Высокопроизводительные параллельные вычисления

Лекция №6

Тема: Параллельные методы матричного умножения

Козбакова Айнур Холдасовна– PhD, профессор, кафедра «Информационные системы» Факультет «Инжиниринг и информационных технологии» 01.09.2025 г.

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.

Постановка задачи

Умножение матрицы A размера $m*n$ и матрицы B размера $n*l$ приводит к получению матрицы Cразмера $m*l$, каждый элемент которой определяется в соответствии с выражением

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l \tag{4.1}$$

Как следует из (4.1), каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

$$c_{ij} = \left(a_i, b_j^T\right), a_i = \left(a_{i0}, a_{i1}, \ldots, a_{in-1}\right), b_j^T = \left(b_{0j}, b_{1j}, \ldots, b_{n-1j}\right)^T \tag{4.2}$$

Этот алгоритм предполагает выполнение $m \cdot n \cdot l$ операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера $n*n$ количество выполненныхопераций 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 $n * n$.

4.1 Последовательный алгоритм

Описание алгоритма $n*n$. Последовательный алгоритм умножения матриц представляется тремя вложеннымициклами:

```
// Алгоритм 4.1

// 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;
```

Алгоритм 4.1. Последовательный алгоритм умножения двух квадратныхматриц

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. 4.1)

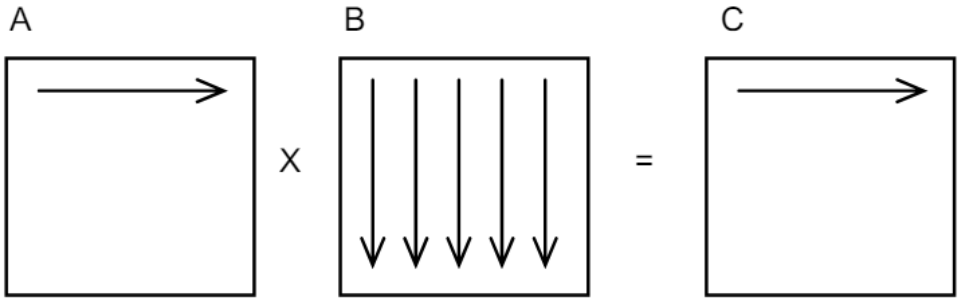

Рисунок 4.1 – На первой итерации цикла по переменной *i* используется перваястрока матрицы *A* и все столбцы матрицы *B* для того, чтобы вычислить элементы первой строки результирующей матрицы *C*

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

$$T_1 = n^2(2n-1)\ \tau \qquad (4.3)$$

где τ есть время выполнения одной элементарной скалярной операции.

Анализ эффективности.

При анализе эффективности последовательного алгоритма умножения матриц будем опираться на положения, изложенные в разделе 3.5

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 (4.3).

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 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:

$$T_1 \approx n^2(2n-1)\tau + 64\beta(2n^3+n^2)/b \qquad (4.4)$$

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 предыдущем разделе, помимо пропускной способности учесть латентность памяти, модель приобретет следующий вид:

$$T_1 \approx n^2(2n-1)\tau + (2n^3+n^2)\alpha + 64/b \qquad (4.5)$$

где α есть латентность оперативной памяти. (4.5) 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.

Как и ранее, введем в разработанную модель (4.5) величину g, $0 \le g \le 1$, для задания *частоты возникновения кэш промахов*. Тогда оценка времени выполнения алгоритма матричного умножения принимает вид:

$$T_1 \approx n^2(2n-1)\tau + g(2n^3+n^2)\alpha + 64/b \qquad (4.6)$$

Программная реализация

Представим возможный вариант последовательной программы умножения матриц.

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

```
// Программа 4.1

// Serial matrix multiplication
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);
```

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 matricies: ");
    scanf("%d", &Size);

    if (Size <= 0) {

      printf("Size of the matricies must be greater than 0! \n ");

    }

  } while (Size <= 0);
```

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++)
```

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.

Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных 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. 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 4.1 и на рис. 4.2.

