Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Образовательный комплекс

Введение в методы параллельного программирования

Лабораторная работа 2.

Параллельные алгоритмы матричного умножения

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

Содержание

Упражнения:

- Определение задачи матричного умножения
- Реализация последовательного алгоритма матричного умножения
- Разработка параллельного алгоритма матричного умножения
- Реализация параллельного алгоритма умножения матриц

Упражнение 1: Определение задачи матричного умножения…

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 матрицы на вектор 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***

Упражнение 1: Определение задачи матричного умножения

```
// Serial algorithm of matrix multiplication
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];
    }
  }
}
```

(здесь и далее предполагается, что перемножаемые матрицы являются квадратными и имеют размер *Size*×*Size*)

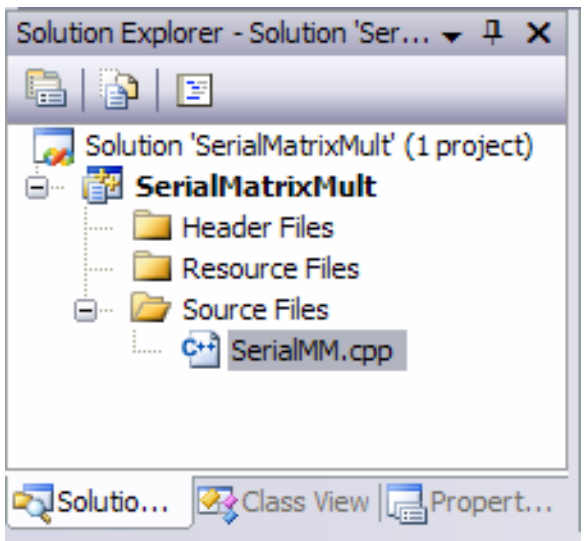
- Трудоемкость вычислений имеет порядок $O(n^3)$.

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Поэтапная разработка последовательного алгоритма:**
  - Задание 1 – Открытие проекта **SerialMatrixMult**
  - Задание 2 – Ввод размеров матриц
  - Задание 3 – Ввод данных
  - Задание 4 – Завершение процесса вычислений
  - Задание 5 – Реализация матричного умножения
  - Задание 6 – Проведение вычислительных экспериментов

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 1** – Открытие проекта **SerialMatrixMult:…**
  - Запустите приложение Microsoft Visual Studio 2005
  - Откройте проект **SerialMatrixMult.sln**, расположенный в папке **C:\MsLabs\SerialMatrixMult**
  - При помощи окна **Solution Explorer** откройте файл **SerialMM.cpp**



Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 1** – Открытие проекта **SerialMatrixMult**:
  - Переменные, которые будут использоваться в программе:

```
double* pAMatrix; // The first argument of multiplication
double* pBMatrix; // The second argument of multiplication
double* pCMatrix; // The result matrix
int Size;         // Size of matrices
```

  - Вывод начального сообщения и ожидание нажатия любой клавиши перед завершением выполнения приложения:

```
  printf ("Serial matrix multiplication program\n");
  getch();
```

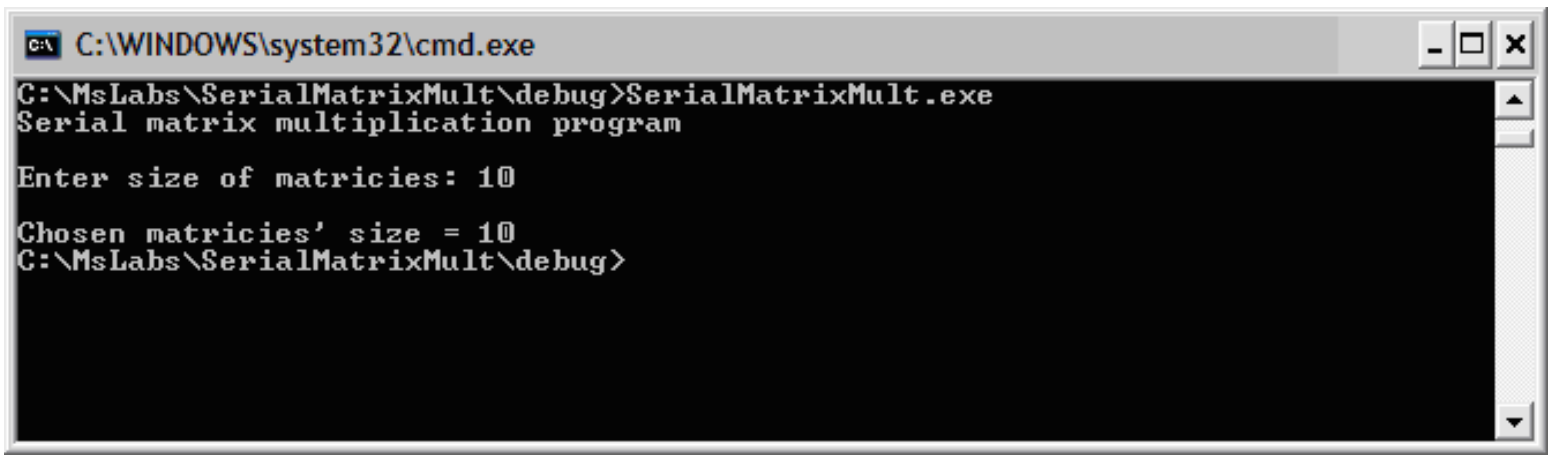
Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 2** – Ввод размеров матриц:…
  - Функция *ProcessInitialization* для задания исходных данных:
    - Определяет размеры матриц,
    - Выделяет память для всех матриц, участвующих в умножении (*pAMatrix, pBMatrix* и *pCMatrix*),
    - Задает значения элементов исходных матриц

```
void ProcessInitialization (double* &pAMatrix,
  double* &pBMatrix, double* &pCMatrix, int &Size),
где
-pAMatrix – первый аргумент матричного умножения (матрица A)
-pBMatrix – второй аргумент матричного умножения (матрица B)
-pCMatrix – результат матричного умножения
-Size     – порядок матриц
```

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 2** – Ввод размеров матриц:
  - Реализуйте ввод размеров (задание значения переменной *Size*) матриц, включая обработку возможных ошибочных ситуаций, внутри функции *ProcessInitialization*,
  - Добавьте вызов функции *ProcessInitialization* в главную функцию (функцию *main*) приложения,
  - Скомпилируйте и запустите приложение. Убедитесь в том, что размер матриц задается корректно



Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 3** – Ввод данных:…
  - Выделение памяти:

```
// Function for process initialization
void ProcessInitialization (double* &pAMatrix,
  double* &pBMatrix, double* &pCMatrix, int &Size) {
  // Setting the size of the matrices
  <…>

 // Memory allocation
  pAMatrix = new double [Size*Size];
  pBMatrix = new double [Size*Size];
  pCMatrix = new double [Size*Size];
}
```

