СУПЕРКОМПЬЮТЕРНЫЙ КОНСОРЦИУМ УНИВЕРСИТЕТОВ РОССИИ

Проект

Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения

Московский государственный университет им. М.В. Ломоносова

Нижегородский государственный университет им. Н.И. Лобачевского

- Национальный исследовательский университет -

Учебный курс

ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ МНОГОПРОЦЕССОРНЫХ МНОГОЯДЕРНЫХ СИСТЕМ

Лекция 7.

Параллельные методы матричного умножения

Гергель В.П., профессор, д.т.н.

Содержание

- Постановка задачи
- Последовательный алгоритм
- Базовый параллельный алгоритм умножения матриц
- Алгоритм умножения матриц, основанный на ленточном разделении данных
- Блочный алгоритм умножения матриц
- Блочный алгоритм, эффективно использующий кэш-память

Постановка задачи…

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

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

- Каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

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

Постановка задачи

- 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 x n количество выполненных операций имеет порядок $O(n^3)$.
- Далее будем предполагать, что все матрицы являются квадратными и имеют размер n x n.

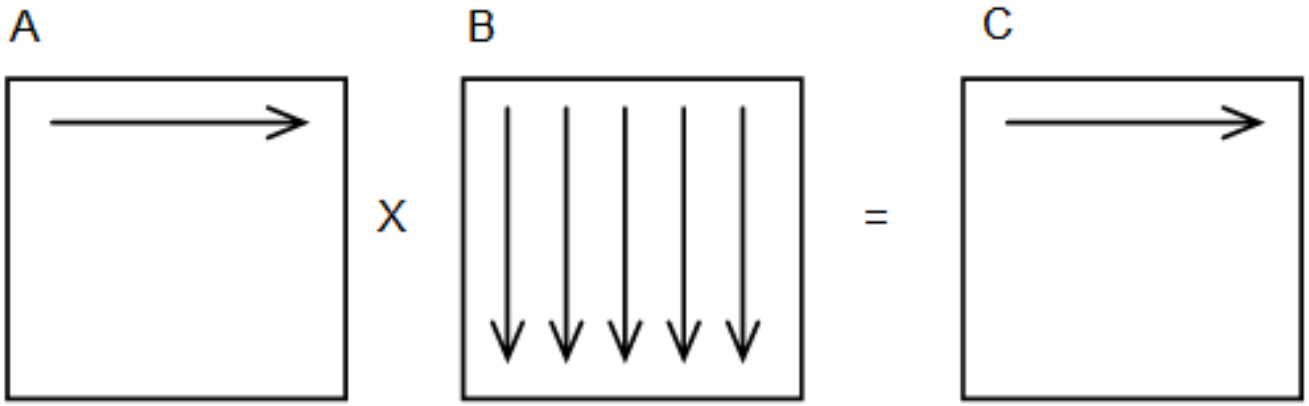
Последовательный алгоритм…

- Последовательный алгоритм умножения двух квадратных матриц

```
// Алгоритм 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];
    }
  }
}
```

Последовательный алгоритм…

- Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы *C*.
  - При выполнении одной итерации внешнего цикла вычисляется одна строка результирующей матрицы



Последовательный алгоритм…

- **Анализ эффективности.**
  - Время выполнения алгоритма складывается из времени, которое тратится непосредственно на *вычисления*, и времени, необходимого на *чтение данных* из оперативной памяти в кэш процессора.
  - Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы C размером n n затратить время:

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

Последовательный алгоритм…

- 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.
  - Оценки количества читаемых из памяти данных справедливы, если все эти данные отсутствуют в кэше.
- Всего необходимо вычислить n^2 элементов результирующей матрицы.
- Если при вычислении каждого очередного элемента требуется прочитать в кэш все необходимые данные, то общий объем данных, необходимых для чтения из оперативной памяти в кэш, не превышает величины $2n^3+n^2$.
- Данная оценка является оценкой сверху!

Последовательный алгоритм…

- Оценка времени выполнения последовательного алгоритма умножения матриц:

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

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

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

  - α есть латентность оперативной памяти.

Последовательный алгоритм…

- Обе построенные модели являются моделями на худший случай и дают сильно завышенную оценку времени выполнения алгоритма.
  - доступ к оперативной памяти происходит не при каждом обращении к элементу
- Как и ранее, введем в разработанную модель величину γ, $0 \le \gamma \le 1$, для задания *частоты возникновения кэш промахов*.

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

Последовательный алгоритм…

- **Программная реализация.**

  1. Главная функция программы.

```
// Программа 7.1
// 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);

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

  // Program termination
  ProcessTermination(pAMatrix, pBMatrix, pCMatrix, Size);
}
```

Последовательный алгоритм…

2. Функция ProcessInitialization.

```
// 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);

  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);
}
```

Последовательный алгоритм…

3. Функция SerialResultCaculation.

```
// 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];
}
```

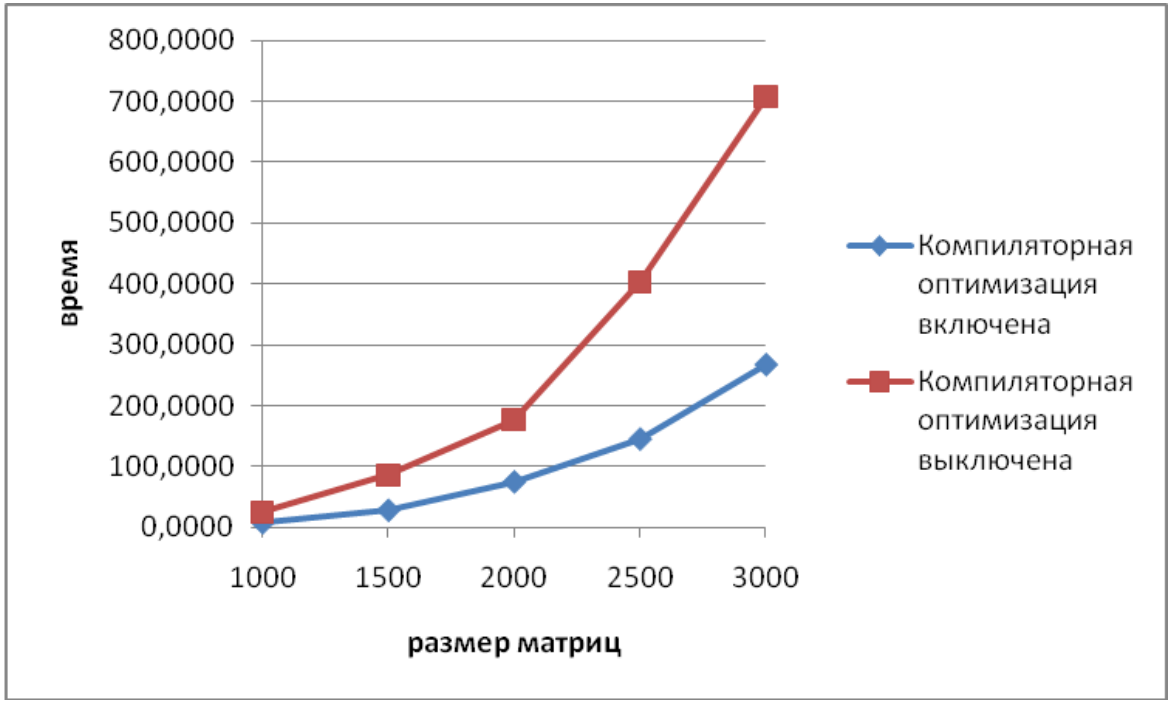
- приведенный программный код может быть оптимизирован (вычисление индексов, использование кэша и т.п.), однако такая оптимизация не является целью данного учебного материала и усложняет понимание программ.

Последовательный алгоритм…

- **Результаты вычислительных экспериментов.**
  - 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.
  - 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*.

