Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Образовательный комплекс

Введение в методы параллельного программирования

Раздел 8.

Параллельные методы матричного умножения

Гергель В.П., профессор, д.т.н.
Кафедра математического обеспечения ЭВМ

Содержание

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

Постановка задачи

Умножение матриц:

$$C = A \cdot B$$

или

$$\begin{pmatrix} c_{0,0}, & c_{0,1}, & \ldots, & c_{0,l-1} \\ & & \ldots & \\ c_{m-1,0}, & c_{m-1,1}, & \ldots, & c_{m-1,l-1} \end{pmatrix} = \begin{pmatrix} a_{0,0}, & a_{0,1}, & \ldots, & a_{0,n-1} \\ & & \ldots & \\ a_{m-1,0}, & a_{m-1,1}, & \ldots, & a_{m-1,n-1} \end{pmatrix} \begin{pmatrix} b_{0,0}, & b_{0,1}, & \ldots, & a_{0,l-1} \\ & & \ldots & \\ b_{n-1,0}, & b_{n-1,1}, & \ldots, & b_{n-1,l-1} \end{pmatrix}$$

↳ Задача умножения матриц может быть сведена к выполнению ***m·n*** независимых операций умножения строк матрицы ***A*** на столбцы матрицы ***B***

$$c_{ij} = \left(a_i, b_j^T\right) = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l$$

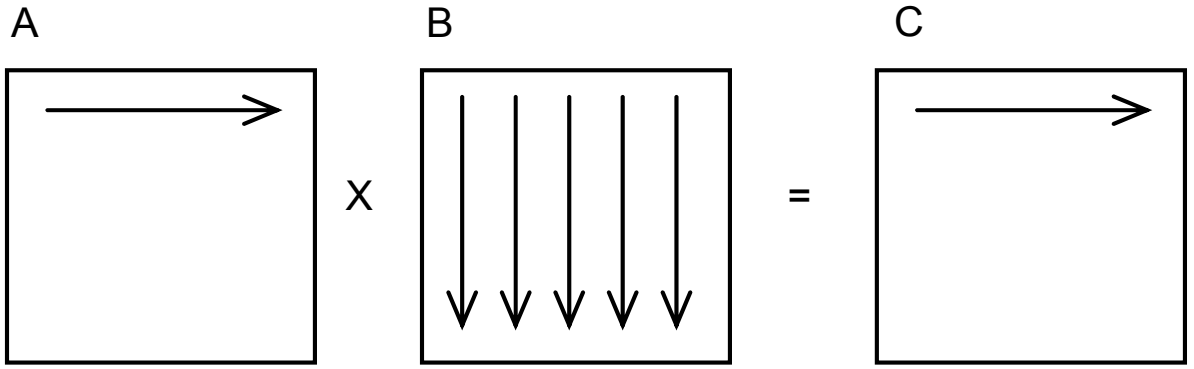
В основу организации параллельных вычислений может быть положен ***принцип распараллеливания по данным***

Последовательный алгоритм…

```
// Алгоритм 8.1
// Последовательный алгоритм матричного умножения
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…

- Алгоритм осуществляет последовательное вычисление строк матрицы ***C***
- На одной итерации цикла по переменной ***i*** используется первая строка матрицы ***A*** и все столбцы матрицы ***B***



- Для выполнения матрично-векторного умножения необходимо выполнить ***m·n*** операций вычисления скалярного произведения
- Трудоемкость вычислений имеет порядок *O(mnl)*.

Последовательный алгоритм…

❑ Анализ эффективности

– Время, которое тратится непосредственно на вычисления (матрицы размером *n*×*n*):

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

– Объем данных, которые считываются из оперативной памяти в кэш процессора:

$$V = 8 \cdot \left(8 \cdot n^2 + n^3 + 8 \cdot n^3\right)$$

– Время, необходимое на считывание данных из оперативной памяти

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

– Время выполнения последовательного алгоритма:

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

Последовательный алгоритм…

❑ Программная реализация…

```
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);
}
```

Последовательный алгоритм…

- **Программная реализация…**
  - Функция *ProcessInitialization* определяет размер матриц и элементы для матриц *A* и *B*, результирующая матрица *C* заполняется нулями. Значения элементов для матриц *A* и *B* определяются в функции *RandomDataInitialization*,
  - Функция *ProcessTermination* освобождает память, выделенную динамически в процессе выполнения алгоритма.

Последовательный алгоритм…

- **Программная реализация**
  - Функция *SerialResultCalculation* выполняет последовательный алгоритм умножения 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];
}
```

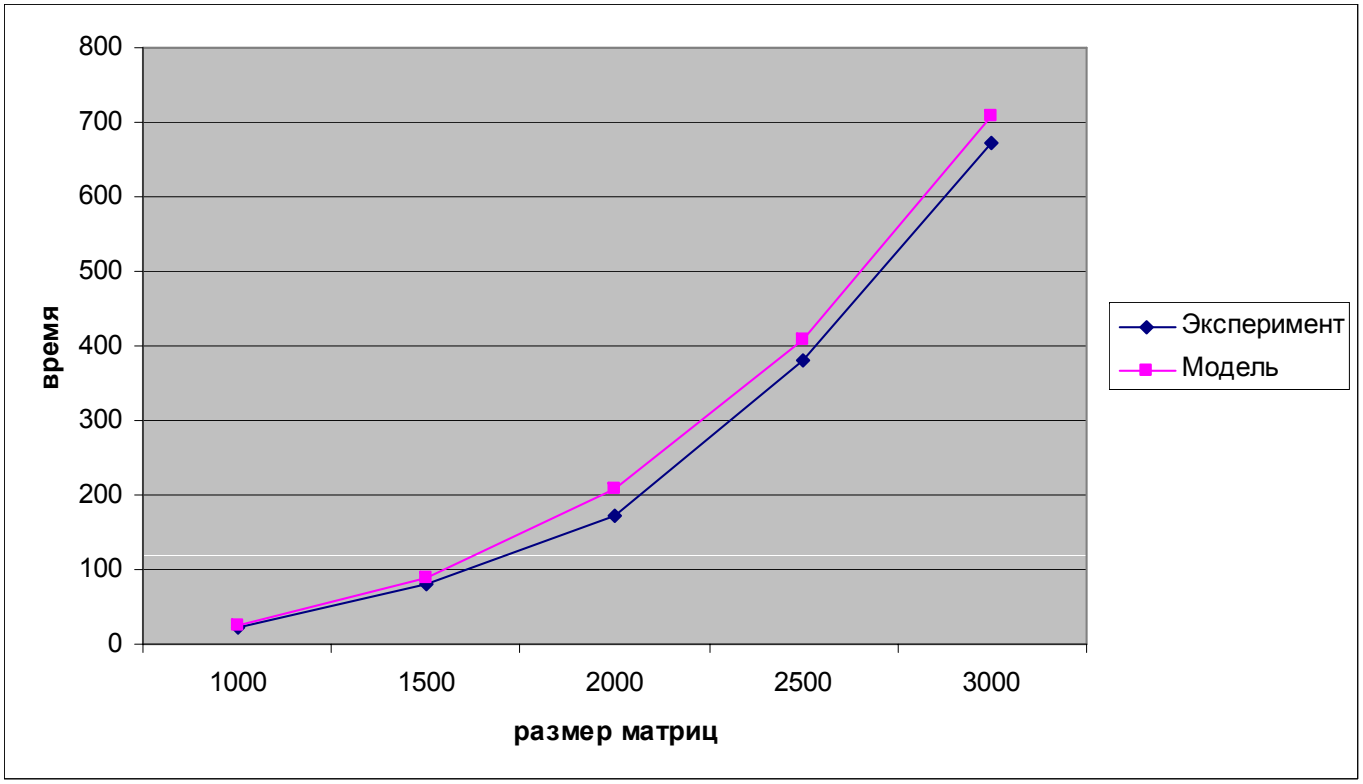
Последовательный алгоритм…

- **Результаты вычислительных экспериментов…**
  - Сравнение теоретических оценок и экспериментальных данных:

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

Последовательный алгоритм

- **Результаты вычислительных экспериментов…**
  - Сравнение теоретических оценок и экспериментальных данных:



Базовый параллельный алгоритм умножения матриц…

- **Возможный подход** – в качестве базовой подзадачи выбирается процедура вычисления одного из элементов матрицы ***C***

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

- Достигнутый уровень параллелизма является избыточным (количество базовых подзадач равно n^2)!
- Как правило $p<n^2$ и необходимым является масштабирование параллельных вычислений

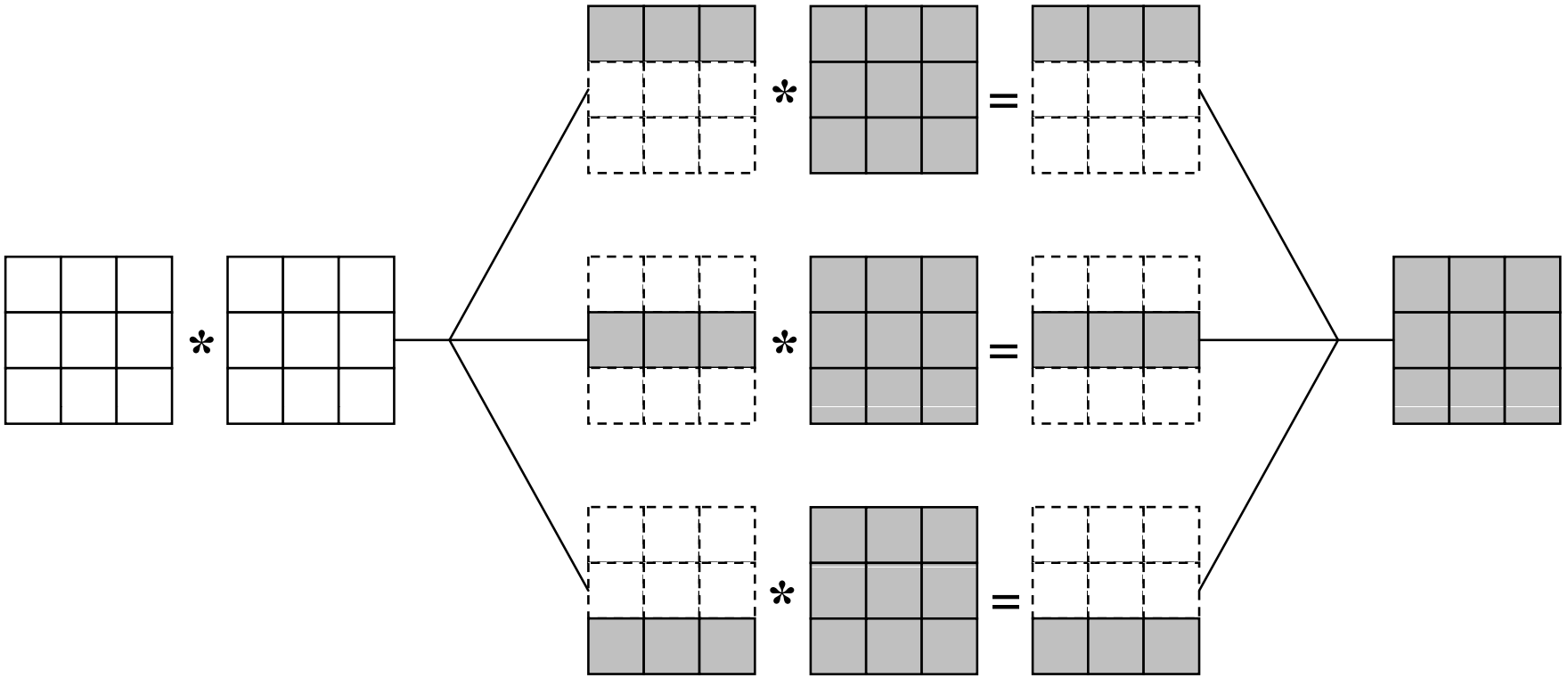
Базовый параллельный алгоритм умножения матриц…

- **Базовая подзадача** (агрегация) - процедура вычисления всех элементов одной из строк матрицы ***C*** (количество подзадач равно ***n***)

- **Распределение данных** – базовой подзадаче должны быть доступны одна из строк матрицы ***A*** и все столбцы матрицы ***B***. Такой подход не приводит к дублированию данных, поскольку разрабатываемый алгоритм ориентирован на применение для вычислительных систем с общей разделяемой памятью.

Базовый параллельный алгоритм умножения матриц…

- Выделение информационных зависимостей



Базовый параллельный алгоритм умножения матриц…

- **Масштабирование и распределение подзадач по вычислительным элементам**
  - Если число вычислительных элементов ***p*** меньше числа базовых подзадач ***n (p<n)***, базовые подзадачи могут быть укрупнены с тем, чтобы каждый вычислительный элемент вычислял несколько строк результирующей матрицы ***C***,
  - В этом случае, исходная матрица ***A*** и матрица-результат ***C*** разбиваются на ряд горизонтальных полос.

Базовый параллельный алгоритм умножения матриц…

❑ Анализ эффективности

– Время, которое тратится непосредственно на вычисления (матрицы размером *n*×*n*):

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

– Объем данных, которые считываются из оперативной памяти в кэш процессора:

$$V = 8 \cdot \left(8n^2 + n^3 + \frac{8n^3}{p} \right)$$

– Время, необходимое на считывание данных из оперативной памяти

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

– Время выполнения параллельного алгоритма:

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

Базовый параллельный алгоритм умножения матриц…

- **Программная реализация**
  - Для реализации базового параллельного алгоритма умножения матриц при помощи технологии OpenMP необходимо внести минимальные изменения в код последовательного алгоритма:

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

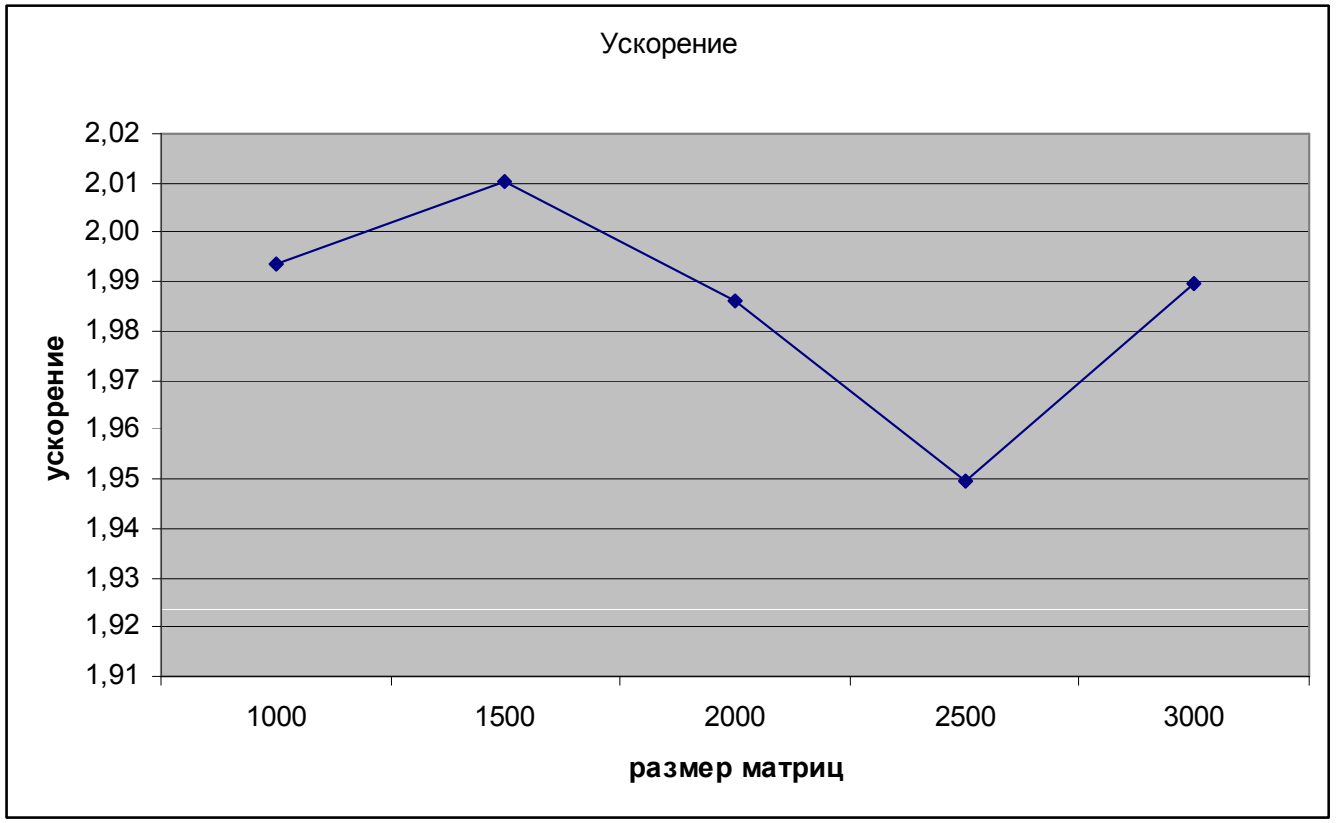
Базовый параллельный алгоритм умножения матриц…