Таблица 4.1 – Сравнение времени выполнения оптимизированной и неоптимизированной версии последовательного алгоритма умножения матриц

Размерматриц	Компиляторная оптимизациявключена	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

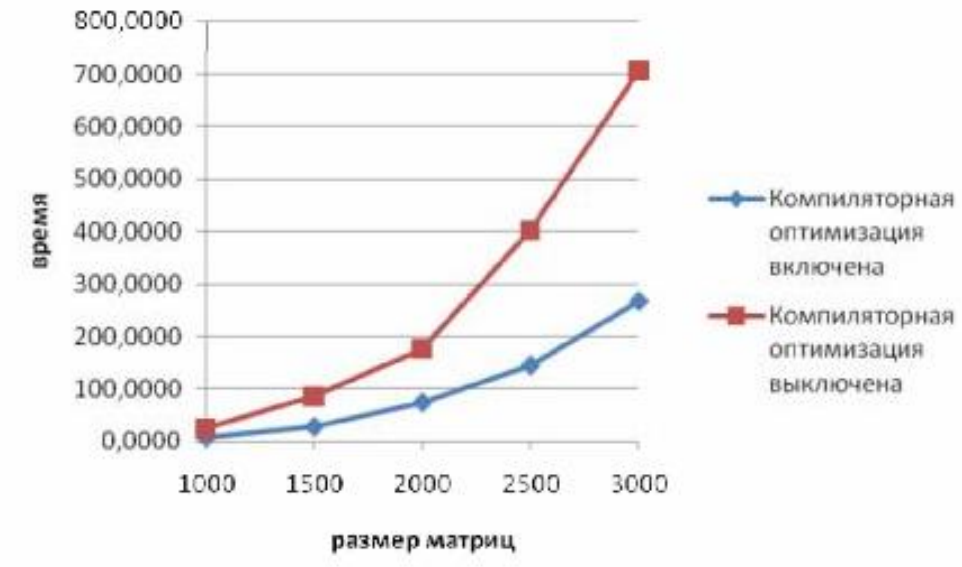

Рисунок 4.2 – Графики зависимости времени выполнения оптимизированной и неоптимизированной версий последовательного алгоритма

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 6,402 нс.

Оценка времени латентности α и величины пропускной способности канала доступа к оперативной памяти β проводилась в п. 3.5 и определена для используемого вычислительного узла как β = 12,44 Гб/с и α = 8,31 нс.

В таблице 4.2 и на рис. 4.3 представлены результаты сравнения времени выполнения последовательного алгоритма умножения матриц со временем, полученным при помощи модели (4.6). 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.

Таблица 4.2 – Сравнение экспериментального и теоретического временивыполнения 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

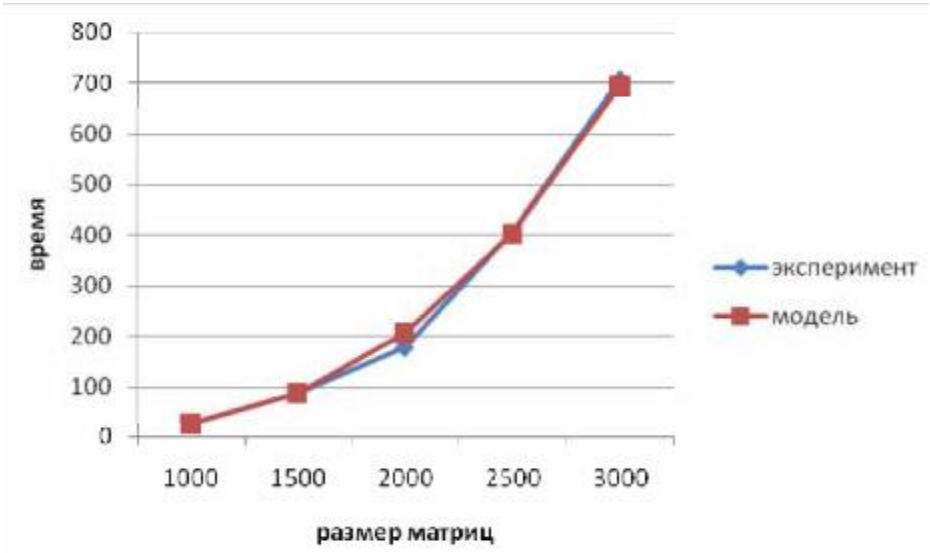

Рисунок 4.3 –График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных

4.2 Базовый параллельный алгоритм умножения матриц

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

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 реальному дублированию данных, поскольку разрабатываемый алгоритм ориентирован на применение для вычислительных систем с общей разделяемой памятью, к которой имеется доступ со всех используемых вычислительных элементов.