Последовательный алгоритм…

– 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.



Последовательный алгоритм…

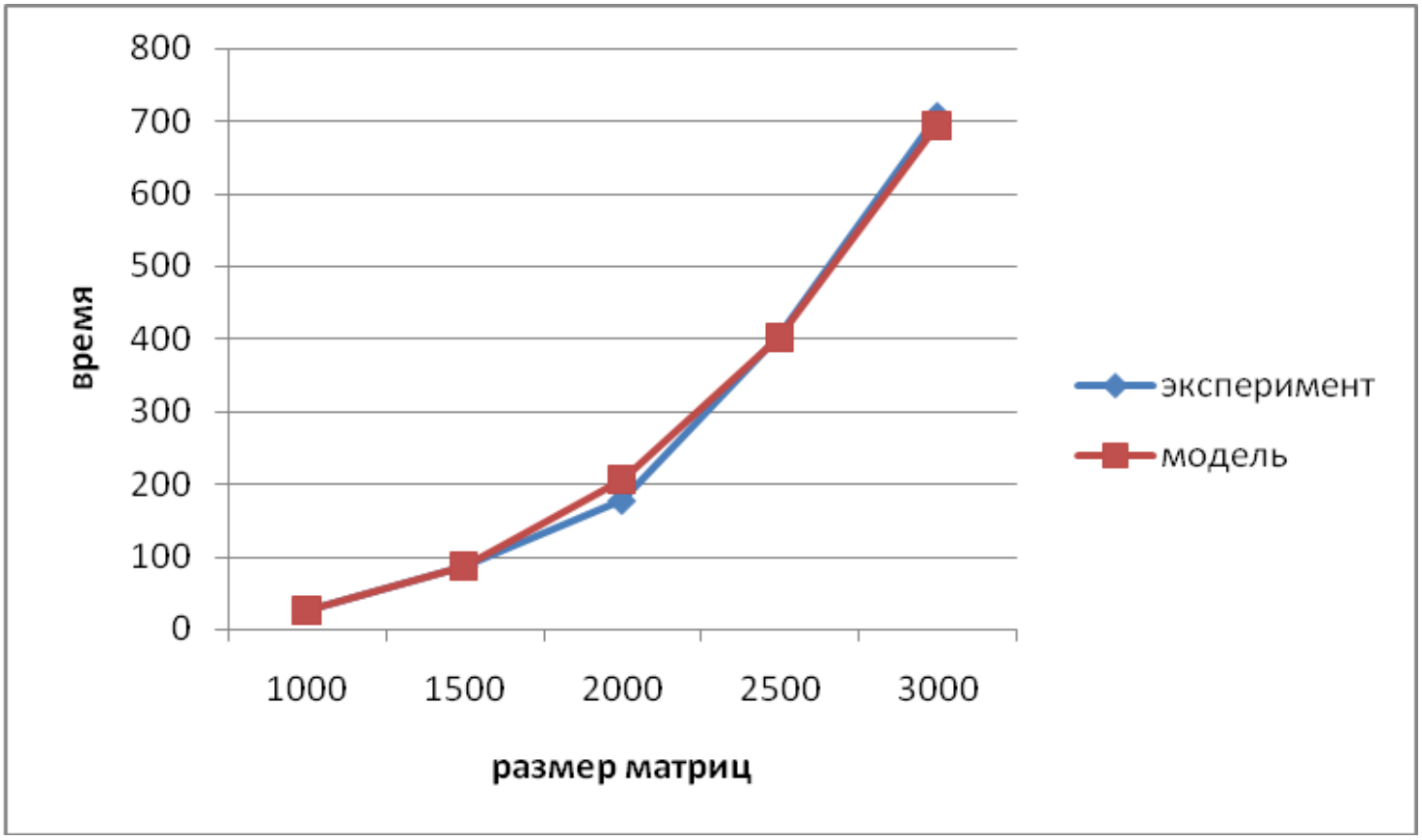
- Для того, чтобы оценить время одной операции τ, измерим время выполнения последовательного алгоритма умножения матриц при малых объемах данных.
- Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицы-аргументы случайными числами, а матрицу-результат нулями.
  - Выполнение этих действий гарантирует предварительное перемещение данных в кэш.
  - При решении задачи все время будет тратиться непосредственно на вычисления.
- Поделив время вычислений на количество выполненных операций, получим искомое время выполнения одной операции.
- Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение τ, равное 6,402 нс.

Последовательный алгоритм…

- Оценка времени латентности α и величины пропускной способности канала доступа к оперативной памяти β проводилась в лекции 6.
  - Для используемого вычислительного узла β = 12,44 Гб/с, а α = 8,31 нс.
- Частота кэш промахов, измеренная с помощью системы VPS оказалась равной 0,505.

Последовательный алгоритм

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



Базовый параллельный алгоритм умножения матриц …

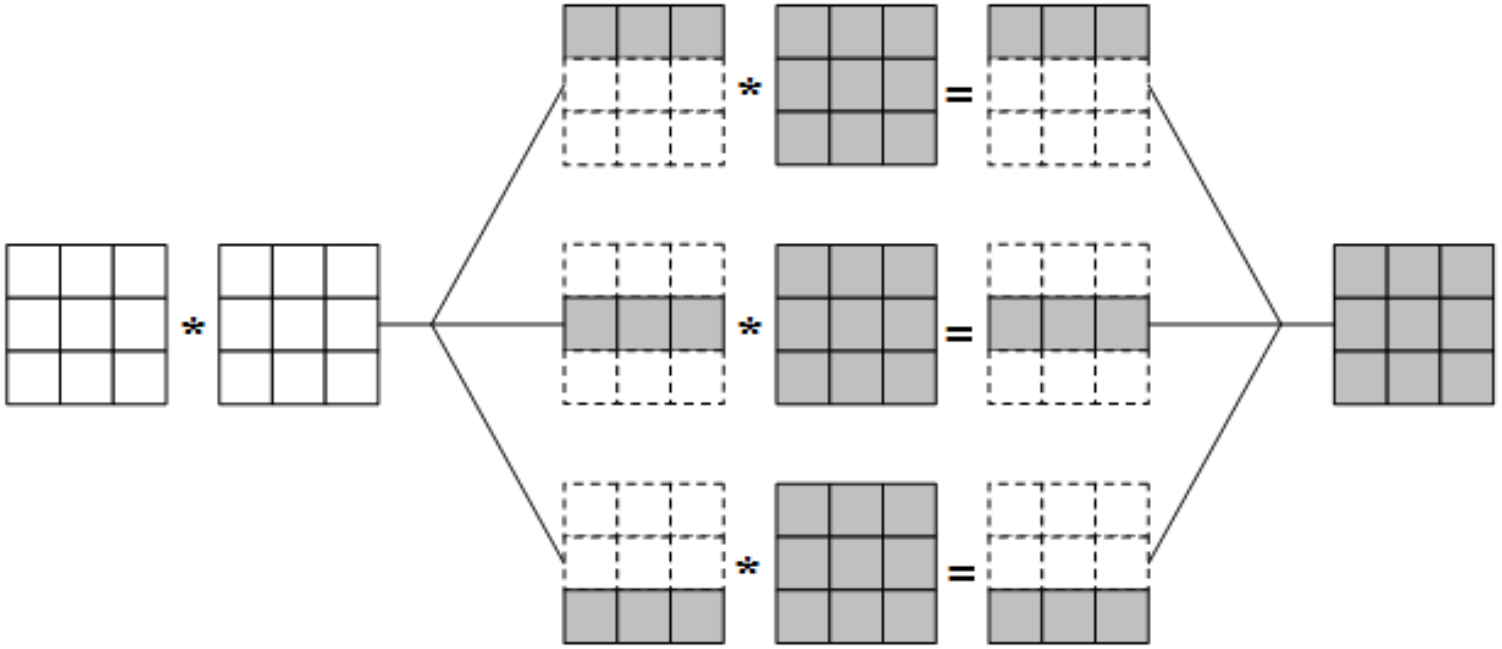
- **Определение подзадач.**
  - Из определения операции матричного умножения следует, что вычисление всех элементов матрицы C может быть выполнено независимо друг от друга.
  - Как результат, возможный подход для организации параллельных вычислений состоит в использовании в качестве базовой подзадачи 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).