- **Результаты вычислительных экспериментов…**
  - Ускорение вычислений:

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

Базовый параллельный алгоритм умножения матриц…

- **Результаты вычислительных экспериментов…**
  - Ускорение вычислений:



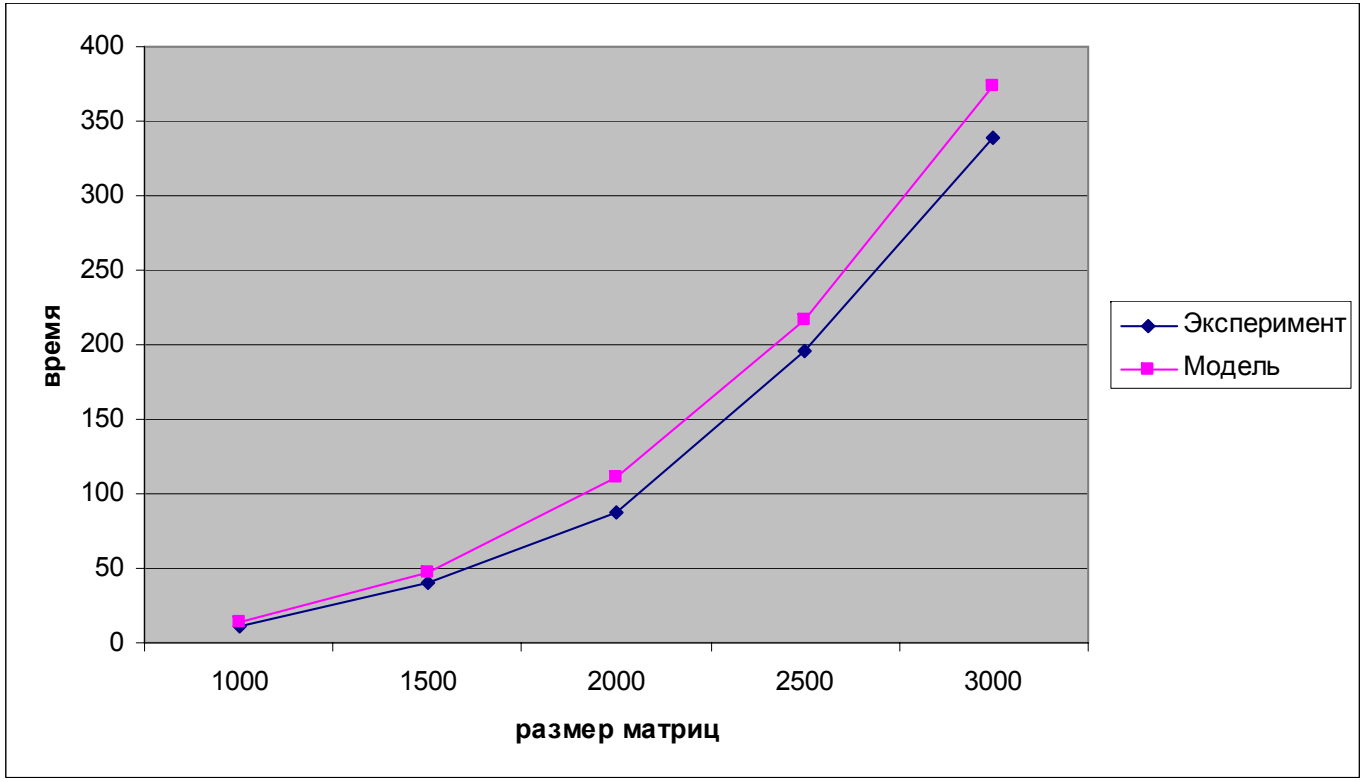
Базовый параллельный алгоритм умножения матриц…

- **Результаты вычислительных экспериментов…**
  - Сравнение теоретических оценок и экспериментальных данных:

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Общее время (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

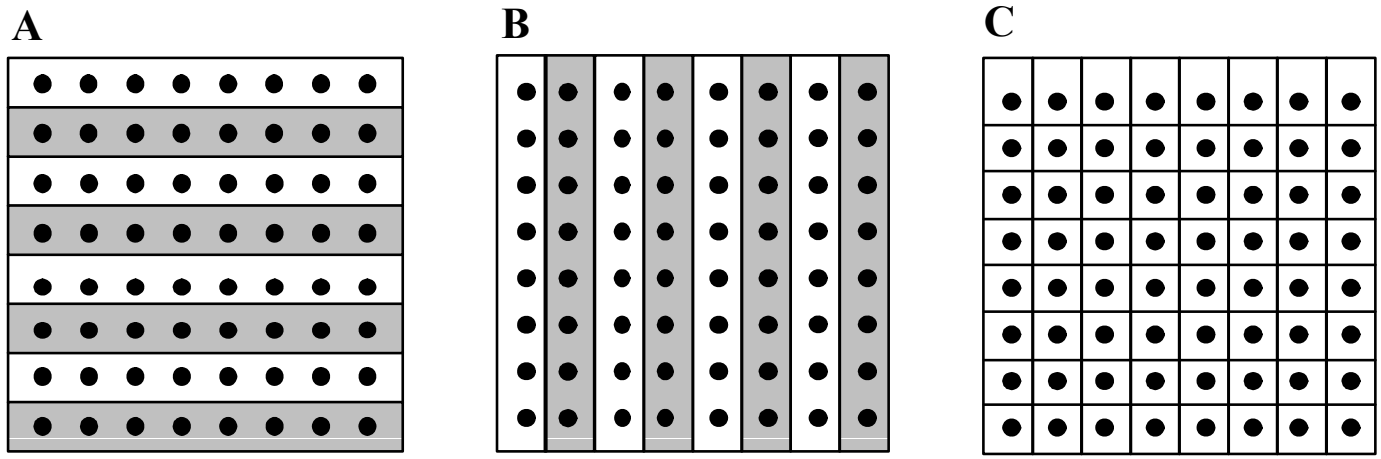
Базовый параллельный алгоритм умножения матриц

- **Результаты вычислительных экспериментов**
  - Сравнение теоретических оценок и экспериментальных данных:



Алгоритм умножения матриц, основанный на ленточном разделении данных…

- **Распределение данных** – Ленточная схема для матриц ***A*** и ***B***, блочная схема для матрицы ***C***