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 3** – Ввод данных:…
  - Реализуйте функцию *DummyDataInitialization*, которая задавала бы значения элементов исходных по шаблону:

$$pAMatrix = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \; pBMatrix = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

  - Интерфейс функции *DummyDataInitialization*:

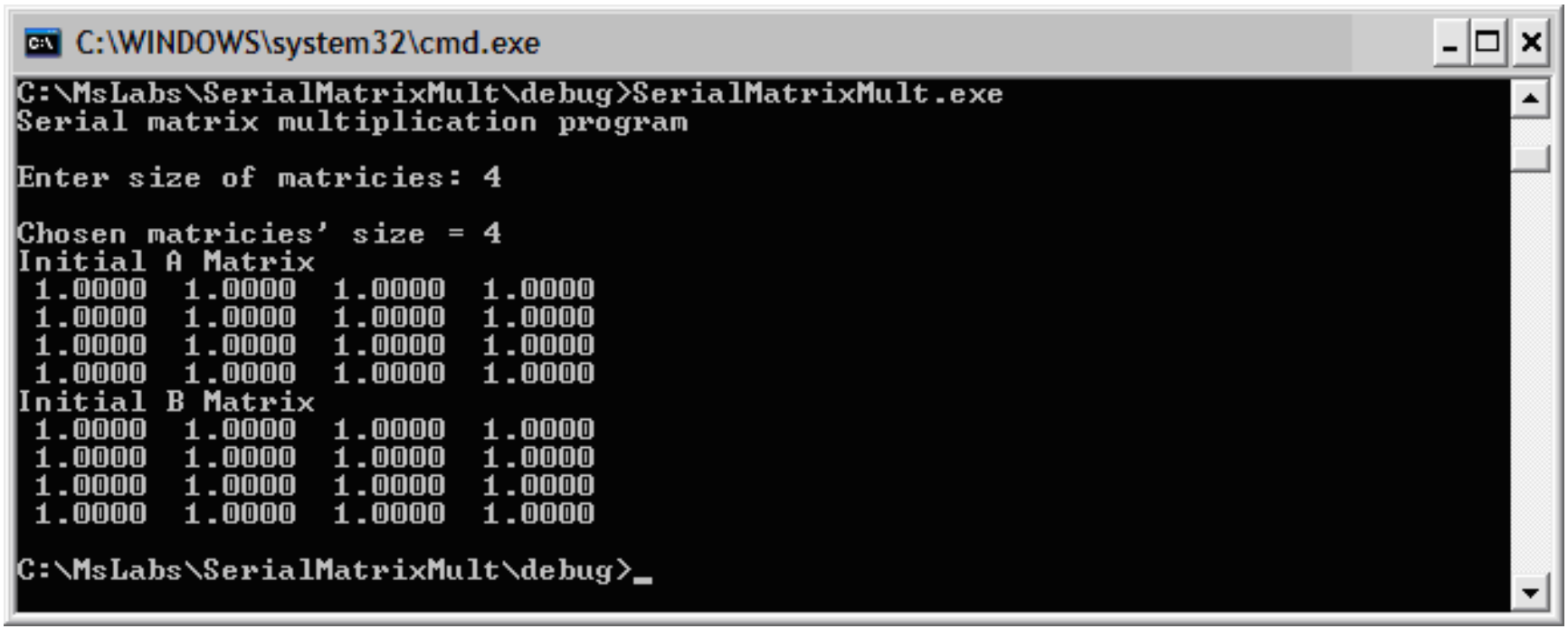
```
// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix,
  double* pBMatrix, int Size);
```

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 3** – Ввод данных:…
  - Реализуйте функцию *DummyDataInitialization,*
  - Добавьте вызов функции *DummyDataInitialization* в функцию инициализации процесса вычислений *ProcessInitialization* для задания значений исходных матриц,
  - Заполните матрицу *C* нулями,
  - В главной функции приложения после вызова функции *ProcessInitialization* распечатайте исходные матрицы, используйте для печати функцию форматированного вывода матрицы *PrintMatrix*, разработанную в лабораторной работе 1,
  - Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных осуществляется корректно.

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 3** – Ввод данных:



Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 4** – Завершение процесса вычислений:…
  - Функция *ProcessTermination* для завершения процесса вычислений:
    - Освобождает память, выделенную в ходе выполнения программы

```
// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
  double* pCMatrix),
где
pAMatrix – первый аргумент матричного умножения (матрица A)
pBMatrix – второй аргумент матричного умножения (матрица B)
pCMatrix – результат матричного умножения
```

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 4** – Завершение процесса вычислений:
  - Реализуйте функцию *ProcessTermination*,
  - Добавьте вызов этой функции в главную функцию приложения,
  - Скомпилируйте и запустите приложение. Убедитесь, что программа выполняется корректно.

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 5** – Реализация матричного умножения:…
  - Функция *SerialResultCalculation* выполняет умножение матриц в соответствии с алгоритмом, изложенным в упражнении 1:

```
// Function for matrix multiplication
void SerialResultCalculation (double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size),
где
- pAMatrix – первый аргумент матричного умножения (матрица A)
- pBMatrix – второй аргумент матричного умножения (матрица B)
- pCMatrix – результат матричного умножения
- Size     - порядок матриц
```