Базовый параллельный алгоритм умножения матриц …

- 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*.
- Для дальнейшего рассмотрения в рамках данной лекции 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.
  - Реального дублирования данных нет, так как разрабатываемый алгоритм ориентирован на применение для вычислительных систем с общей разделяемой памятью.

Базовый параллельный алгоритм умножения матриц …

- **Выделение информационных зависимостей.**
  - Организация вычислений при выполнении параллельного алгоритма умножения матриц, основанного на разделении матриц по строкам



Базовый параллельный алгоритм умножения матриц …

- **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 следует выбрать равным $k=n/p$ (в предположении, что n кратно p).

Базовый параллельный алгоритм умножения матриц …

- **Анализ эффективности.**
  - Данный параллельный алгоритм обладает хорошей «локальностью вычислений».
    - Данные, которые обрабатывает один из потоков параллельной программы, не изменяются другим потоком.
  - Нет взаимодействия между потоками, нет необходимости в синхронизации.
  - Следовательно, время выполнения параллельного алгоритма, равно сумме времени потраченного на вычисления и на чтение данных из оперативной памяти в кэш процессора.

Базовый параллельный алгоритм умножения матриц …

- 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:

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

Базовый параллельный алгоритм умножения матриц …

- Для вычисления одного элемента результирующей матрицы C необходимо прочитать в кэш $n+8n+8$ элементов данных.
- Каждый поток вычисляет n/p элементов матрицы C.
  - 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.
- Чтение строк матрицы A и элементов матрицы C в предельном случае должно быть выполнено полностью и последовательно.
- Следовательно, время работы с оперативной памятью:

$$T_{mem} = \left(n^3 + n^3/p + n^2 \right) \left(\alpha + 64/\beta \right)$$

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

Базовый параллельный алгоритм умножения матриц …

– Следовательно, время выполнения параллельного алгоритма составляет:

$$T_p = (n^2/p)\left(2n-1\right)\cdot\tau + \left(2n^3+n^3/p+n^2\right)\left(\alpha+64/\beta\right)$$

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

– Итого, модельное время вычислений:

$$T_p = (n^2/p)\left(2n-1\right)\cdot\tau + \gamma\left(2n^3+n^3/p+n^2\right)\left(\alpha+64/\beta\right)$$

Базовый параллельный алгоритм умножения матриц …

- **Программная реализация.**
  - Для того, чтобы разработать параллельную программу, реализующую описанный подход, при помощи технологии OpenMP, достаточно добавить одну директиву *parallel for* в функции *SerialResultCaclualtion* (назовем новый вариант функции *ParallelResultCaclualtion*):

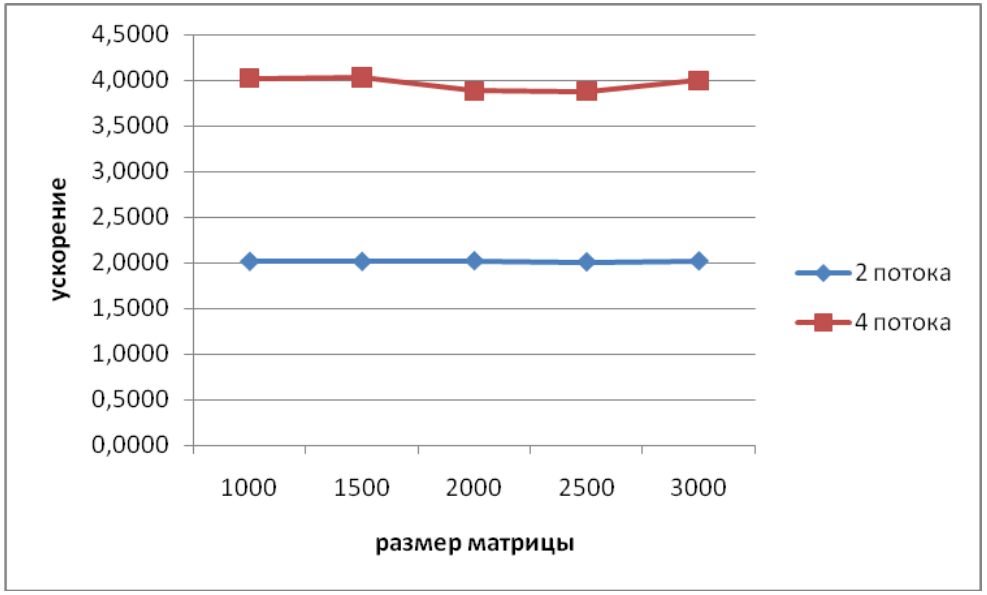
```
// Программа 7.2
// Function for parallel calculating matrix multiplication
void ParallelResultCalculation(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size) {
  int i, j, k;  // Loop variables
#pragma omp paralell for private (j, k)
  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++)
      for (k=0; k<Size; k++)
        pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
}
```

Базовый параллельный алгоритм умножения матриц …

- **Результаты вычислительных экспериментов.**
  - 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.

Базовый параллельный алгоритм умножения матриц …

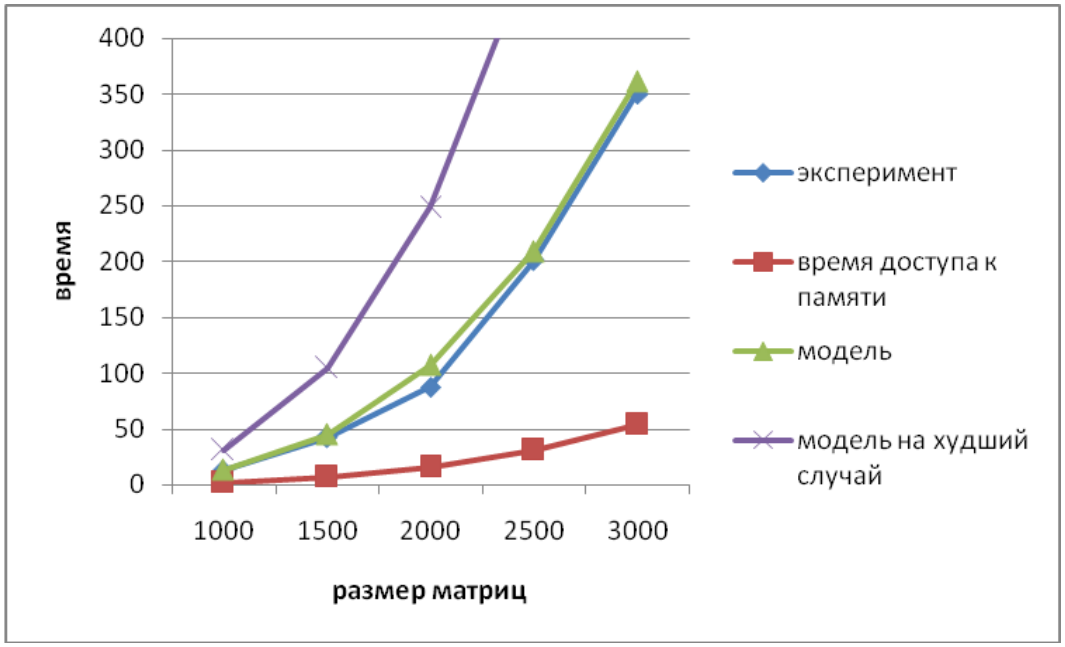
– Зависимость ускорения от количества исходных данных при выполнении базового параллельного алгоритма умножения матриц