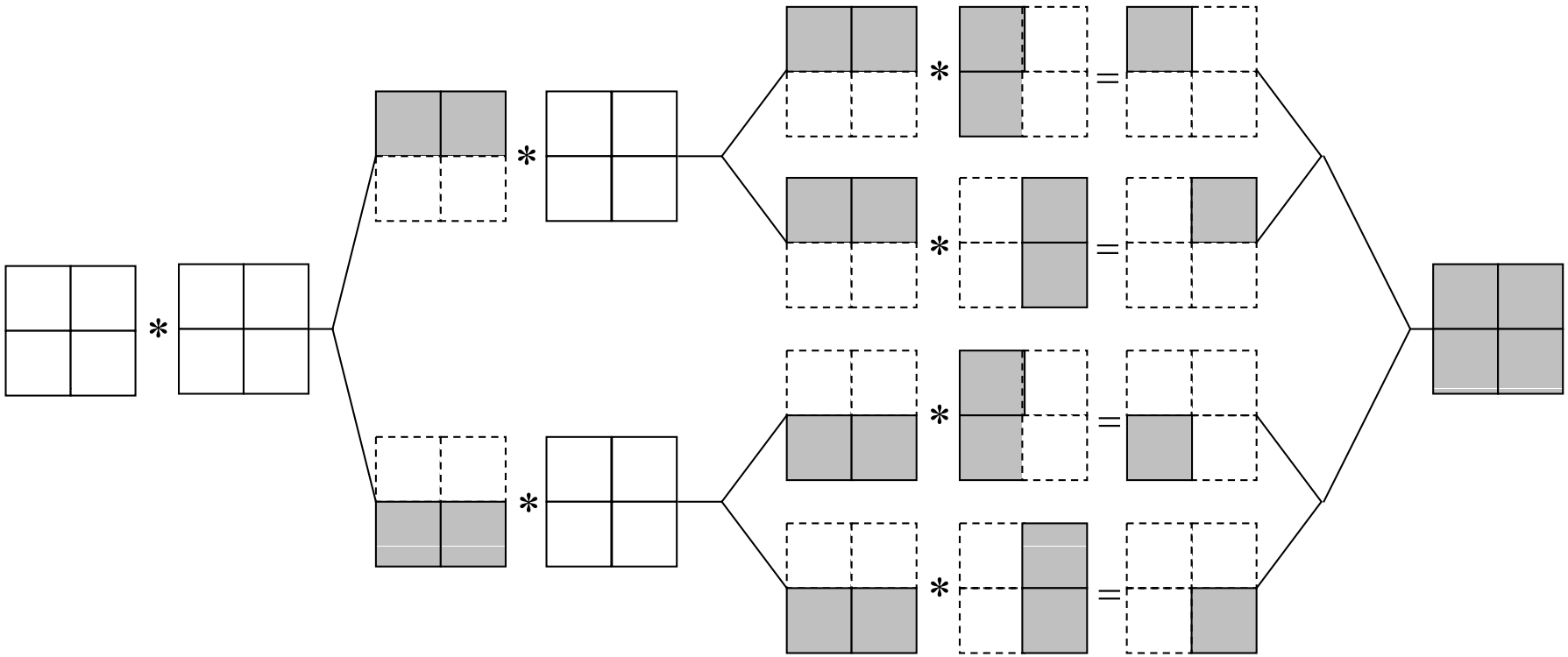


- **Базовая подзадача** - процедура вычисления всех элементов одного из блоков матрицы *C*

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

Алгоритм умножения матриц, основанный на ленточном разделении данных…

- Выделение информационных зависимостей



Алгоритм умножения матриц, основанный на ленточном разделении данных…

- **Масштабирование и распределение подзадач по вычислительным элементам**
  - 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.

Алгоритм умножения матриц, основанный на ленточном разделении данных…

❑ Анализ эффективности:

– Время, которое тратится непосредственно на вычисления (матрицы размером *n*×*n*):

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

– Время, необходимое на считывание данных из оперативной памяти:

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

– Время, необходимое для организации параллелизма (для каждой операции создания и завершения параллельной секции):

$$T_{par} = \delta \frac{n}{q}$$

– Время выполнения параллельного алгоритма:

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

Алгоритм умножения матриц, основанный на ленточном разделении данных…

- **Программная реализация**
  - Воспользуемся механизмом вложенного параллелизма:

```
// 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];
}
```

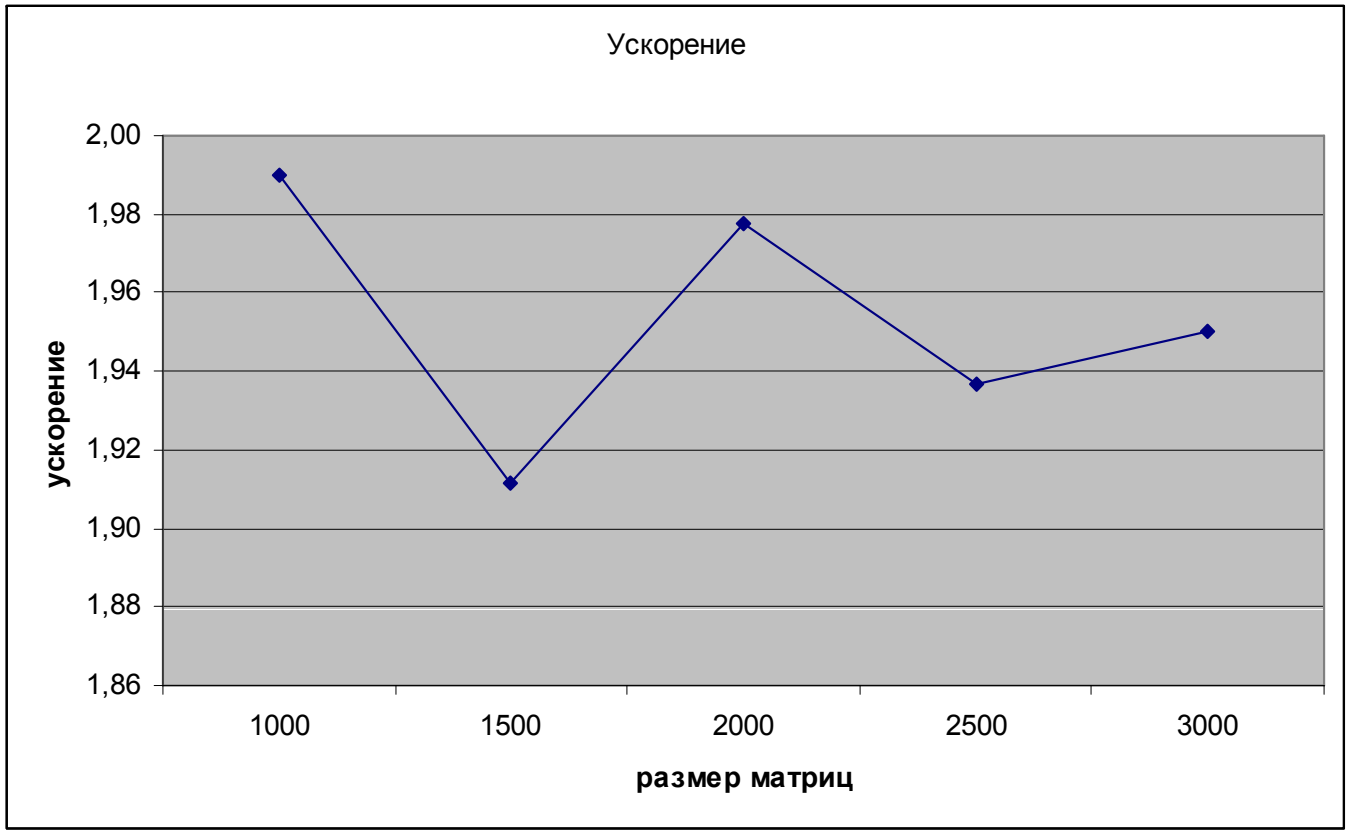
Алгоритм умножения матриц, основанный на ленточном разделении данных…

- **Результаты вычислительных экспериментов…**
  - Ускорение вычислений:

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

Алгоритм умножения матриц, основанный на ленточном разделении данных…

- **Результаты вычислительных экспериментов…**
  - Ускорение вычислений:



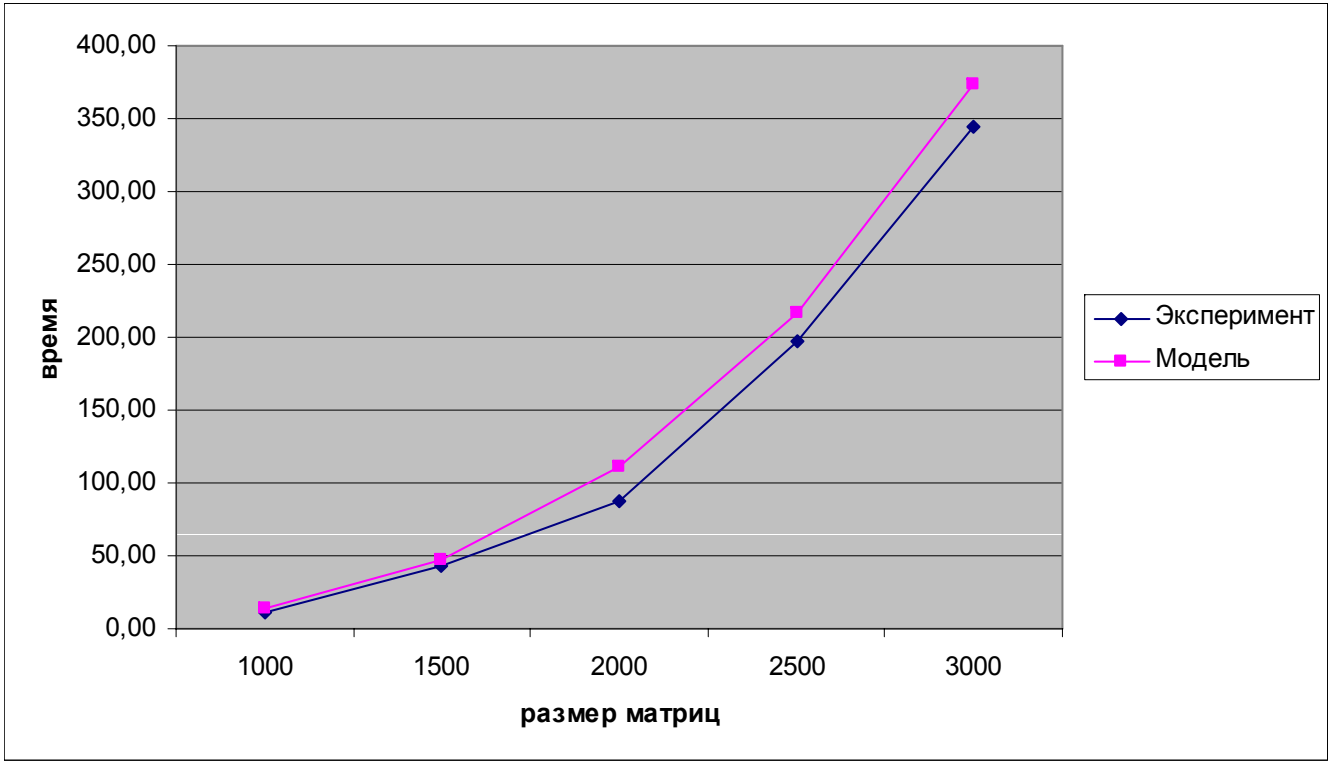
Алгоритм умножения матриц, основанный на ленточном разделении данных…

- **Результаты вычислительных экспериментов…**
  - Сравнение теоретических оценок и экспериментальных данных:

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

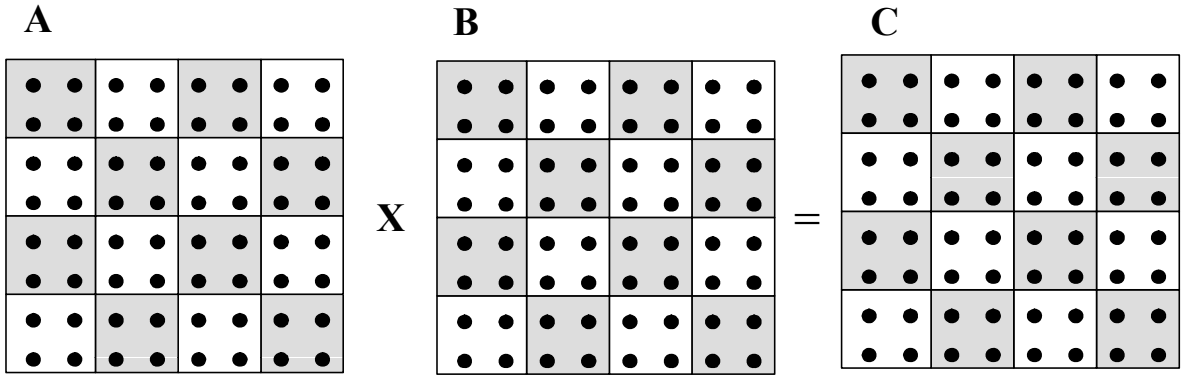
Алгоритм умножения матриц, основанный на ленточном разделении данных

- **Результаты вычислительных экспериментов:**
  - Сравнение теоретических оценок и экспериментальных данных:



Блочный алгоритм умножения матриц…

- **Распределение данных** – Блочная схема



- **Базовая подзадача** - процедура вычисления всех элементов одного из блоков матрицы ***C***

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0q-1} \\ \dots & \dots & \dots & \dots \\ A_{q-10} & A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0q-1} \\ \dots & \dots & \dots & \dots \\ B_{q-10} & B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0q-1} \\ \dots & \dots & \dots & \dots \\ C_{q-10} & C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix} \quad C_{ij} = \sum_{s=1}^{q} A_{is} B_{sj}$$

Блочный алгоритм умножения матриц…

- **Выделение информационных зависимостей:**
  - Подзадача *(i,j)* отвечает за вычисление блока C_{ij}, все подзадачи образуют прямоугольную решетку размером ***qxq***,
  - 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,
  - Количество итераций определяется размером блочной решетки.

Блочный алгоритм умножения матриц…

- **Масштабирование и распределение подзадач по вычислительным элементам:**
  - Размеры блоков могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом вычислительных элементов p,
  - В случае, когда число вычислительных элементов представимо в виде $p=\sigma^2$ можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (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.

Блочный алгоритм умножения матриц…

❑ Анализ эффективности:

– Время, которое тратится непосредственно на вычисления (матрицы размером *n*×*n*):

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

– Объем данных, которые считываются каждым потоком из оперативной памяти в кэш процессора:

$$V = 8 \cdot \left(8(n/q)^2 + 9(n/q)^3\right)$$

– Время, необходимое на считывание данных из оперативной памяти

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

– Время выполнения последовательного алгоритма:

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

Блочный алгоритм умножения матриц…

❑ Программная реализация

```
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…**
  - Ускорение вычислений:

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

Блочный алгоритм умножения матриц…

- **Результаты вычислительных экспериментов…**
  - Ускорение вычислений:

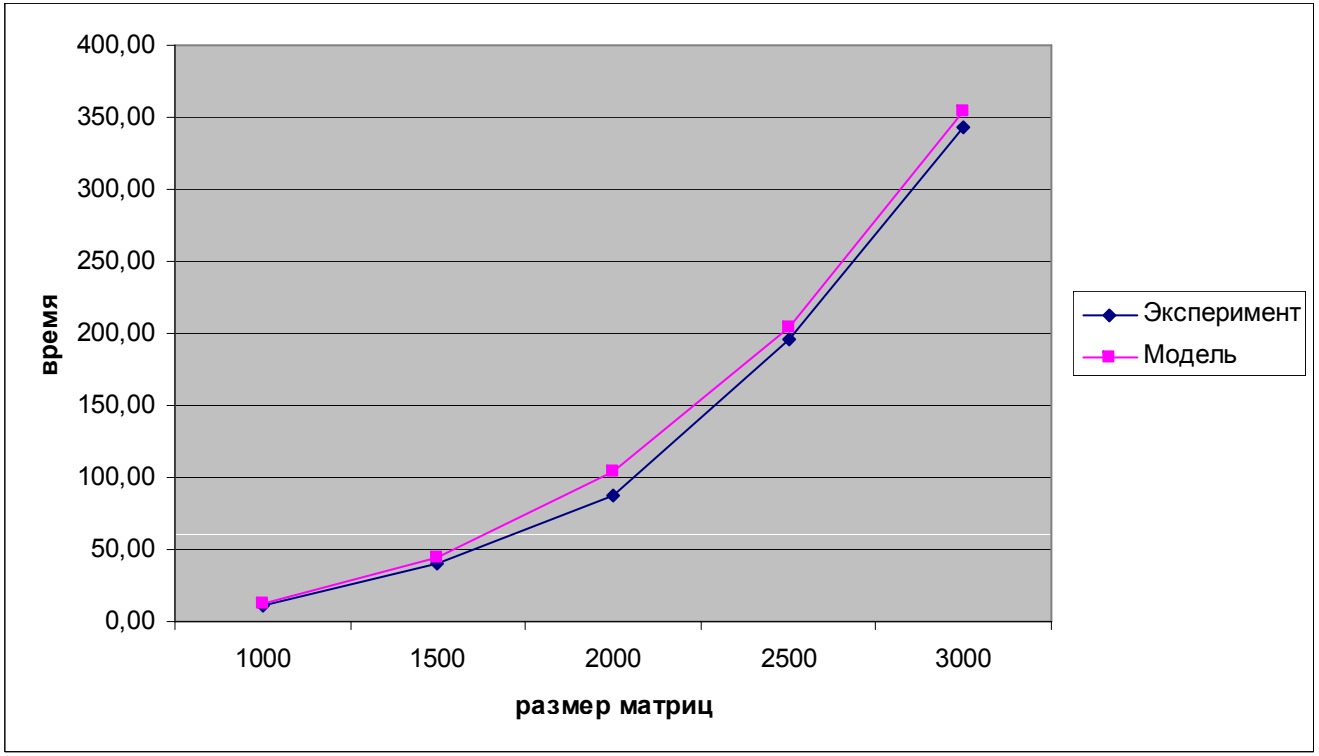
Блочный алгоритм умножения матриц…

- **Результаты вычислительных экспериментов…**
  - Сравнение теоретических оценок и экспериментальных данных:

Размер матриц	Эксперимент	Время счета (модель)	Время доступа к памяти (модель)	Общее время (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

Блочный алгоритм умножения матриц

- **Результаты вычислительных экспериментов:**
  - Сравнение теоретических оценок и экспериментальных данных:



Блочный алгоритм, эффективно использующий кэш-память...

- **Недостатки базового алгоритма умножения матриц:**
  - Матрицы ***A*** и ***B*** считываются в кэш многократно, что приводит к существенному увеличению времени работы базового алгоритма за счет затрат на загрузку необходимых данных

Блочный алгоритм, эффективно использующий кэш-память…

- **Последовательный алгоритм…**
  - В случае, когда матрицы, участвующие в умножении, настолько велики, что не могут быть помещены в кэш полностью, данные матрицы можно разделить на несколько матричных блоков меньшего размера и воспользоваться идеей блочного умножения матриц,
  - Разделение на блоки следует проводить таким образом, чтобы размер блока был настолько мал, что три блока, участвующие в умножении на данной итерации, можно было полностью 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.

Блочный алгоритм, эффективно использующий кэш-память…

- **Последовательный алгоритм…**
  - Выбор размера матричного блока:
    - В кэш одновременно должны помещаться три матричных блока – блоки матриц *A*, *B* и *C*. Пусть *V* – объем кэш в байтах. Тогда количество элементов типа double, которые могут быть помещены в кэш, составляет *V/8*. Таким образом, 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:

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

Блочный алгоритм, эффективно использующий кэш-память…

- **Последовательный алгоритм:**
  - Программная реализация

```
// 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…**
  - Базовая подзадача (поток) отвечает за вычисление блока результирующей матрицы,
  - Каждый поток обеспечивает получение несколько матричных блоков результирующей матрицы *C*,
  - На каждой итерации внешнего цикла потоки последовательно выполняют поблочное умножение горизонтальной полосы матрицы *A* на несколько вертикальных полос матрицы *B*.

Блочный алгоритм, эффективно использующий кэш-память...

- **Параллельный алгоритм.** Анализ эффективности
  - Время, которое тратится непосредственно на вычисления (матрицы размером *n*×*n*):

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

  - Время, необходимое на считывание данных из оперативной памяти

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

  - Время выполнения параллельного алгоритма:

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

Блочный алгоритм, эффективно использующий кэш-память…

- **Параллельный алгоритм…**
  - Программная реализация

```
// 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];
}
```

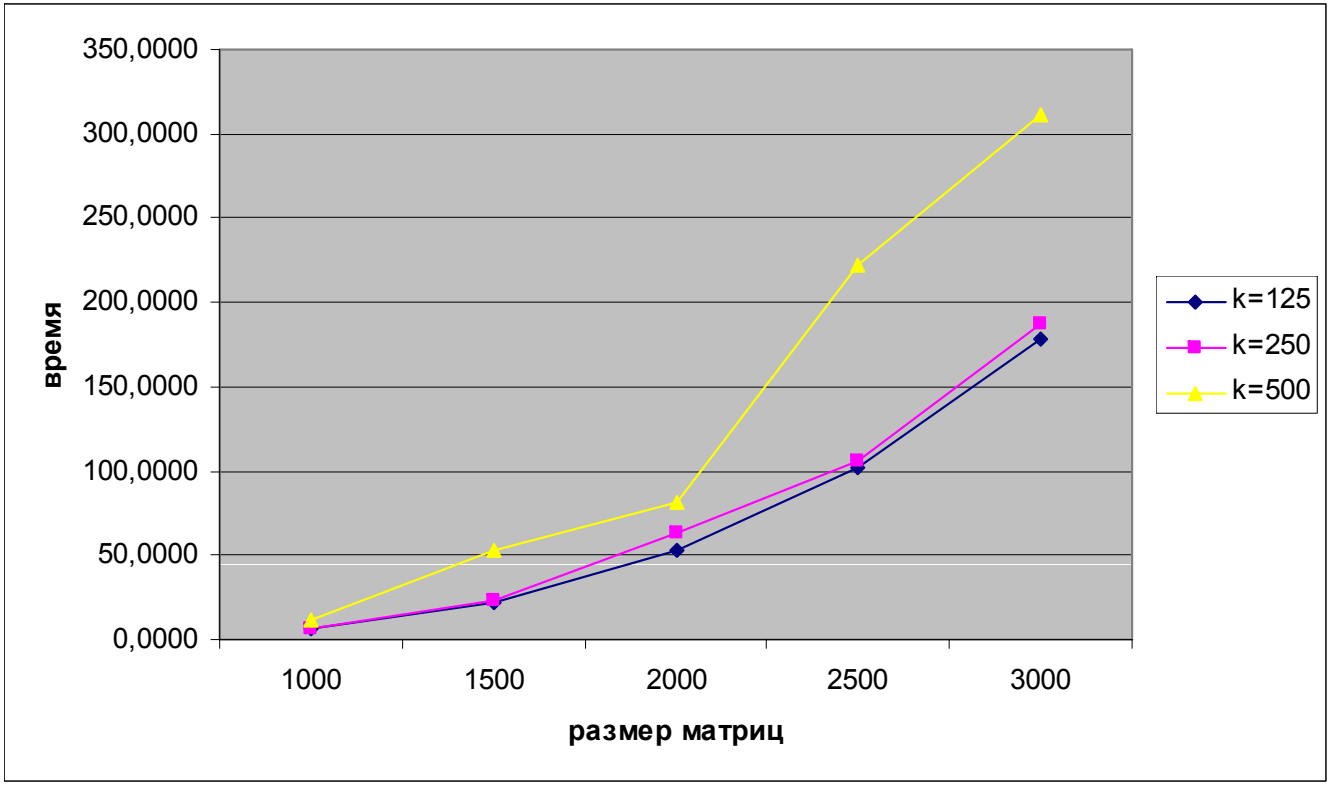
Блочный алгоритм, эффективно использующий кэш-память...

