 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, при разном размере матричных блоков, приведены в таблице 8.9 и на рис. 8.13 (параметр k выбирался таким образом, чтобы матрицы могли быть поделены на равные блоки указанного размера).

Таблица 8.9. 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	6,5351	6,8506	11,3545
1500	22,1875	23,4035	52,3986
2000	52,6805	63,4422	80,8310
2500	102,6425	105,9484	221,7003
3000	178,0164	186,9117	311,5686

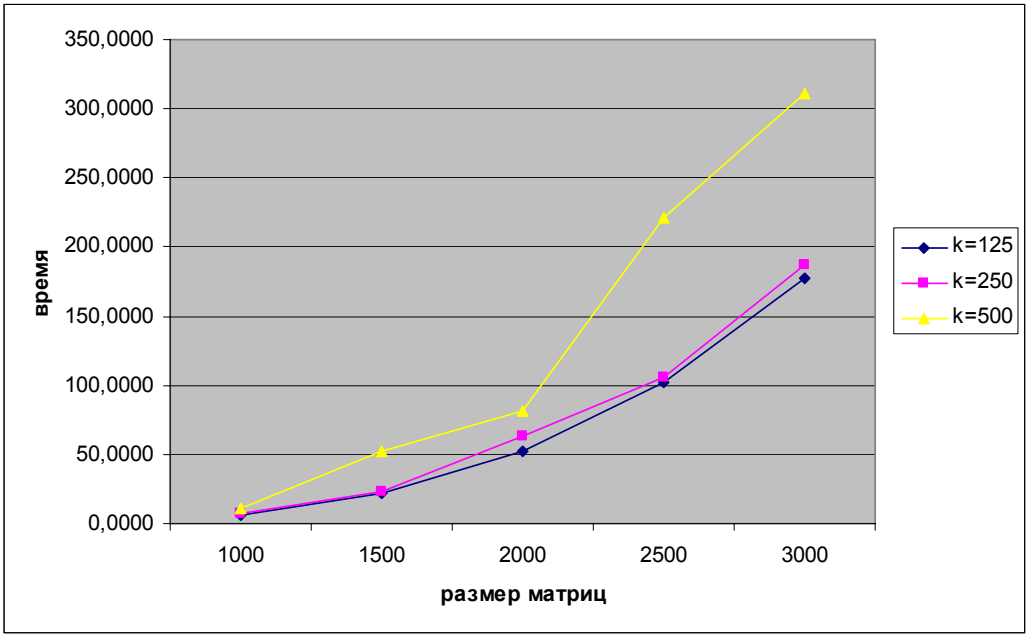

Рис. 8.13. Время выполнения параллельного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, при разных размерах матричных блоков

Как видно из представленных результатов, если размер матричных блоков достаточно мал (k=125 и k=250) 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 организацию параллелизма (последнее слагаемое в модели (8.13) 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.

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при блочном разделении данных, ориентированного на эффективное использование кэш, проводились при условиях, указанных в п. 8.3.6. Результаты вычислительных экспериментов приведены в таблице 8.10. Времена выполнения алгоритма указаны в секундах.

Таблица 8.10. 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	23,0975	13,6600	6,8506	1,9940	3,3716
1500	80,9673	46,8507	23,4035	2,0019	3,4596
2000	172,8723	125,6692	63,4422	1,9808	2,7249
2500	381,3749	210,9799	105,9484	1,9913	3,5996
3000	672,9809	373,8709	186,9117	2,0003	3,6005

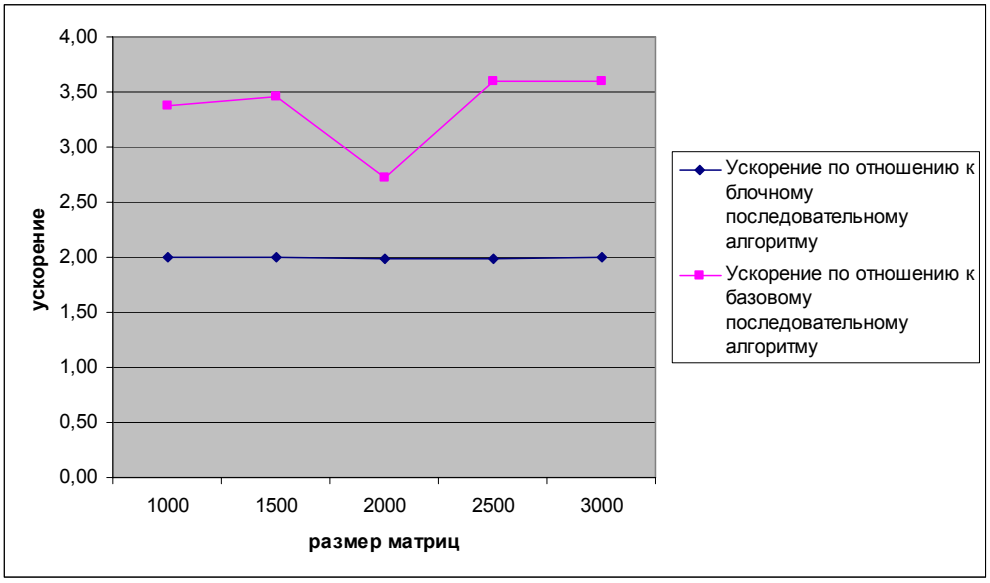

Рис. 8.14. Ускорение параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, в зависимости от размера матриц

В таблице 8.11 и на рис. 8.15 представлены результаты сравнения времени выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении данных, с использованием двух потоков со временем, полученным при помощи модели (8.13). Необходимо отметить, что модель (8.13) справедлива только в случае, когда размер матричного блока подобран таким образом, что все матричные блоки, участвующие в умножении на данной итерации алгоритма, могли быть одновременно размещены в кэш-памяти.