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

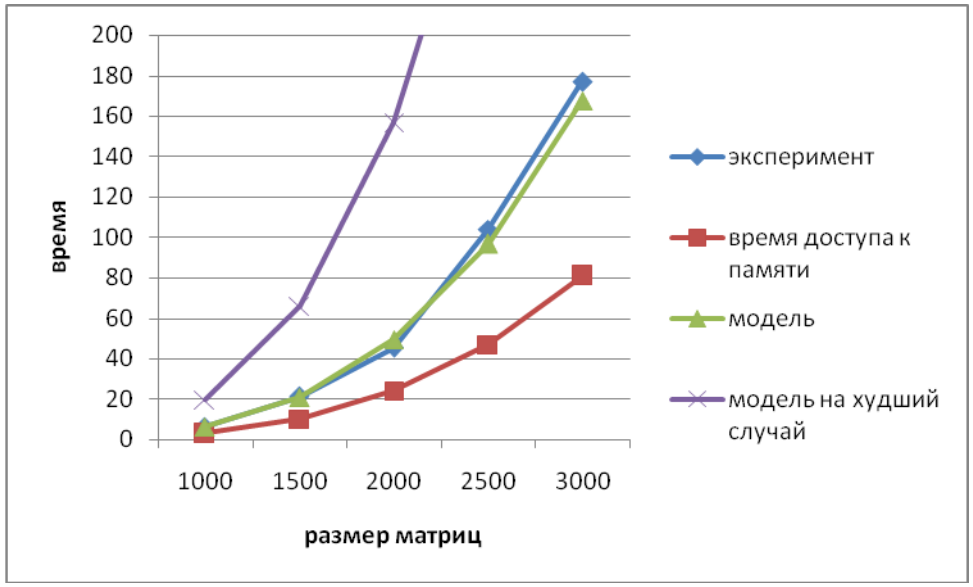
Базовый параллельный алгоритм умножения матриц …

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



Базовый параллельный алгоритм умножения матриц

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



Алгоритм умножения матриц, основанный на ленточном разделении данных …

- **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.
  - Для укрупнения базовой подзадачи, определим базовую подзадачу как процедуру вычисления всех элементов прямоугольного блока матрицы C.

Алгоритм умножения матриц, основанный на ленточном разделении данных …

- 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 ($\pi = q^2$).
- Для эффективного выполнения вычислений количество потоков π должно быть, по крайней мере, кратным числу вычислительных элементов (процессоров и/или ядер) p.

Алгоритм умножения матриц, основанный на ленточном разделении данных …

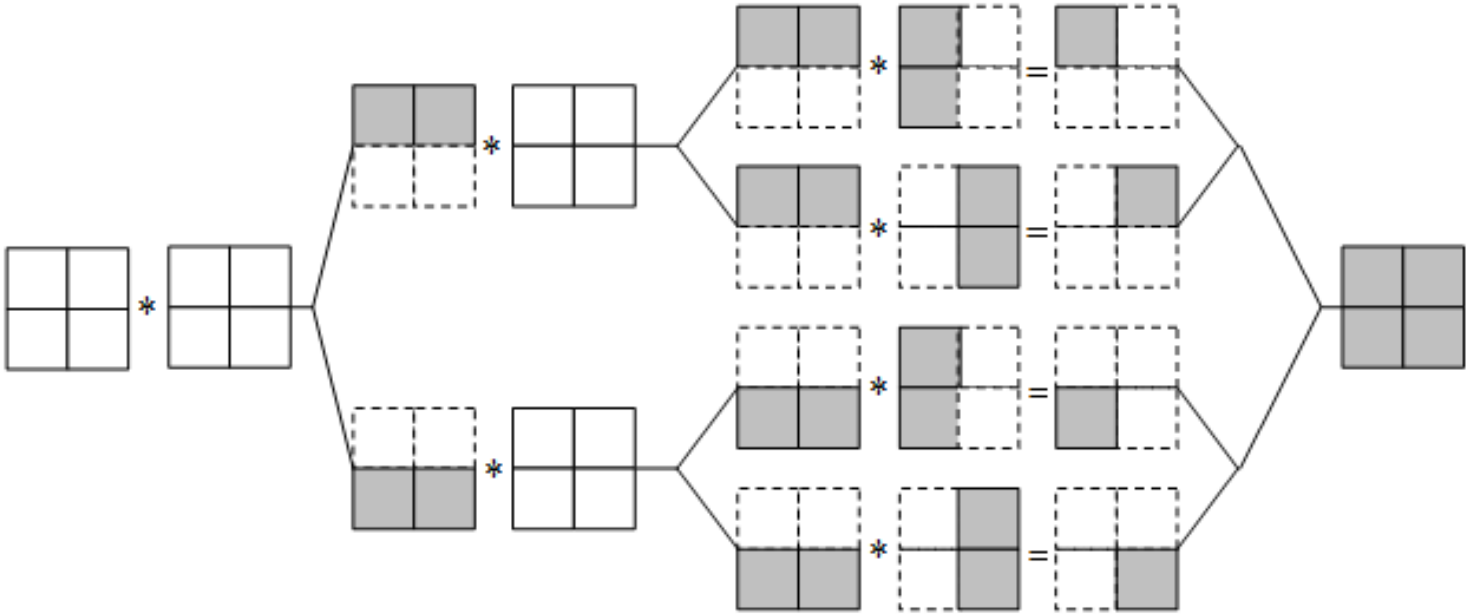
- **Выделение информационных зависимостей.**
  - Для вычисления всех элементов прямоугольного блока результирующей матрицы

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

    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.

Алгоритм умножения матриц, основанный на ленточном разделении данных …

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



Алгоритм умножения матриц, основанный на ленточном разделении данных …

- **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 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.

Алгоритм умножения матриц, основанный на ленточном разделении данных …

- **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.

$$n^2 \cdot (2n-1)/\pi$$

  - Следовательно время вычислений:

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

Алгоритм умножения матриц, основанный на ленточном разделении данных …

- Каждый поток выполняет умножение горизонтальной полосы матрицы A на вертикальную полосу матрицы B для того, чтобы получить прямоугольный блок результирующей матрицы.
- 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.
  - для определения полного объема переписываемых в кэш данных следует учитывать, что чтение значений из оперативной памяти может выполняться только последовательно.
- 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.

Алгоритм умножения матриц, основанный на ленточном разделении данных …

– Следовательно, время, необходимое на чтение необходимых данных из оперативной памяти составляет:

$$T_{mem} = \left(n^3 + n^3 / q + n^2 \right) \left(\alpha + 64 / \beta \right)$$

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

– При выполнении представленного алгоритма, реализованного с помощью вложенного параллелизма, «внутренние» параллельные секции создаются и закрываются n/q раз.

– 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.

– Величину накладных расходов на организацию и закрытие одной параллельной секции обозначим через δ.

Алгоритм умножения матриц, основанный на ленточном разделении данных …

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

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

– Если учесть частоту кэш промахов γ, выражение принимает вид:

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

Алгоритм умножения матриц, основанный на ленточном разделении данных …

- **Программная реализация.**
  - Использование механизма вложенного параллелизма OpenMP позволяет существенно упростить реализацию алгоритма.
  - На данный момент не все компиляторы, реализующие стандарт OpenMP, поддерживают вложенный параллелизм.
  - Для компиляции кода использовался компилятор Intel C++ Compiler 10.0 for Windows, поддерживающий вложенный параллелизм.
  - Для того, чтобы разработать параллельную программу, реализующую описанный подход, прежде всего, необходимо включить поддержку вложенного параллелизма при помощи вызова функции *omp_set_nested*.