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. 4.4.

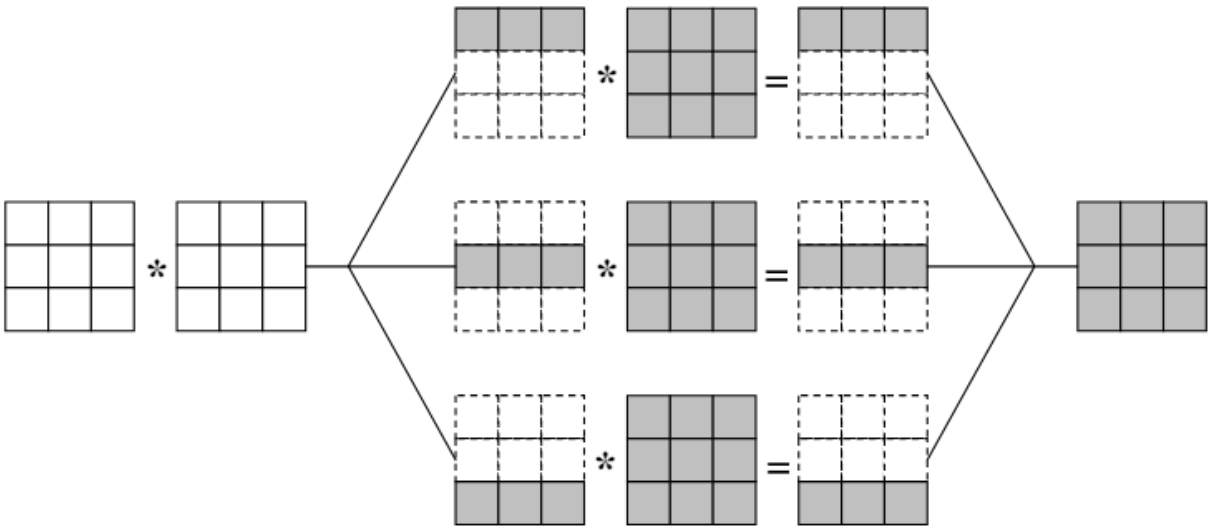

Рисунок 4.4 –Организация вычислений при выполнении параллельного алгоритма умножения матриц, основанного на разделении матриц по строкам

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.

Анализ эффективности

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 (доступк 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:

$$T_{calc} = \frac{n^2(2n-1)}{p} * \tau \qquad (4.7)$$

Для оценки объема данных, которые необходимо прочитать из оперативной памяти в кэш. 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. 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 матрицыA 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:

$$T_{mem} = \frac{8\,(n^3+\frac{8n^3}{p}+8\,n^2)}{\beta} \tag{4.8}$$

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

Следовательно, время выполнения параллельного алгоритма составляет:

$$T_p = \frac{n^2(2n-1)}{p} * \tau + \frac{8\,(n^3+\frac{8n^3}{p}+8\,n^2)}{\beta} \tag{4.9}$$

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:

$$\frac{8\,(n^3+\frac{8n^3}{p}+8\,n^2)}{\beta} \le T_p \le \frac{n^2(2n-1)}{p} * \tau + \frac{8\,(n^3+\frac{8n^3}{p}+8\,n^2)}{\beta} \tag{4.10}$$