Таблица 8.10. Сравнение экспериментального и теоретического времени выполнения параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, с использованием двух потоков

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Модель
1000	6,8506	6,5469	0,0175	6,5643
1500	23,4035	22,0994	0,0589	22,1583
2000	63,4422	52,3880	0,1396	52,5277
2500	105,9484	102,3255	0,2727	102,5982
3000	186,9117	176,8243	0,4713	177,2956

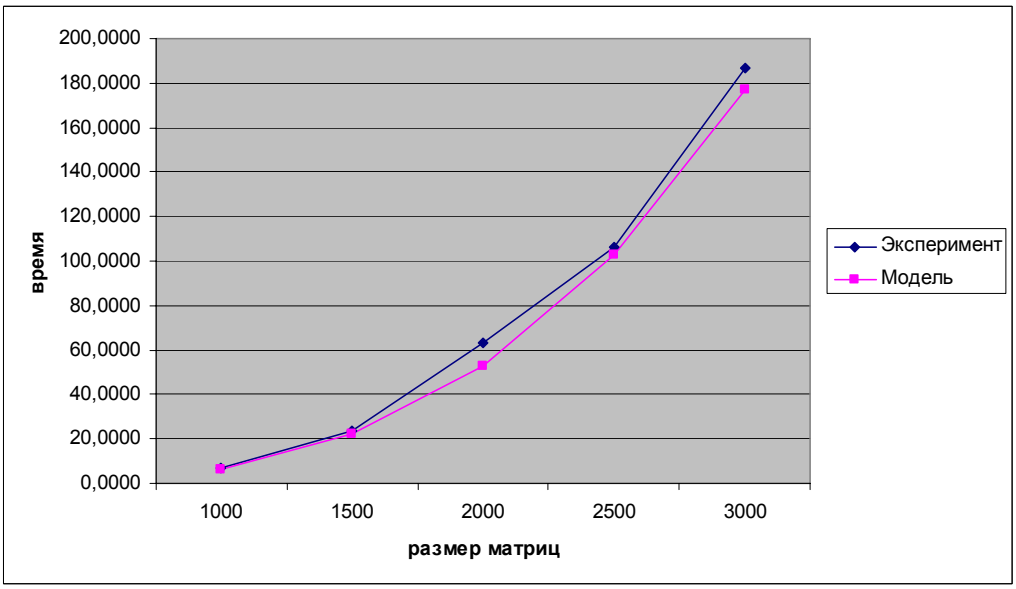

Рис. 8.15. График зависимости экспериментального и теоретического времени выполнения параллельного блочного алгоритма, ориентированного на эффективное

использование кэш-памяти, от объема исходных данных при использовании двух потоков

8.7. Краткий обзор раздела

В данном разделе 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 разделе рассматриваются два блочных алгоритма умножения матриц, последний из которых основывается на разбиении матриц на блоки такого размера, чтобы блоки можно было полностью поместить в кэш-память.

На рис. 8.16 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов.

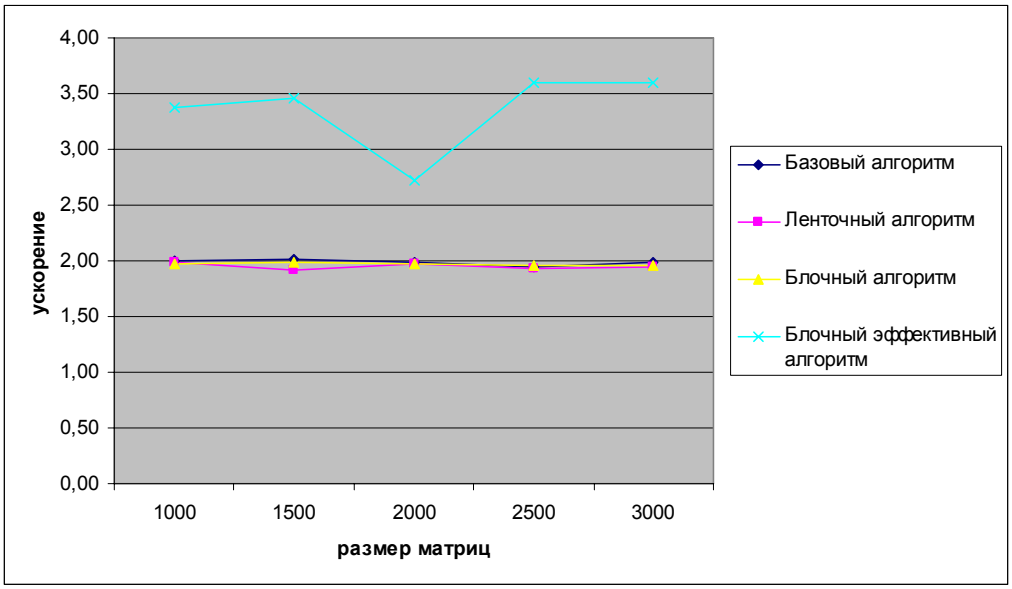

Рис. 8.16. Ускорение вычислений при матричном умножении для всех четырех рассмотренных в разделе параллельных алгоритмов

8.8. Обзор литературы

Задача умножения матриц широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы Воеводин В.В. и Воеводин Вл.В. (2002), Kumar (1994) и Quinn (2003). Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе Dongarra, et al. (1999).

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа Blackford, et al. (1997). В данной работе рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

8.9. 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 алгоритмов?

8.10. 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.