Алгоритм умножения матриц, основанный на ленточном разделении данных …

```
// Программа 7.3
// Function for parallel 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];
}
```

Алгоритм умножения матриц, основанный на ленточном разделении данных …

- **Результаты вычислительных экспериментов.**
  - Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма умножения матриц, основанного на ленточном разделении матриц

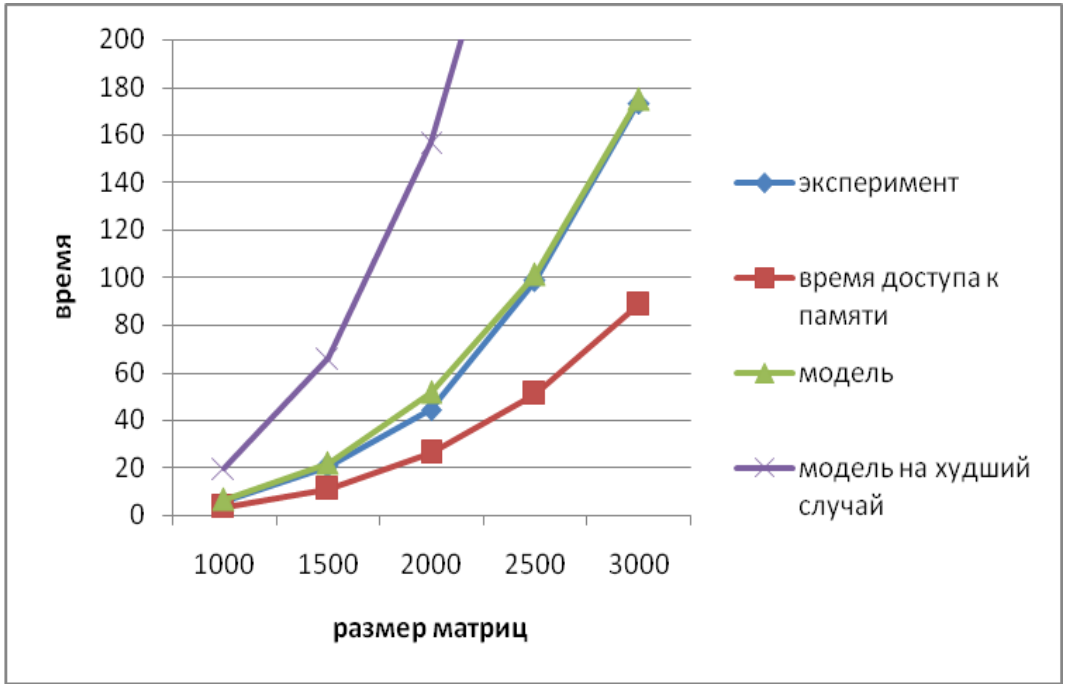
Алгоритм умножения матриц, основанный на ленточном разделении данных …

– Для того, чтобы оценить величину накладных расходов на параллелизм, разработаем еще одну реализацию параллельного алгоритма умножения матриц.

```
// Программа 7.4
// Function for parallel calculating matrix multiplication
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)];
  }
}
```

Алгоритм умножения матриц, основанный на ленточном разделении данных

- Как и в лекции 6, время δ, необходимое на организацию и закрытие параллельной секции, примерно равно 0,25 мкс.
- График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма, основанного на ленточном разделении данных, от объема исходных данных при использовании четырех потоков



Блочный алгоритм умножения матриц …

- **Определение подзадач.**
  - При блочном способе разделения данных исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков.
  - Будем предполагать далее, что все матрицы являются квадратными размера n n, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен k k, $k=n/q$).

Блочный алгоритм умножения матриц …

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

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

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

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj}$$

– При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками.

  • определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы C.

Блочный алгоритм умножения матриц …

- 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 неизбежно приведет к дублированию 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.

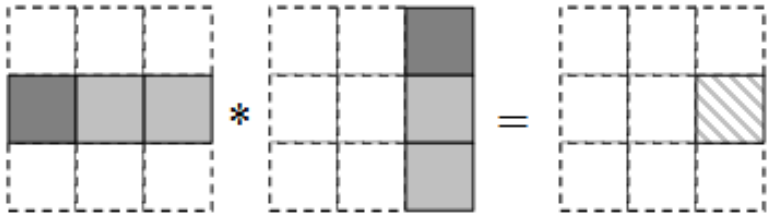
Блочный алгоритм умножения матриц …

- К числу алгоритмов, реализующих описанный подход, относятся *алгоритм Фокса* (*Fox*) и *алгоритм Кэннона* (*Cannon*).
- 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.

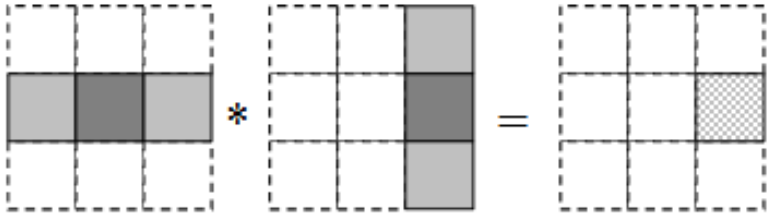
Блочный алгоритм умножения матриц …

- **Выделение информационных зависимостей.**
  - Схема организации блочного умножения полос

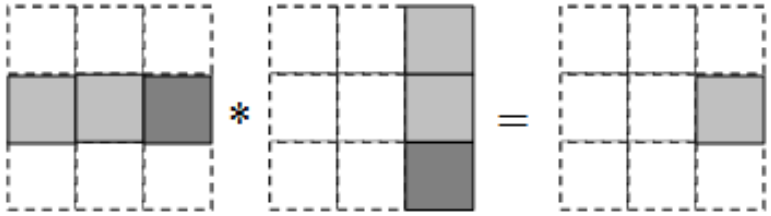
Итерация 1



Итерация 2



Итерация 3



Блочный алгоритм умножения матриц …

- **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.
  - В наиболее простом случае, когда число вычислительных элементов представимо в виде $p=\sigma^2$ можно выбрать количество блоков в матрицах по вертикали и горизонтали равным σ (т.е. $q=\sigma$).
  - В данном случае, 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.

Блочный алгоритм умножения матриц …

- **Анализ эффективности.**
  - 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.
  - Следовательно, вычислительная сложность блочного алгоритма составляет:

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

  - На каждой итерации алгоритма каждый поток выполняет умножение двух матричных блоков размером *(n/q) (n/q)*.
  - Объем данных, загружаемый из оперативной памяти в кэш, с поправкой на размер матричного блока, может быть получен из модели последовательного алгоритма.

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

Блочный алгоритм умножения матриц …

- Для вычисления блока результирующей матрицы каждый поток выполняет q итераций.
- 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:

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

- При более подробном анализе алгоритма можно заметить, что блок, который считывается в кэш одним из потоков, одновременно используется и другими (q-1) потоками, следовательно:

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

Блочный алгоритм умножения матриц …

– Общее время выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении данных в худшем случае, определяется соотношением:

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

– Если учесть частота кэш промахов, то выражение принимает вид:

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

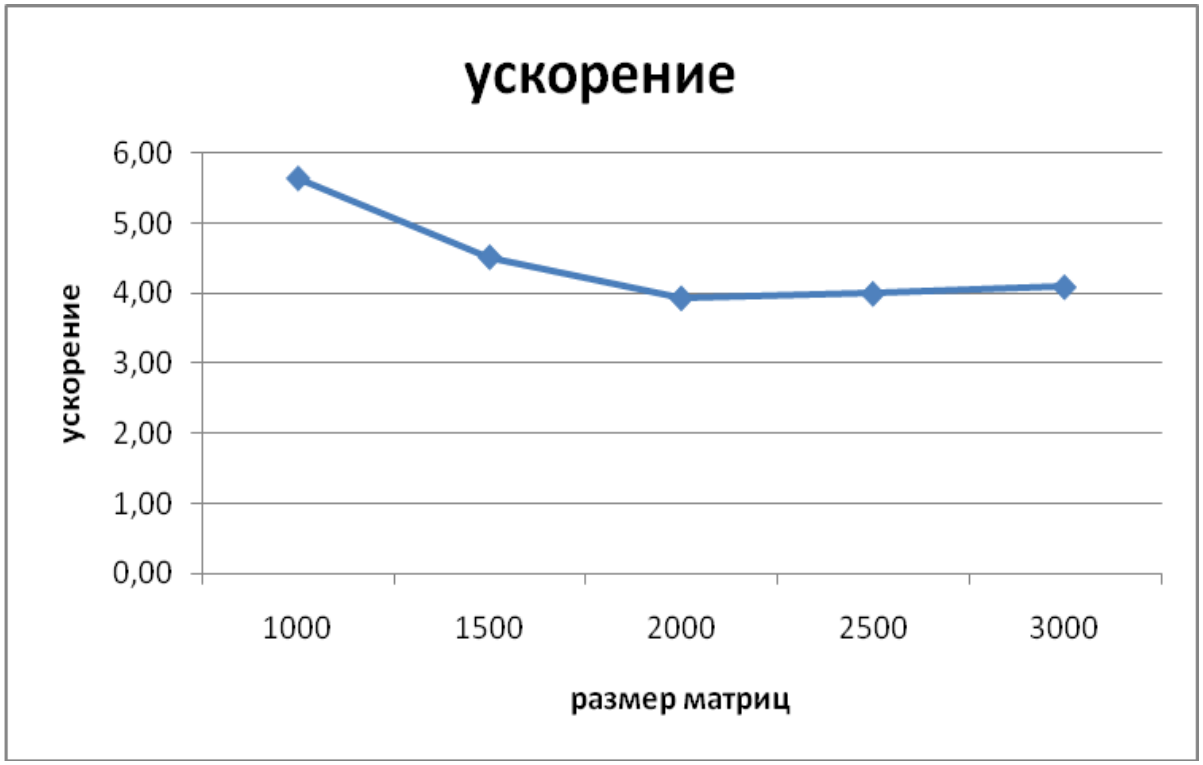
Блочный алгоритм умножения матриц …

- **Программная реализация.**

```
// Программа 7.5
// Function for parallel calculating matrix multiplication
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
}
```

Блочный алгоритм умножения матриц …

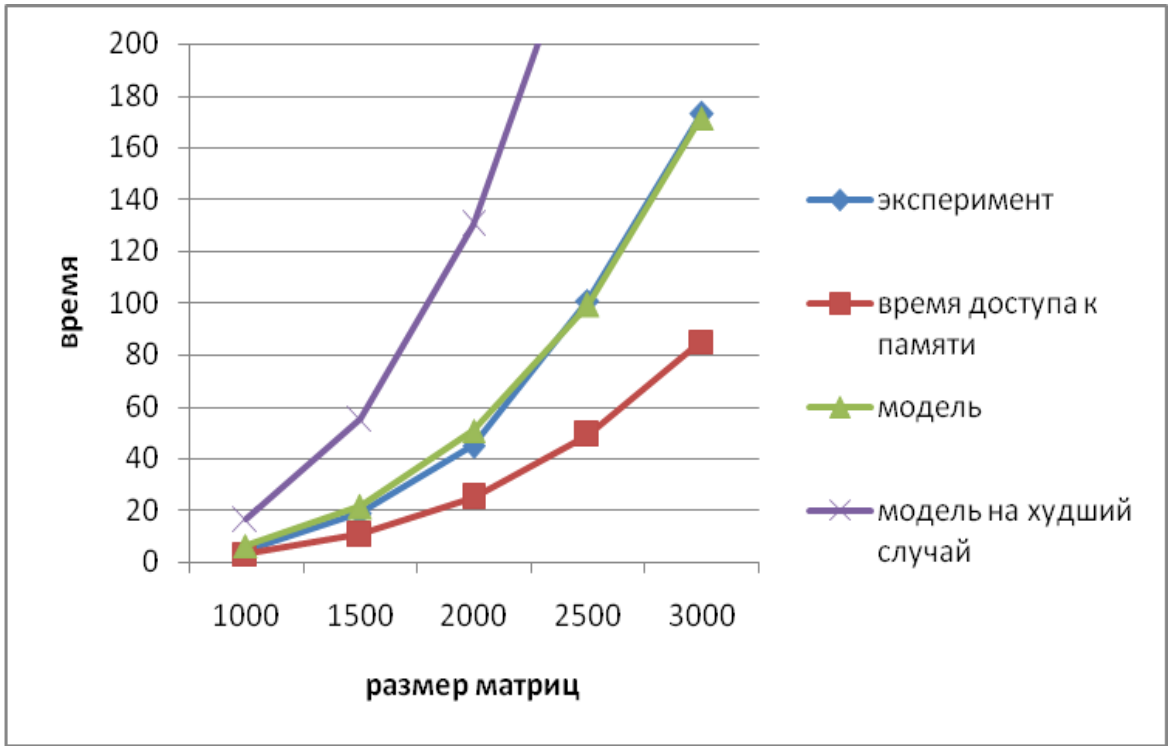
- **Результаты вычислительных экспериментов.**
  - Зависимость ускорения от количества исходных данных при выполнении параллельного блочного алгоритма умножения матриц



Блочный алгоритм умножения матриц

– Частота кэш промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,24.

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



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.
- Для вычисления первого элемента строки результирующей матрицы необходимо прочитать в кэш все элементы первой строки матрицы *A* и первого столбца матрицы *B*.
- 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.

Блочный алгоритм, эффективно использующий кэш-память …

- Полоса матрицы *B* может быть настолько велика, что чтение «последних» элементов этой полосы ведет к вытеснению из кэш «первых» элементов строки матрицы *A*.
- 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.
- Итого, если размер матриц составляет *n* *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.

Блочный алгоритм, эффективно использующий кэш-память …

- **Последовательный алгоритм.**
  - Для организации алгоритма умножения матриц, который более эффективно использует кэш-память, воспользуемся разделением матриц на блоки.
  - Количество разбиений матриц на блоки должно быть таким, чтобы в кэш одновременно могли быть помещены три матричных блока – блоки матриц A, B и C.

Блочный алгоритм, эффективно использующий кэш-память …

- Пусть V_{cache} – объем кэш в байтах.
- Тогда количество элементов типа double, которые могут быть помещены в кэш, составляет *V/8*.
  - Для хранения одного элемента типа double используется 8 байт.
- Максимальный размер квадратного k 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 n составляет:

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

- 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.