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 parallel 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++)
```

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.

Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности параллельного алгоритма матричного умножения. Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных 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 таблице 4.3. Времена выполнения алгоритмов указаны в секундах.

Таблица 4.3. 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

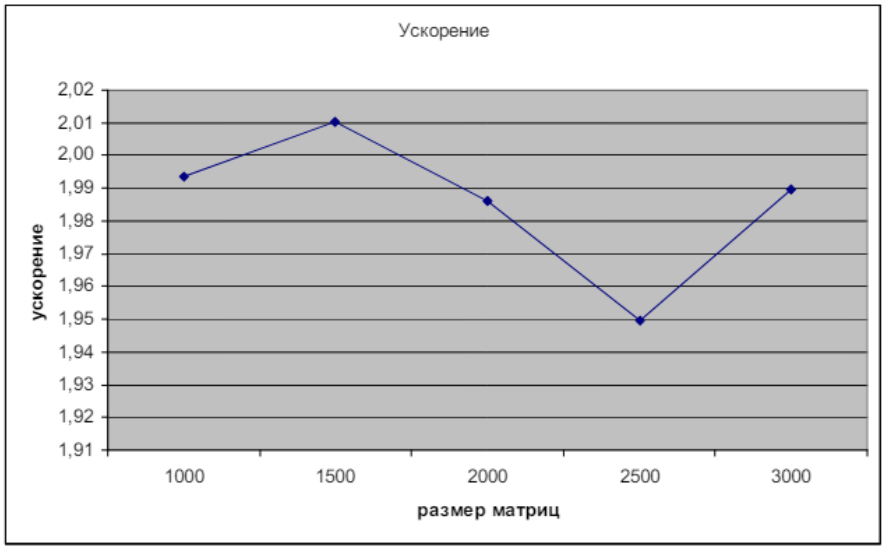

Рисунок 4.5 – Зависимость ускорения от количества исходных данных при выполнении базового параллельного алгоритма умножения матриц

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

В таблицах 4.4 и 4.5 и на рис. 4.6 и 4.7 представлены результаты сравнения времени выполнения T_p параллельного алгоритма умножения матриц с использованием двух и четырех потоков со временем T^*, полученным при помощи модели (4.4). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,3371, а для четырех потоков значение этой величины была оценена как 0,1832.

Таблица 4.4. Сравнение экспериментального и теоретического временивыполнения базового параллельного алгоритма умножения матриц с использованием двух потоков

Размер матриц	T_p	T^* (*calc*)*p*(модель)	Модель 4.4 – оценка сверху		Модель 4.5 – уточненная оценка	
			T^* (*mem*) *p*	T^* *p*	T^* (*mem*) *p*	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

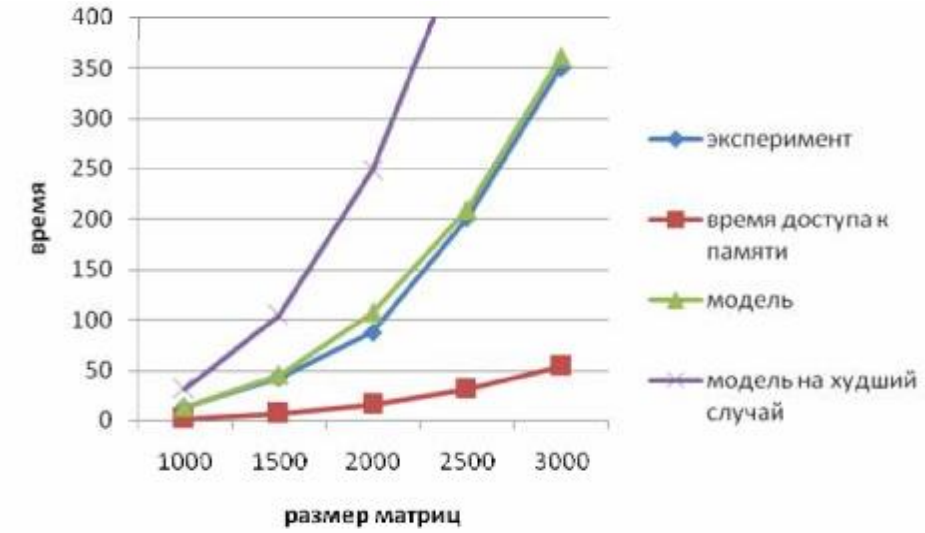

Рисунок 4.6–График зависимости экспериментального и теоретическоговремени выполнения базового параллельного алгоритма от объема исходных данных при использовании двух потоков