- 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

Блочный алгоритм, эффективно использующий кэш-память...

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



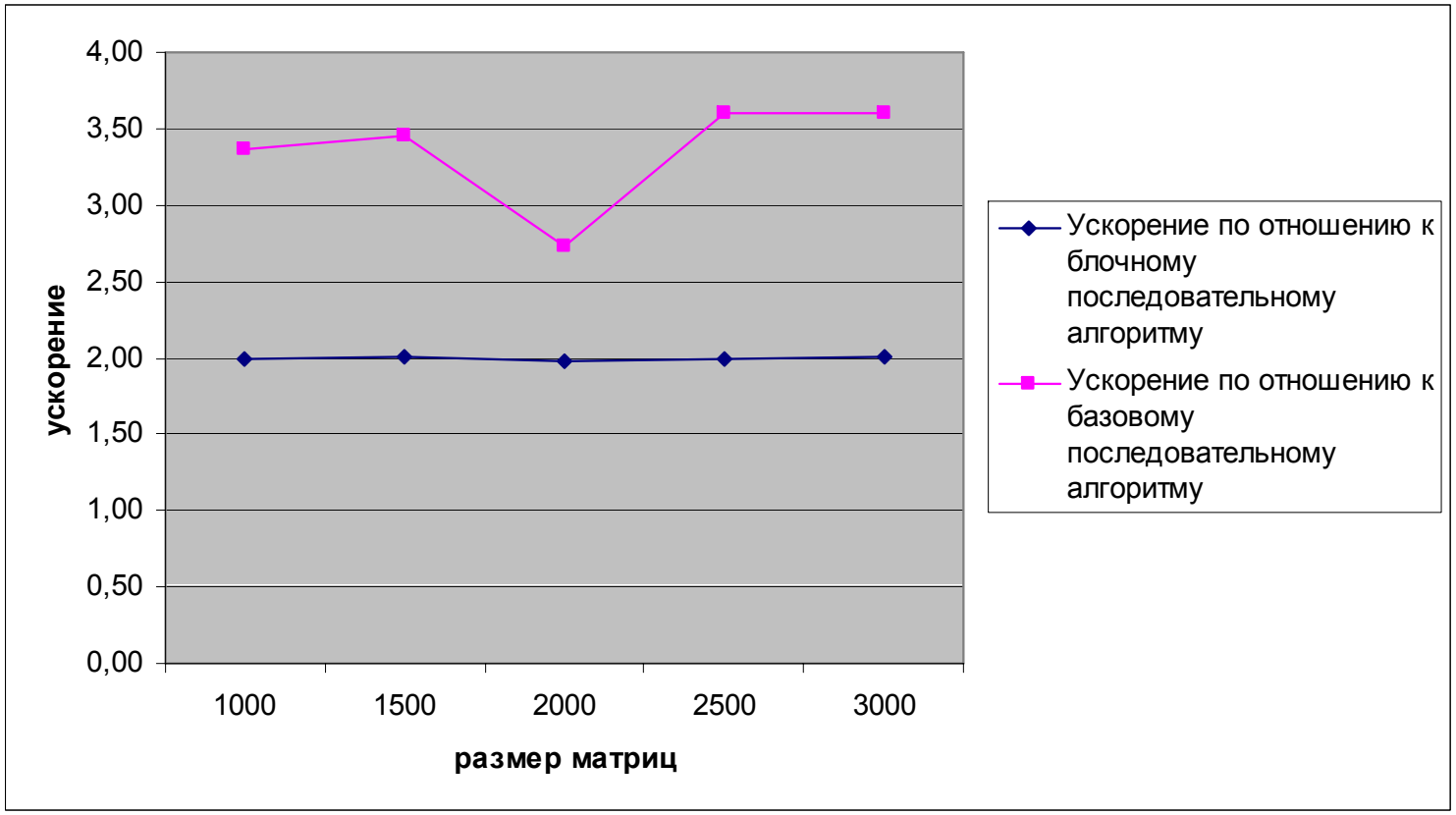
Блочный алгоритм, эффективно использующий кэш-память…

- **Результаты вычислительных экспериментов…**
  - Ускорение вычислений:

Размер матриц	Базовый последовательный алгоритм (T_1)	Блочный последовательный алгоритм (T_1')	Параллельный алгоритм		
			Время (T_p)	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

Блочный алгоритм, эффективно использующий кэш-память…

- **Результаты вычислительных экспериментов…**
  - Ускорение вычислений:



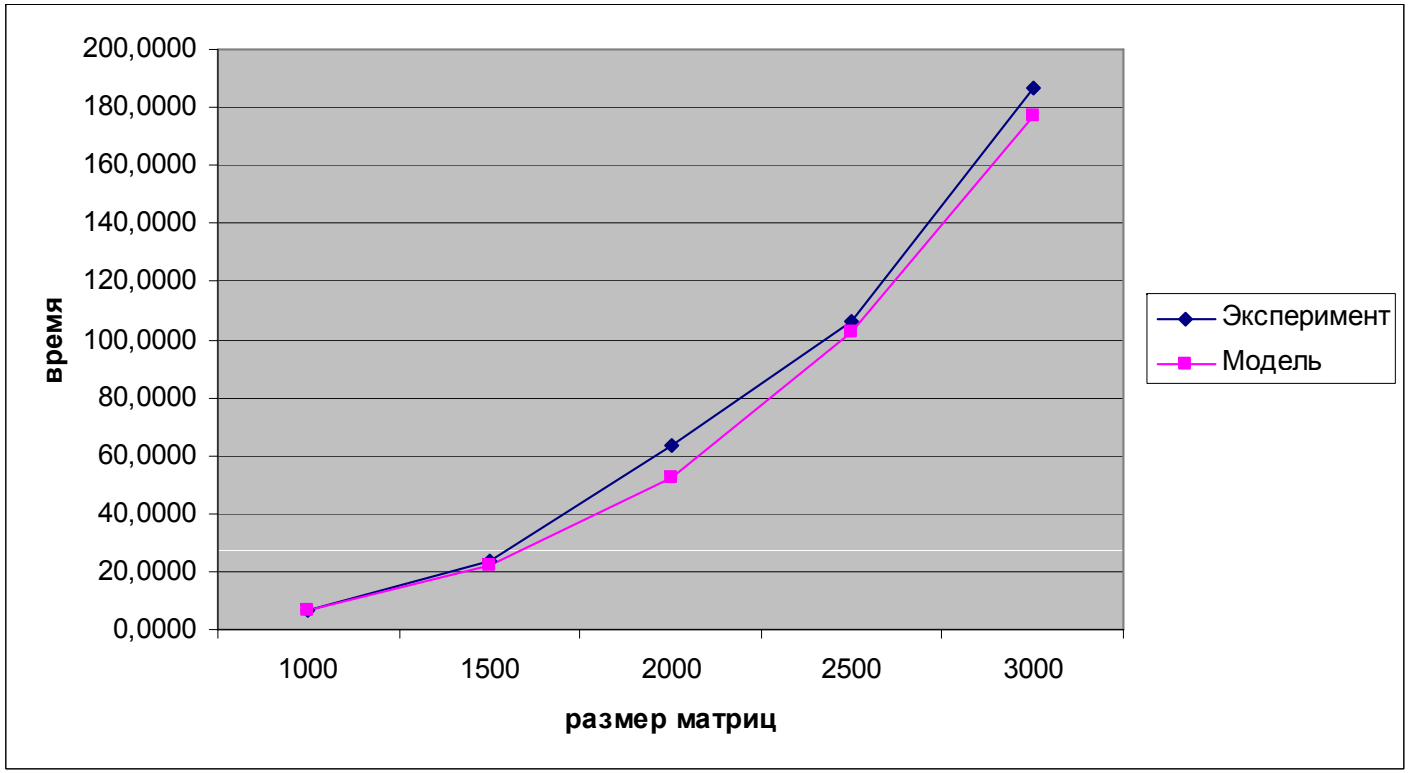
Блочный алгоритм, эффективно использующий кэш-память…