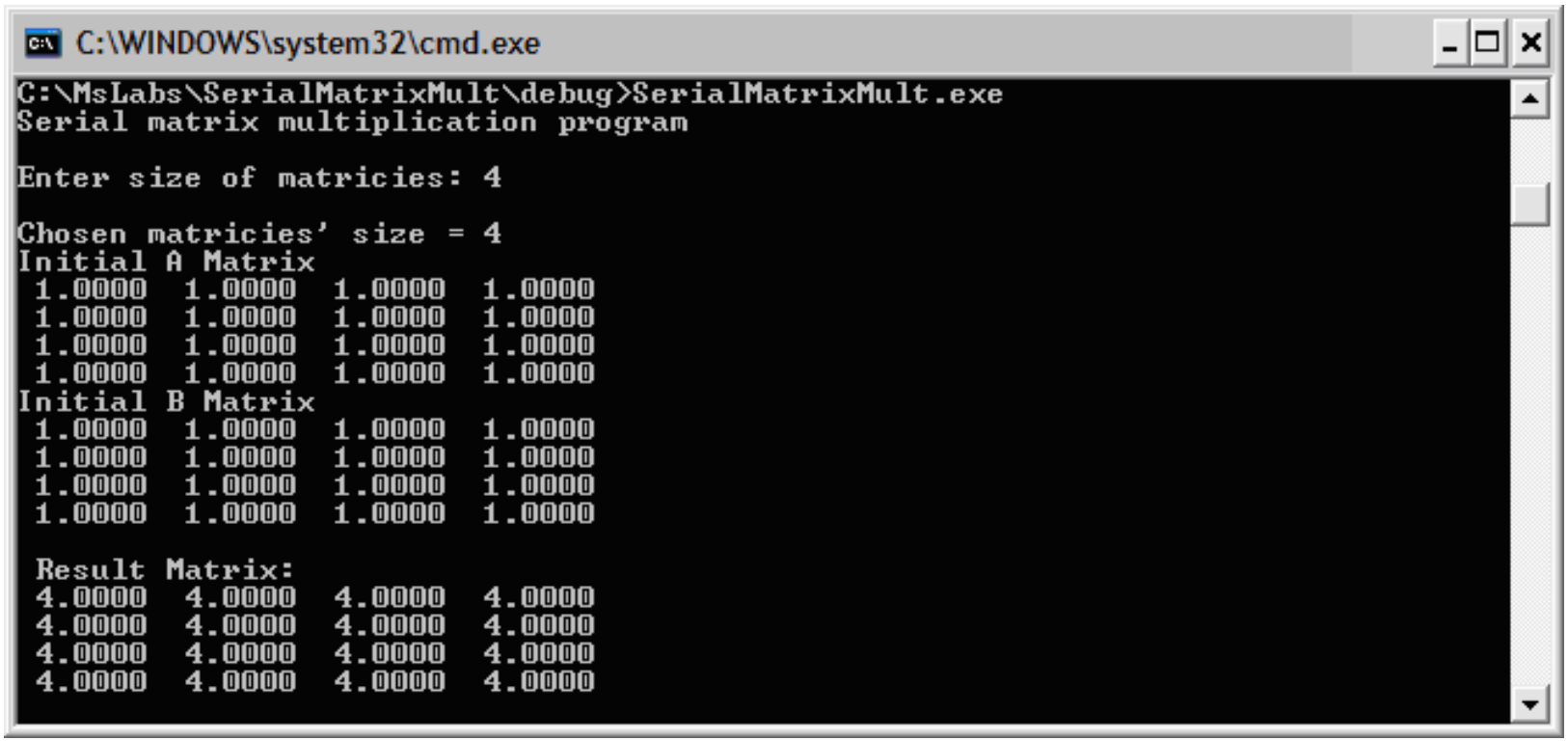
Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 5** – Реализация матричного умножения:…
  - Добавьте вызов функции выполнения матричного умножения в основную функцию программы,
  - Для контроля правильности выполнения умножения, распечатайте результирующую матрицу,
  - Проанализируйте результат работы алгоритма умножения матриц:
    - если алгоритм реализован правильно, то в результате должна быть получена матрица, значения всех элементов которой равны порядку этой матрицы

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{pmatrix}$$

Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 5** – Реализация матричного умножения:



Упражнение 2: Реализация последовательного алгоритма матричного умножения…

- **Задание 6** – Проведение вычислительных экспериментов:…
  - Функция для задания значений элементов матриц при помощи датчика случайных чисел:

```
// Function for random initialization of the matrix elements
void RandomDataInitialization (double* pAMatrix,
  double* pBMatrix, int Size);
```

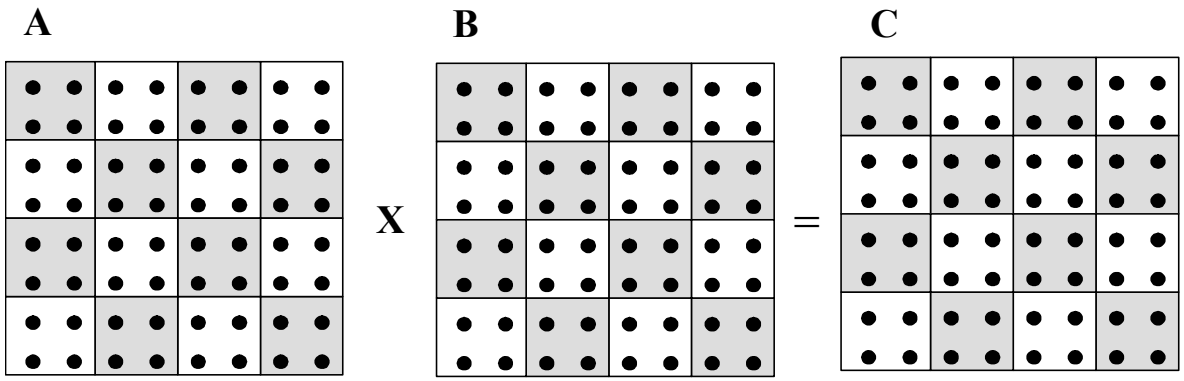
  - Замените вызов функции *DummyDataInitialization* на *RandomDataInitialization* в функции *ProcessInitialization*,
  - Добавьте вычисление и вывод времени выполнения матричного умножения,
  - Протестируйте работоспособность приложения

Упражнение 2: Реализация последовательного алгоритма матричного умножения

- **Задание 6** – Проведение вычислительных экспериментов:
  - Измерьте времена работы последовательного алгоритма матричного умножения при различных размерах матрицы,
  - Рассчитайте теоретическое время выполнения алгоритма,
  - Занесите результаты измерений и вычислений в таблицу.

Упражнение 3: Разработка параллельного алгоритма матричного умножения (метод Фокса)…

- **Распределение данных** – Блочная схема



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

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

Упражнение 3: Разработка параллельного алгоритма матричного умножения (метод Фокса)…

- **Выделение информационных зависимостей**
  - Подзадача *(i,j)* отвечает за вычисление блока C_{ij}, - как результат, все подзадачи образуют прямоугольную решетку размером $q \times q$,
  - В ходе вычислений в каждой подзадаче располагаются четыре матричных блока:
    - блок C_{ij} матрицы C, вычисляемый подзадачей,
    - блок A_{ij} матрицы A, размещаемый в подзадаче перед началом вычислений,
    - блоки A'_{ij}, B'_{ij} матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

Упражнение 3: Разработка параллельного алгоритма матричного умножения (метод Фокса)…