Блочный алгоритм, эффективно использующий кэш-память …

– Рассмотрим программную реализацию последовательного блочного алгоритма матричного умножения.

```
// Программа 7.6
// 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];
}
```

Блочный алгоритм, эффективно использующий кэш-память …

- **Параллельный алгоритм**
  - 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.
    - Объем вычислительных операций не изменяется.
  - Вычислительная сложность блочного алгоритма составляет:

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

Блочный алгоритм, эффективно использующий кэш-память …

– 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.
– Размер блоков определяется таким образом, чтобы на каждой итерации они могли быть одновременно помещены в кэш память. Следовательно, время, необходимое на чтение данных из оперативной памяти в кэш может быть вычислено по формуле:

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

Блочный алгоритм, эффективно использующий кэш-память …

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

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

– Учитывая частоту кэш промахов, выражение приобретает следующий вид:

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

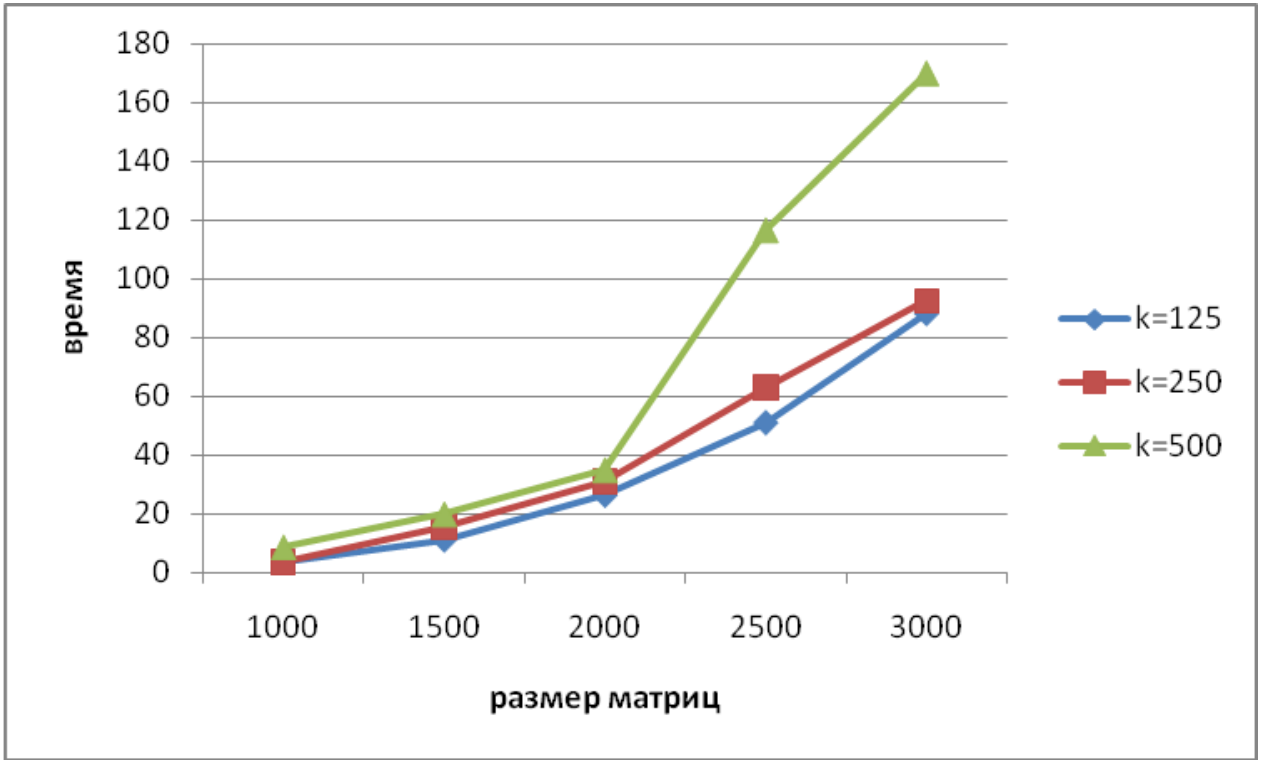
Блочный алгоритм, эффективно использующий кэш-память …

– Программная реализация описанного подхода может быть получена следующим образом:

```
// Программа 7.7
// 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];
}
```

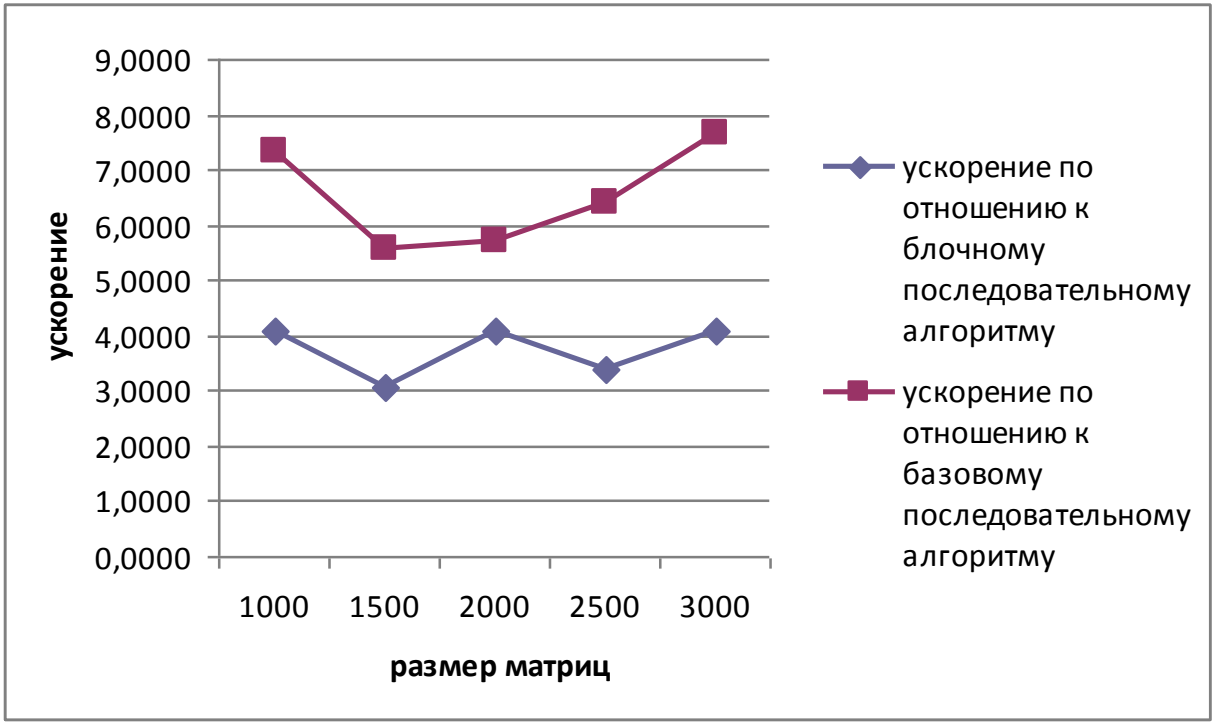
Блочный алгоритм, эффективно использующий кэш-память …

- **Результаты вычислительных экспериментов.**
  - Время выполнения параллельного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, при разных размерах матричных блоков



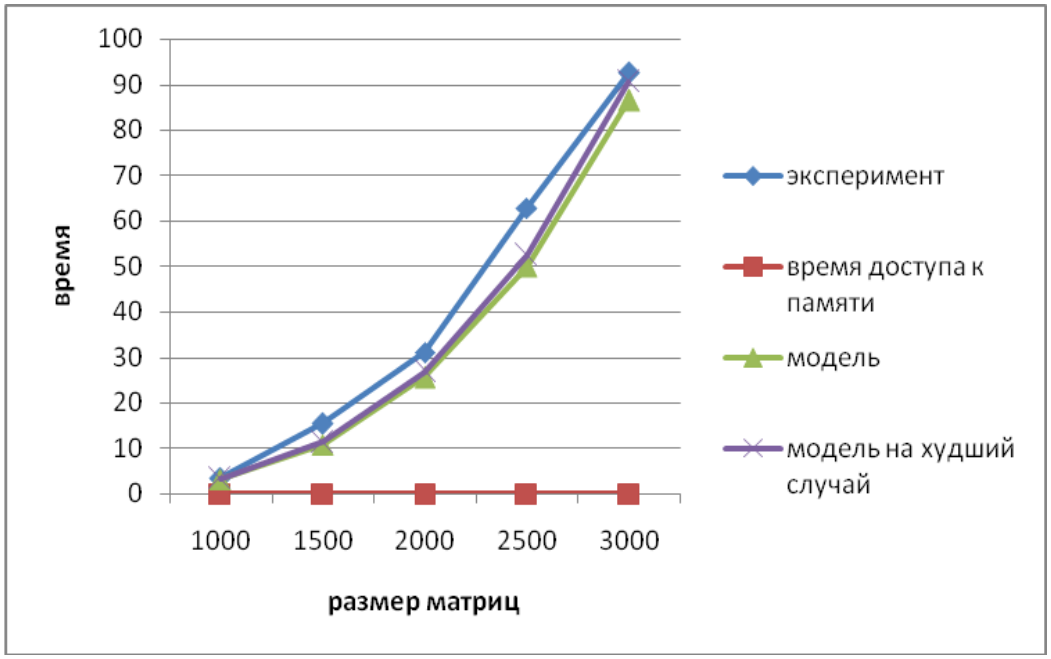
Блочный алгоритм, эффективно использующий кэш-память …