- **Результаты вычислительных экспериментов…**
  - Сравнение теоретических оценок и экспериментальных данных:

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

Блочный алгоритм, эффективно использующий кэш-память

- **Результаты вычислительных экспериментов:**
  - Сравнение теоретических оценок и экспериментальных данных:

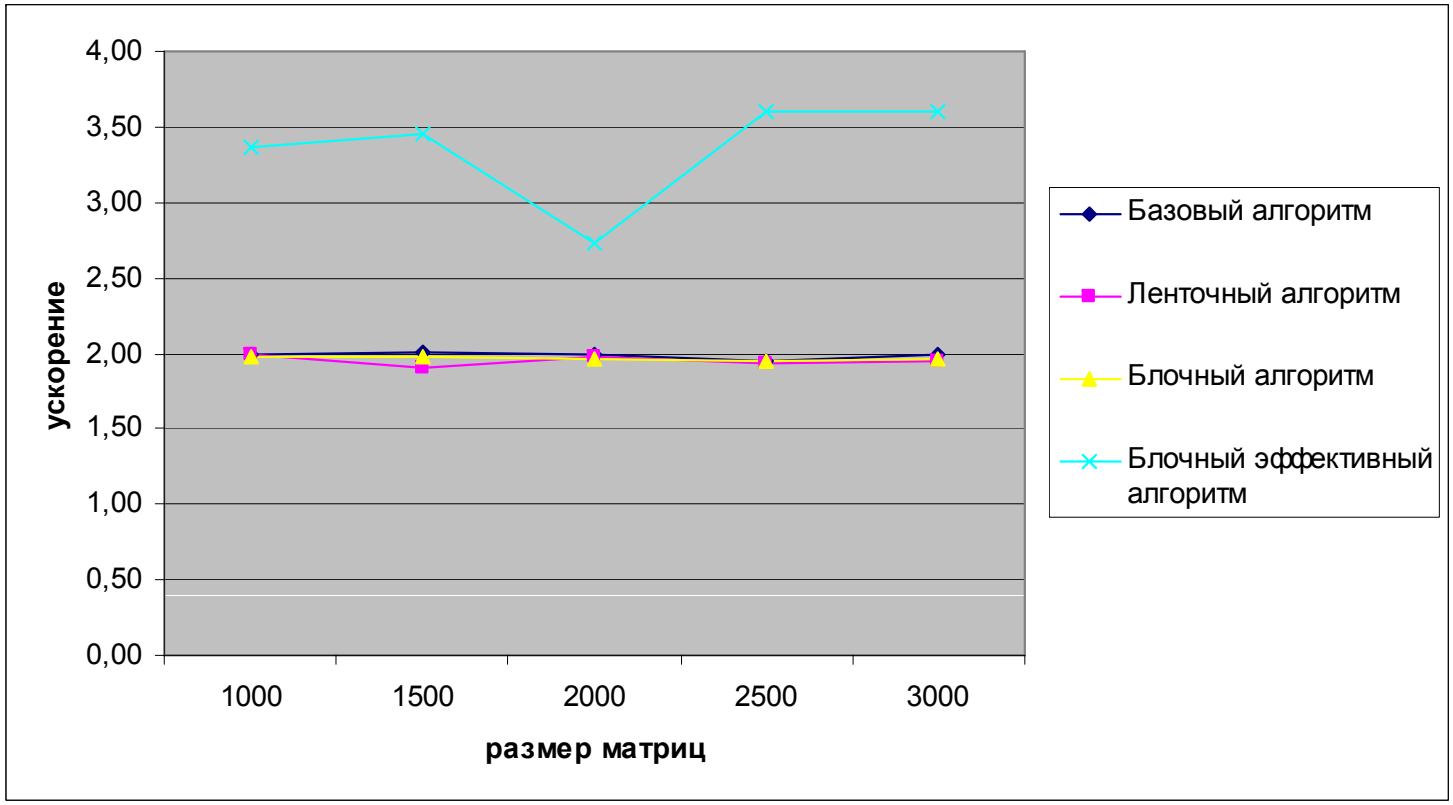


Заключение…

- Рассмотрены четыре параллельных метода для выполнения операции матричного умножения:
  - Алгоритм, основанный на разделении между потоками одной из матриц-аргументов (матрицы ***A***) и матрицы-результата,
  - Алгоритм, основанный на разделении первой матрицы на горизонтальные полосы, а второй матрицы – на вертикальные полосы,
  - Базовый блочный параллельный алгоритм,
  - Блочный алгоритм, эффективно использующий кэш-память.
- Теоретические оценки позволяют достаточно точно определить показатели эффективности параллельных вычислений

Заключение

- Показатели ускорения рассмотренных параллельных алгоритмов при умножении матриц по результатам вычислительных экспериментов



Вопросы для обсуждения

- Какие последовательные алгоритмы выполнения операции умножения матриц вы знаете? Какова их вычислительная трудоемкость?
- Какой основной подход используется при разработке параллельных алгоритмов матричного умножения?
- Какой из алгоритмов обладает наилучшими показателями ускорения и эффективности?
- Какие функции библиотеки OpenMP оказались необходимыми при программной реализации 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.

Литература

- **Kumar** V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc. (2nd edn., 2003)
- **Quinn**, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
- **Fox**, G.C., Otto, S.W. and Hey, A.J.G. (1987) Matrix Algorithms on a Hypercube I: Matrix Multiplication. Parallel Computing. 4 H. 17-31.

Следующая тема

- **Параллельные методы решения систем линейных уравнений**

Авторский коллектив

Гергель В.П., профессор, д.т.н., руководитель

Гришагин В.А., доцент, к.ф.м.н.

Абросимова О.Н., ассистент (раздел 10)

Лабутин Д.Ю., ассистент (система ПараЛаб)

Курылев А.Л., ассистент (лабораторные работы 4, 5)

Сысоев А.В., ассистент (раздел 1)

Гергель А.В., аспирант (раздел 12, лабораторная работа 6)

Лабутина А.А., аспирант (разделы 7,8,9, лабораторные работы 1, 2, 3, система ПараЛаб)

Сенин А.В., аспирант (раздел 11, лабораторные работы по Microsoft Compute Cluster)

Ливерко С.В. (система ПараЛаб)

О проекте

Целью проекта является создание образовательного комплекса "Многопроцессорные вычислительные системы и параллельное программирование", обеспечивающий рассмотрение вопросов параллельных вычислений, предусматриваемых рекомендациями Computing Curricula 2001 Международных организаций IEEE-CS и ACM. Данный образовательный комплекс может быть использован для обучения на начальном этапе подготовки специалистов в области информатики, вычислительной техники и информационных технологий.

Образовательный комплекс включает **учебный курс "Введение в методы параллельного программирования" и лабораторный практикум "Методы и технологии разработки параллельных программ"**, что позволяет органично сочетать фундаментальное образование в области программирования и практическое обучение методам разработки масштабного программного обеспечения для решения сложных вычислительно-трудоемких задач на высокопроизводительных вычислительных системах.

Проект выполнялся в Нижегородском государственном университете им. Н.И. Лобачевского на кафедре математического обеспечения ЭВМ факультета вычислительной математики и кибернетики (http://www.software.unn.ac.ru). Выполнение проекта осуществлялось при поддержке компании Microsoft.