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



Блочный алгоритм, эффективно использующий кэш-память

- Частота кэш промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,0026.
- График зависимости экспериментального и теоретического времени выполнения параллельного блочного алгоритма, ориентированного на эффективное использование кэш-памяти, от объема исходных данных при использовании четырех потоков

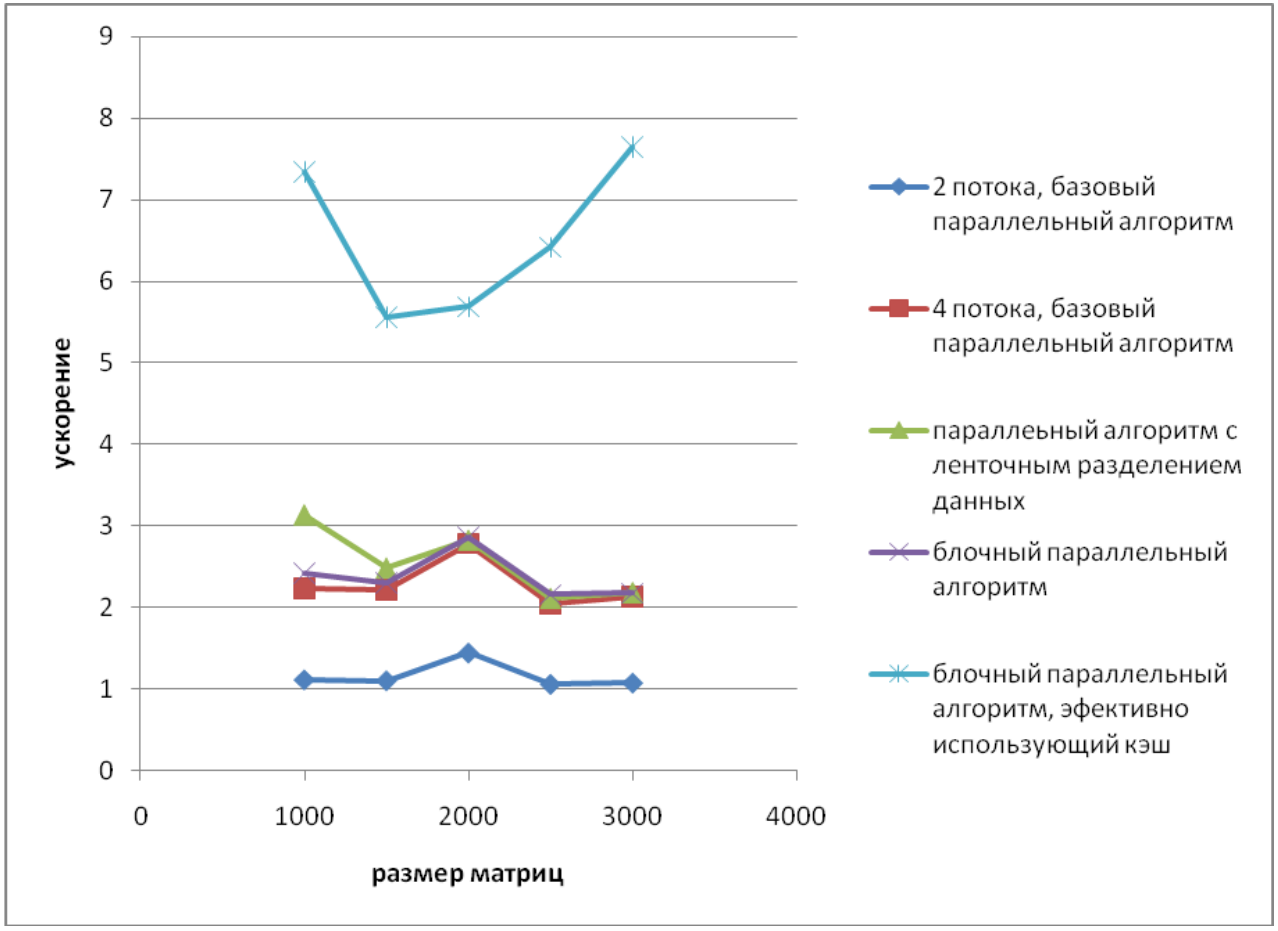


Заключение…

- 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 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.

Заключение

- Ускорение вычислений при матричном умножении для всех четырех рассмотренных в разделе параллельных алгоритмов



Вопросы для обсуждения

- В чем состоит постановка задачи умножения матриц?
- Приведите примеры различных последовательных алгоритмов выполнения операции умножения матриц. Отличается ли их вычислительная трудоемкость?
- Какие способы разделения данных используются при разработке параллельных алгоритмов матричного умножения?
- Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных?
- Какие информационные взаимодействия выполняются для блочных алгоритмов умножения матриц?
- Какой из рассмотренных алгоритмов обладает наилучшими показателями ускорения и эффективности?
- Оцените возможность выполнения матричного умножения как последовательности операций умножения матрицы на вектор.
- Какие функции библиотеки OpenMP оказались необходимыми при программной реализации параллельных алгоритмов?

Темы заданий для самостоятельной работы

- Выполните реализацию блочных алгоритмов умножения матриц, которые могли бы быть выполнены для прямоугольных решеток потоков общего вида.
- Выполните реализацию матричного умножения с использованием ранее разработанных программ умножения матрицы на вектор.

Обзор литературы…

- Задача умножения матриц широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы
  - Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
  - Гергель В.П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
  - Kumar V., Grama A., Gupta A., Karypis G. Introduction to Parallel Computing , Inc. 1994 (2th edition, 2003)
  - Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004

Обзор литературы

- Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе
  - Dongarra J.J., Duff L.S., Sorensen D.C., Vorst H.A.V. Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc. For Industrial & Applied Math., 1999
- При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа
  - Blackford L.S., Choi J., Cleary A., D'Azevedo E., Demmel J., Dhillon I., Dongarra J.J., Hammarling S., Henry G., Petitet A., Stanley D., Walker R.C., Whaley K. Scalapack Users'Guide (Software, Environments, Tools). Soc. For Industrial & Applied Math., 1997

Следующая тема

- Параллельные методы решения систем линейных уравнений

О проекте

Целью проекта является создание национальной системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения.

Задачами по проекту являются:

Задача 1. Создание сети научно-образовательных центров суперкомпьютерных технологий (НОЦ СКТ).

Задача 2. Разработка учебно-методического обеспечения системы подготовки, переподготовки и повышения квалификации кадров в области суперкомпьютерных технологий.

Задача 3. Реализация образовательных программ подготовки, переподготовки и повышения квалификации кадров в области суперкомпьютерных технологий.

Задача 4. Развитие интеграции фундаментальных и прикладных исследований и образования в области суперкомпьютерных технологий. Обеспечение взаимодействия с РАН, промышленностью, бизнесом.

Задача 5. Расширение международного сотрудничества в создании системы суперкомпьютерного образования.

Задача 6. Разработка и реализации системы информационного обеспечения общества о достижениях в области суперкомпьютерных технологий.

См. http://www.hpc-education.ru