- **Выделение информационных зависимостей -** для каждой итерации l, $0 \le l < q$, выполняется:
  - блок $\boldsymbol{A_{ij}}$ подзадачи (i,j) пересылается на все подзадачи той же строки i решетки подзадач; индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением:

    $$j = (i+l) \bmod q,$$

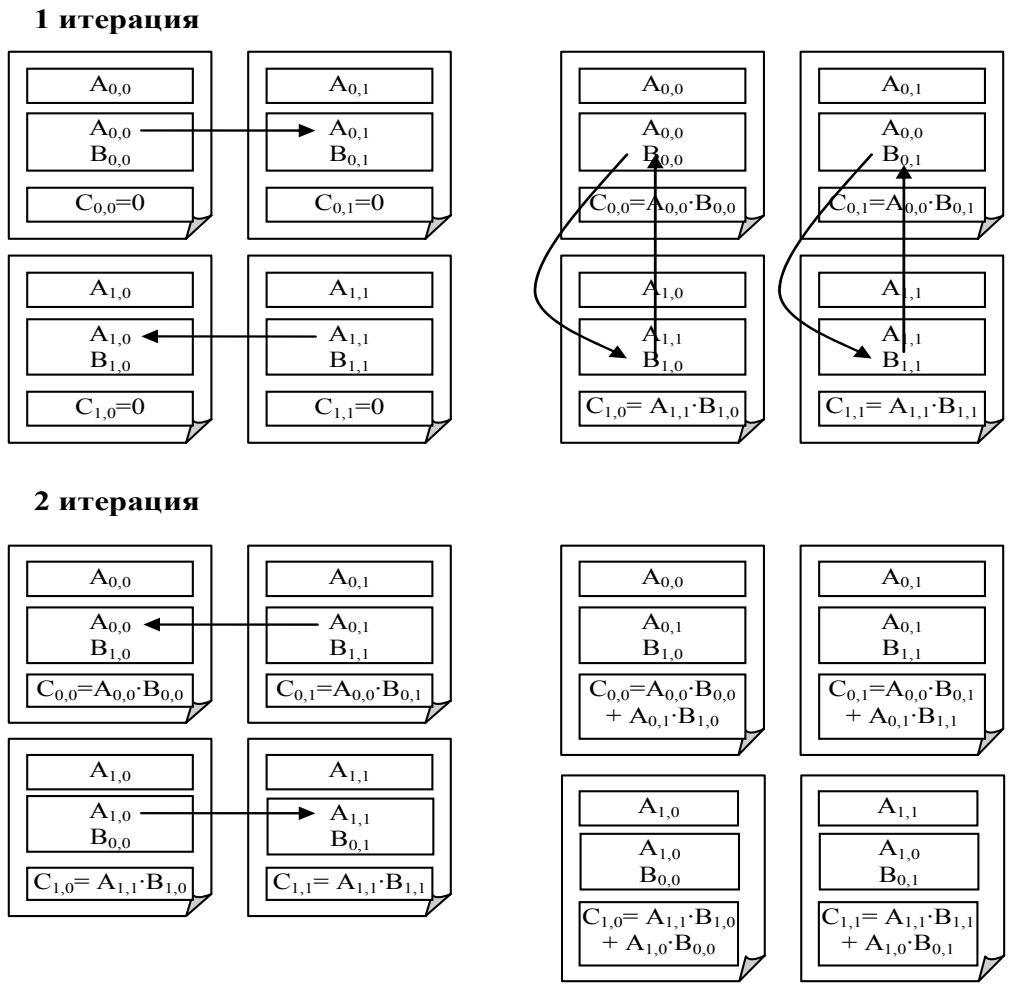
    где mod есть операция получения остатка от целого деления,
  - полученные в результате пересылок блоки $\boldsymbol{A_{ij}'}$, $\boldsymbol{B_{ij}'}$ каждой подзадачи (i,j) перемножаются и прибавляются к блоку $\boldsymbol{C_{ij}}$

    $$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij}$$

  - блоки $\boldsymbol{B_{ij}'}$ каждой подзадачи (i,j) пересылаются подзадачам, являющимися соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Упражнение 3: Разработка параллельного алгоритма матричного умножения (метод Фокса)…

- Схема информационного взаимодействия



Упражнение 3: Разработка параллельного алгоритма матричного умножения (метод Фокса)

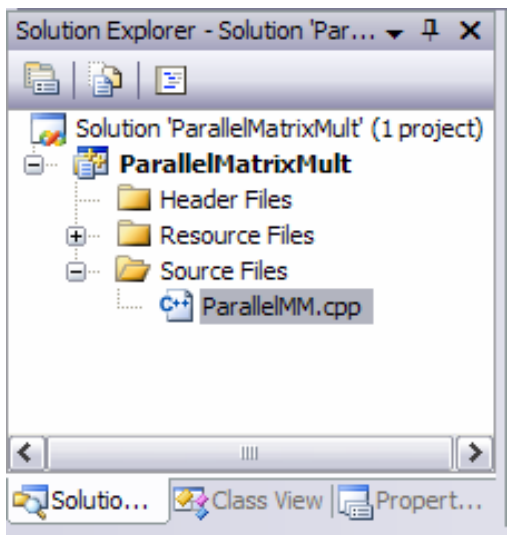
- **Масштабирование и распределение подзадач по процессорам:**
  - Размеры блоков могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров ***p***,
  - Наиболее эффективное выполнение метода Фокса может быть обеспечено при представлении множества имеющихся процессоров в виде квадратной решетки,
  - В этом случае можно осуществить непосредственное отображение набора подзадач на множество процессоров - базовую подзадачу (*i,j*) следует располагать на процессоре $\boldsymbol{p}_{i,j}$

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Поэтапная разработка параллельного алгоритма:**
  - Задание 1 – Открытие проекта **ParallelMatrixMult**
  - Задание 2 – Создание виртуальной декартовой топологии
  - Задание 3 – Определение размеров объектов и ввод исходных данных
  - Задание 4 – Завершение процесса вычислений
  - Задание 5 – Распределение данных между процессами
  - Задание 6 – Начало реализации параллельного алгоритма матричного умножения
  - Задание 7 – Рассылка блоков матрицы *A*
  - Задание 8 – Циклический сдвиг блоков матрицы *B* вдоль столбцов процессорной решетки
  - Задание 9 – Умножение матричных блоков
  - Задание 10 – Сбор результатов
  - Задание 11 – Проверка правильности работы программы
  - Задание 12 – Проведение вычислительных экспериментов

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 1** – Открытие проекта **ParallelMatrixMult:…**
  - Запустите приложение Microsoft Visual Studio 2005
  - Откройте проект **ParallelMatrixMult.sln**, расположенный в папке **C:\MsLabs\ParallelMatrixMult**
  - При помощи окна **Solution Explorer** откройте файл **ParallelMM.cpp**



Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 1** – Открытие проекта **ParallelMatrixMult**:
  - Проект содержит функции:
    - *DummyDataInitialization* - начальное задание исходных данных,
    - *RandomDataInitialization* - задание исходных данных при помощи датчика случайных чисел,
    - *SerialResultCalculation* - последовательный алгоритм матричного умножения,
    - *PrintMatrix* – печать матрицы
  - В главной функции приложения объявлены перk, ременные *ProcNum*, *ProcRank*, *pAMatrix*, *pBMatrix*, *pCMatrix*, *Size*,
  - Проинициализирована среда выполнения MPI-программы, определены количество процессов *ProcNum* и ранг каждого процесса *ProcRank,*
  - Скомпилируйте и запустите приложение. Убедитесь, что на экран выводится приветствие:

```
"Parallel matrix multiplication program"
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 2** – Создание виртуальной декартовой топологии:…
  - Для эффективного выполнения алгоритма Фокса необходимо организовать доступные процессы MPI-программы в виртуальную топологию в виде двумерной квадратной решетки,
  - Это возможно только в случае, когда число доступных процессов является полным квадратом (*ProcNum=GridSize×GridSize*),
  - Объявите глобальную переменную *GridSize*. В основной функции приложения выполните проверку, является ли число процессов полным квадратом. Если нет, выведите диагностическое сообщение,
  - В случае, если число процессов является полным квадратом, определите значение переменной *GridSize*.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 2** – Создание виртуальной декартовой топологии:…
  - Реализуйте функцию *CreateGridCommunicators*:
    - Создает коммуникатор в виде двумерной квадратной решетки *GridComm*,
    - Создает коммуникатор *RowComm* для каждой строки решетки и коммуникатор *ColComm* для каждого столбца решетки

```
MPI_Comm GridComm;    // Grid communicator
MPI_Comm RowComm;     // Row communicator
MPI_Comm ColComm;     // Column communicator

// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators ();
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 2** – Создание виртуальной декартовой топологии:
  - Реализуйте функцию *CreateGridCommunicators*,
  - Добавьте вызов функции *CreateGridCommunicators* в главную функцию приложения,
  - Скомпилируйте приложение,
  - Запустите приложение несколько раз, изменяя количество доступных процессов. Убедитесь в том, что в случае, когда доступное число процессов не является полным квадратом, выдается диагностическое сообщение и приложение завершает работу.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 3** – Определение размеров объектов и ввод исходных данных:…
  - Переменные, необходимые для выполнения параллельного алгоритма умножения матриц:

```
int BlockSize;           // Sizes of matrix blocks on current
                         // process
double *pMatrixAblock; // Initial block of matrix A on
                         // current process
double *pAblock;         // Current block of matrix A on
                         // current process
double *pBblock;         // Current block of matrix B on
                         // current process
double *pCblock;         // Block of result matrix C on
                         // current process
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 3** – Определение размеров объектов и ввод исходных данных:…
  - Функция *ProcessInitialization:*
    - Организует диалог с пользователем на одном из процессов для того, чтобы определить размер исходных матриц *Size*, рассылает значение переменной *Size* на все процессы,
    - Вычисляет размер матричных блоков *BlockSize*,
    - Выделяет память для хранения исходных матриц на нулевом процессе и для хранения матричных блоков на всех процессах,
    - Определяет значения элементов исходных матриц, заполняет блоки результирующей матрицы нулями.

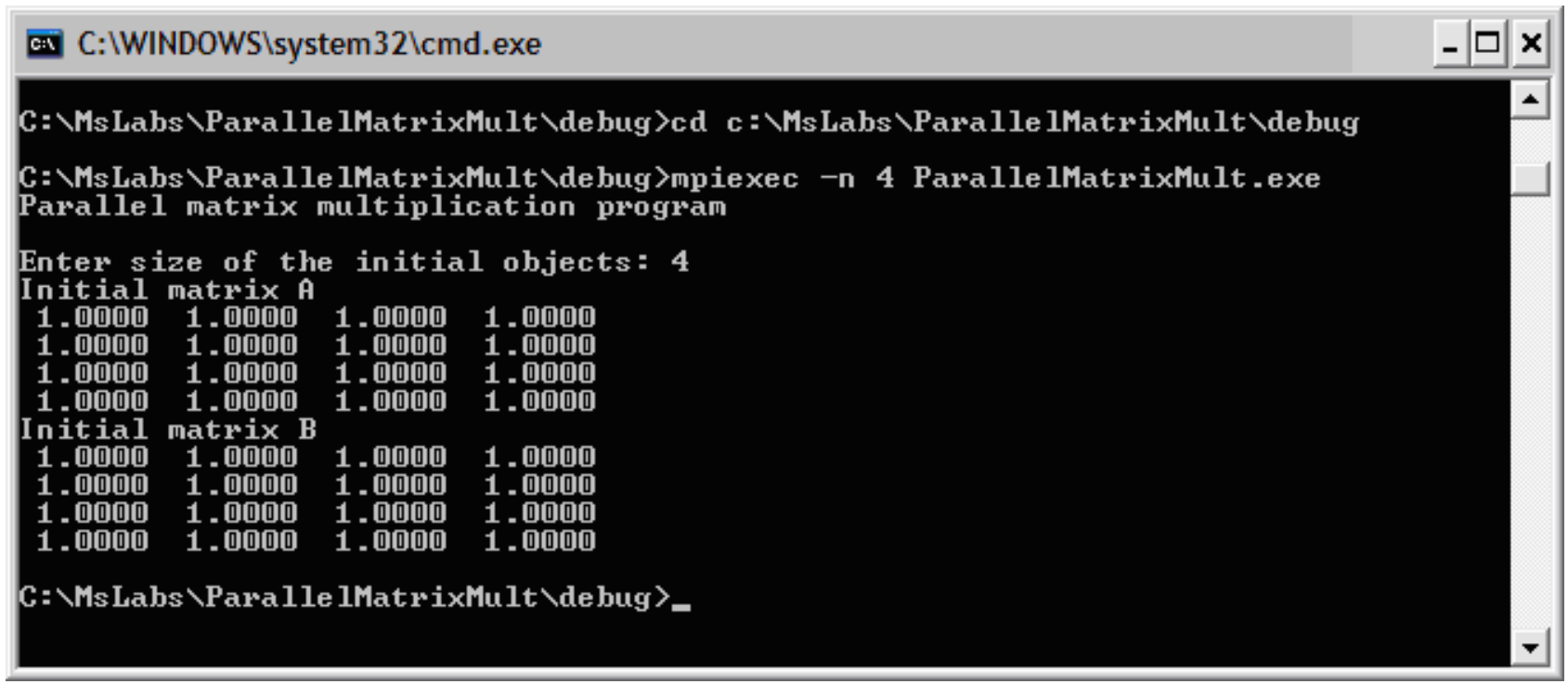
```
void ProcessInitialization (double* &pAMatrix,
  double* &pBMatrix, double* &pCMatrix, double* &pAblock,
  double* &pBblock, double* &pCblock,double* &pMatrixAblock,
  int &Size, int &BlockSize )
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 3** – Определение размеров объектов и ввод исходных данных:…
  - Реализуйте функцию *ProcessInitialization*,
  - Добавьте вызов функции инициализации процесса вычислений в основную функцию параллельного приложения,
  - Для контроля правильности выполнения этого этапа, распечатайте исходные матрицы на нулевом процессе, используя функцию *PrintMatrix*,
  - Скомпилируйте и запустите приложение. Убедитесь в его работоспособности.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 3** – Определение размеров объектов и ввод исходных данных:



Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 4** – Завершение процесса вычислений:…
  - Функция *ProcessTermination*:
    - Корректное завершение работы приложения,
    - Освобождение памяти, выделенной динамически в ходе выполнения приложения

```
// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
  double* pCMatrix, double* pAblock, double* pBblock,
  double* pCblock, double* pMatrixAblock)
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

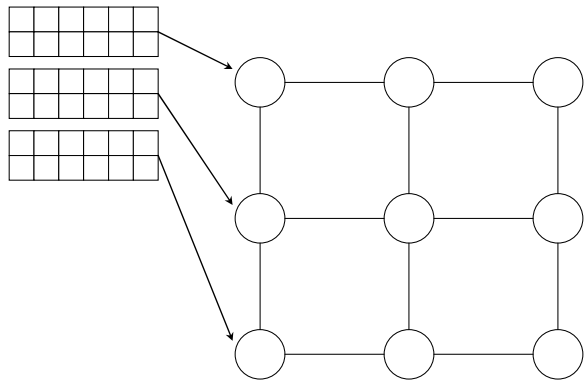
- **Задание 4** – Завершение процесса вычислений:
  - Реализуйте функцию *ProcessTermination,*
  - Добавьте вызов функции *ProcessTermination* в функцию *main,*
  - Протестируйте работоспособность приложения

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

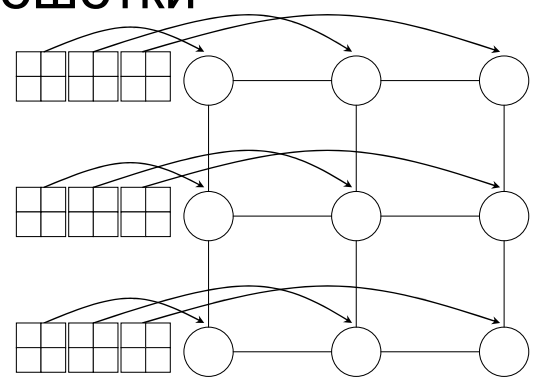
- **Задание 5** – Распределение данных между процессами:…
  - Исходные матрицы расположены на ведущем процессе, который расположен в левом верхнем углу процессорной решетки,
  - Нужно распределить матрицы поблочно между процессами так, чтобы блоки A_{ij} и B_{ij} были помещены на процессе, расположенном на пересечении i-ой строки и j-ого столбца процессорной решетки

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 5** – Распределение данных между процессами:…
  - **Первый этап**: Распределение блочных полос между процессами левого крайнего столбца решетки



  - **Второй этап**: Распределение блоков между процессами в каждой строке решетки



Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 5** – Распределение данных между процессами:…
  - Функция *CheckerboardMatrixScatter*:
    - Распределяет матрицу поблочно между процессами решетки процессов

```
// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter (double* pMatrix,
  double* pMatrixBlock, int Size, int BlockSize),
где
- pMatrix       – указатель на матрицу, которую необходимо распределить,
- pMatrixBlock  – указатель на буфер, куда будет помещен матричный блок,
- Size          – порядок матрицы pMatrix,
- BlockSize     – размер матричного блока.
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 5** – Распределение данных между процессами:…
  - Функция *DataDistribution*:
    - Распределяет исходные матрицы A и B поблочно между процессами решетки процессов

```
// Function for data distribution among the processes
void DataDistribution (double* pAMatrix, double* pBMatrix,
  double* pMatrixAblock, double* pBblock, int Size,
  int BlockSize) {
  CheckerboardMatrixScatter(pAMatrix, pMatrixAblock, Size,
    BlockSize);
  CheckerboardMatrixScatter(pBMatrix, pBblock, Size,
    BlockSize);
}
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 5** – Распределение данных между процессами:…
  - Для того, чтобы проконтролировать правильность выполнения этапа распределения данных, реализуйте функцию *TestBlocks*, которая последовательно распечатает матричные блоки со всех процессов:

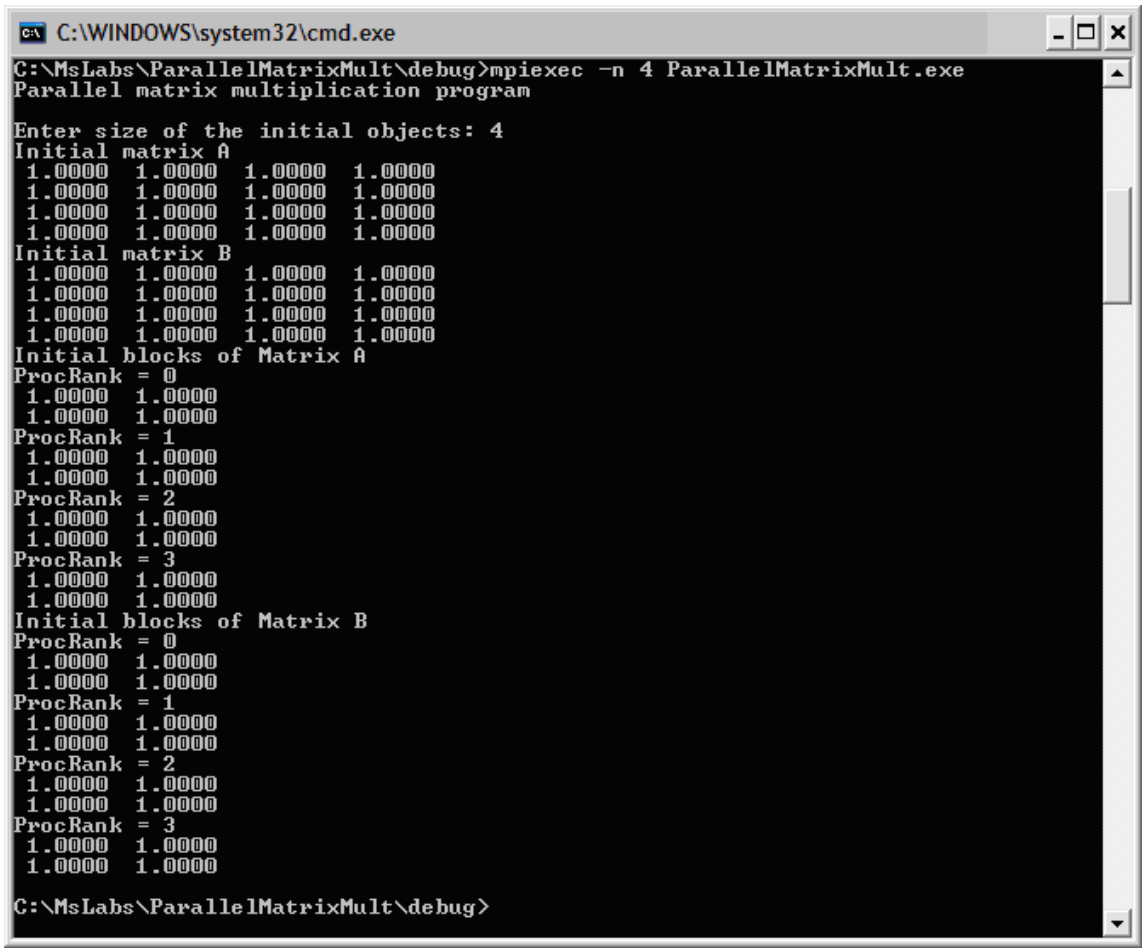
```
// Test printing of the matrix block
void TestBlocks (double* pBlock, int BlockSize, char str[]),
где
-pBlock – указатель на матричный блок, который нужно распечатать,
-BlockSize – размер матричного блока,
-str       – поясняющая строка.
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 5** – Распределение данных между процессами:…
  - Реализуйте функцию *CheckerboardMatrixScatter*,
  - Реализуйте функцию *DataDistribution*,
  - Добавьте вызов функции *DataDistribution* в главную функцию параллельного приложения,
  - Реализуйте функцию *TestBlocks*,
  - После выполнения рассылки матриц, распечатайте матричные блоки при помощи функции *TestBlocks*,
  - Скомпилируйте и запустите приложение. Проверьте его работоспособность.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 5** – Распределение данных между процессами:



Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 6** – Начало реализации параллельного алгоритма матричного умножения:…
  - Функция *ParallelResultCalculation*:
    - Рассылка блоков матрицы *A* вдоль строки решетки процессов,
    - Выполнение умножения блоков,
    - Сдвиг блоков матрицы *B* вдоль столбца решетки процессов.

```
void ParallelResultCalculation (double* pAblock,
  double* pMatrixAblock, double* pBblock, double* pCblock,
  int BlockSize),
где
- pAblock – текущий блок матрицы A на данном процессе,
- pMatrixAblock – исходный блок матрицы A на данном процессе,
- pBblock – текущий блок матрицы B на данном процессе,
- pCblock – блок результирующей матрицы на данном процессе
- BlockSize – размер матричных блоков.
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 6** – Начало реализации параллельного алгоритма матричного умножения:

```
// Parallel matrix multiplication
void ParallelResultCalculation(double* pAblock,
  double* pMatrixAblock, double* pBblock, double* pCblock,
  int BlockSize) {
for (int iter = 0; iter < GridSize; iter ++) {
  // Sending blocks of matrix A to the process grid rows
  ABlockCommunication(iter,pAblock,pMatrixAblock,BlockSize);
  // Block multiplication
  BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);
  // Cyclic shift of B matrix blocks in process grid columns
  BblockCommunication(pBblock, BlockSize, ColComm);
  }
}
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 7** – Рассылка блоков матрицы A:…
  - Функция *ABlockCommunication* для рассылки блоков матрицы *A*:
    - Определяет номер рассылающего процесса по формуле:

      $$Pivot = (i+iter) \bmod GridSize,$$

      где i – номер строки,
    - Копирует содержимое блока *pMatrixAblock* в буфер *pAblock*,
    - Рассылает блок *pAblock* при помощи функции *MPI_Bcast*.

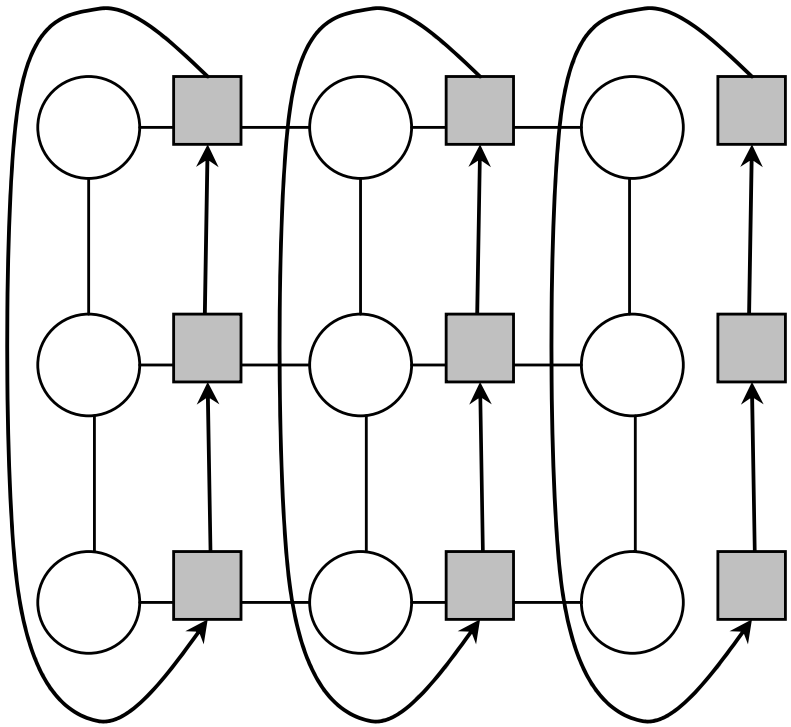
```
// Broadcasting matrix A blocks to process grid rows
void ABlockCommunication (int iter, double *pAblock,
  double* pMatrixAblock, int BlockSize);
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 7** – Рассылка блоков матрицы *A*:
  - Реализуйте функцию *ABlockCommunication*,
  - Закомментируйте вызовы функций, соответствующих этапам умножения матричных блоков и сдвига блоков матрицы *B*, в функции *ParallelResultCalculation*,
  - Добавьте вызов функции *ParallelResultCalculation* в функцию *main*,
  - Для контроля правильности рассылки блоков матрицы *A*, добавьте в функцию *ParallelResultCalculation* функцию *TestBlocks*, которая распечатает блоки матрицы *A* со всех процессов на каждой итерации цикла,
  - Скомпилируйте и запустите приложение, убедитесь в его работоспособности.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 8** – Циклический сдвиг блоков матрицы *B* вдоль столбцов процессорной решетки:…



Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 8** – Циклический сдвиг блоков матрицы *B* вдоль столбцов процессорной решетки:…

```
// Cyclic shift of matrix B blocks in the grid columns
void BblockCommunication (double *pBblock, int BlockSize,
  MPI_Comm ColumnComm);
```

  - Выполнение циклического сдвига возможно при помощи функции *MPI_Senrecv_replace*:

```
int MPI_Sendrecv_replace (void *buf, int count, MPI_Datatype type,
  int dest, int stag, int source, int rtag, MPI_Comm comm,
  MPI_Status* status);
где
- sbuf,scount,stype,dest,stag - параметры передаваемого сообщения,
- rbuf,rcount,rtype,source,rtag - параметры принимаемого сообщения,
- comm   - коммуникатор, в рамках которого выполняется передача данных,
- status – информация о результате выполнения операции.
```

Упражнение 4^ Реализация параллельного алгоритма умножения матриц…

- **Задание 8** – Циклический сдвиг блоков матрицы *B* вдоль столбцов процессорной решетки:
  - Реализуйте функцию *BBlockCommunication*,
  - Добавьте вызов этой функции в функцию *ParallelResultCalculation*,
  - Для контроля правильности рассылки блоков матрицы *B*, добавьте в функцию *ParallelResultCalculation* функцию *TestBlocks*, которая распечатает блоки матрицы *B* со всех процессов на каждой итерации цикла,
  - Скомпилируйте и запустите приложение, убедитесь в его работоспособности.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 9** – Умножение матричных блоков:…
  - Для реализации умножения матричных блоков предназначена функция *BlockMultiplication*:

```
// Function for block multiplication
void BlockMultiplication(double* pAblock, double* pBblock,
  double* pCblock, int Size);
```

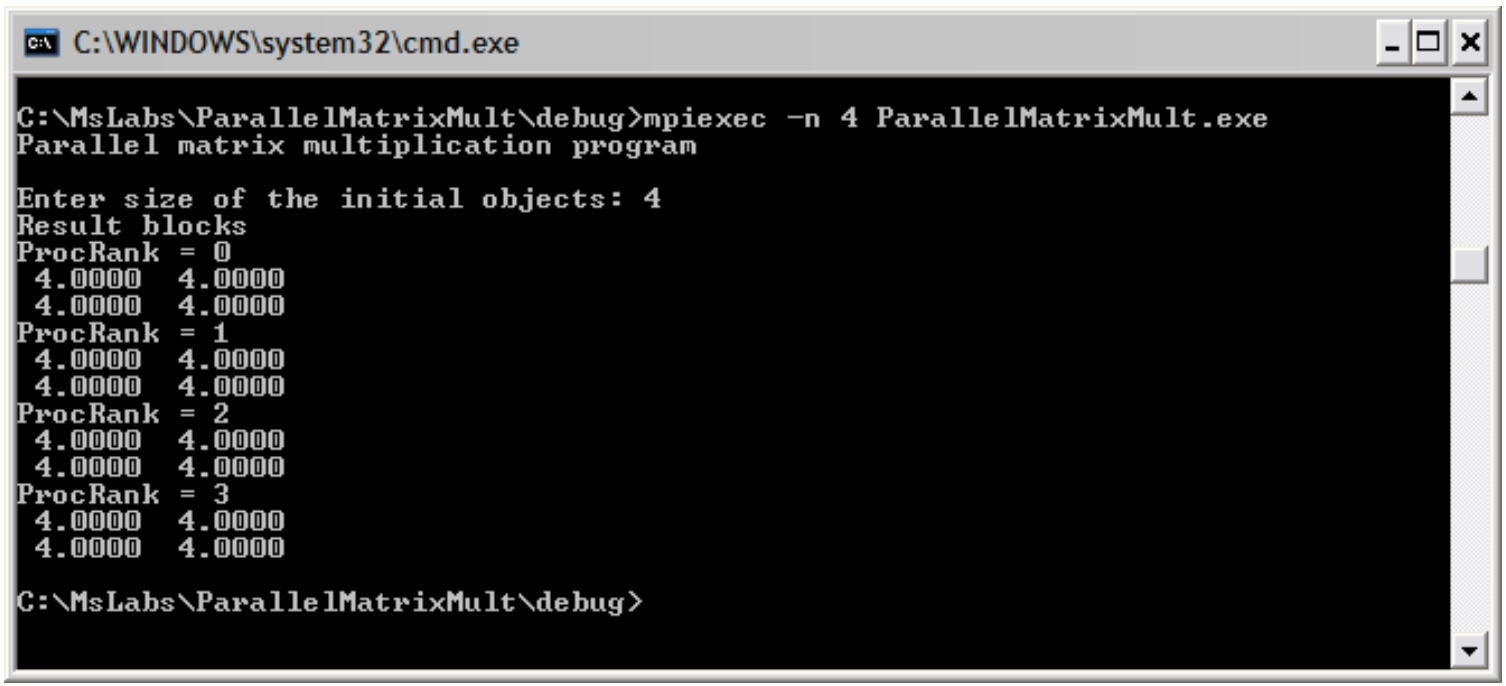
  - Для непосредственного умножения матричных блоков на каждом процессе можно использовать функцию *SerialResultCalculation*, описанную в упражнении 2, с соответствующими аргументами.

Упражнение 4 - Реализация параллельного алгоритма умножения матриц…

- Задание 9 – Умножение матричных блоков^…
  - Реализуйте функцию *BlockMultiplication*,
  - Добавьте вызов функции *BlockMultiplication* в функцию *ParallelResultCalculation*,
  - Закомментируйте вызовы функций, выполняющих отладочную печать внутри функции *ParallelResultCalculation*,
  - В главной функции приложения после выполнения функции *ParallelResultCalculation* распечатайте содержимое блоков результирующей матрицы со всех процессов при помощи функции *TestBlocks*,
  - Скомпилируйте и запустите приложение. Убедитесь в его работоспособности.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 9** – Умножение матричных блоков:
  - Если исходные матрицы задаются при помощи функции *DummyDataInitialization*, то все элементы результирующей матрицы должны быть равны порядку матриц:



Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 10** – Сбор результатов:…
  - Процедура сбора результатов повторяет процедуру распределения исходных данных, но этапы необходимо выполнять в обратном порядке:
    - Осуществить сбор полосы результирующей матрицы из блоков, расположенных на процессорах одной строки процессорной решетки,
    - Собрать матрицу из полос, расположенных на процессорах, составляющих крайний левый столбец процессорной решетки.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 10** – Сбор результатов:…
  - Сбор осуществляется при помощи функции *MPI_Gather*:

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype stype,
  void *rbuf,int rcount,MPI_Datatype rtype, int root,
  MPI_Comm comm),
где
- sbuf, scount, stype - параметры передаваемого сообщения,
- rbuf, rcount, rtype - параметры принимаемого сообщения,
- root – ранг процесса, выполняющего сбор данных,
- comm - коммуникатор, в рамках которого выполняется передача
         данных.
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

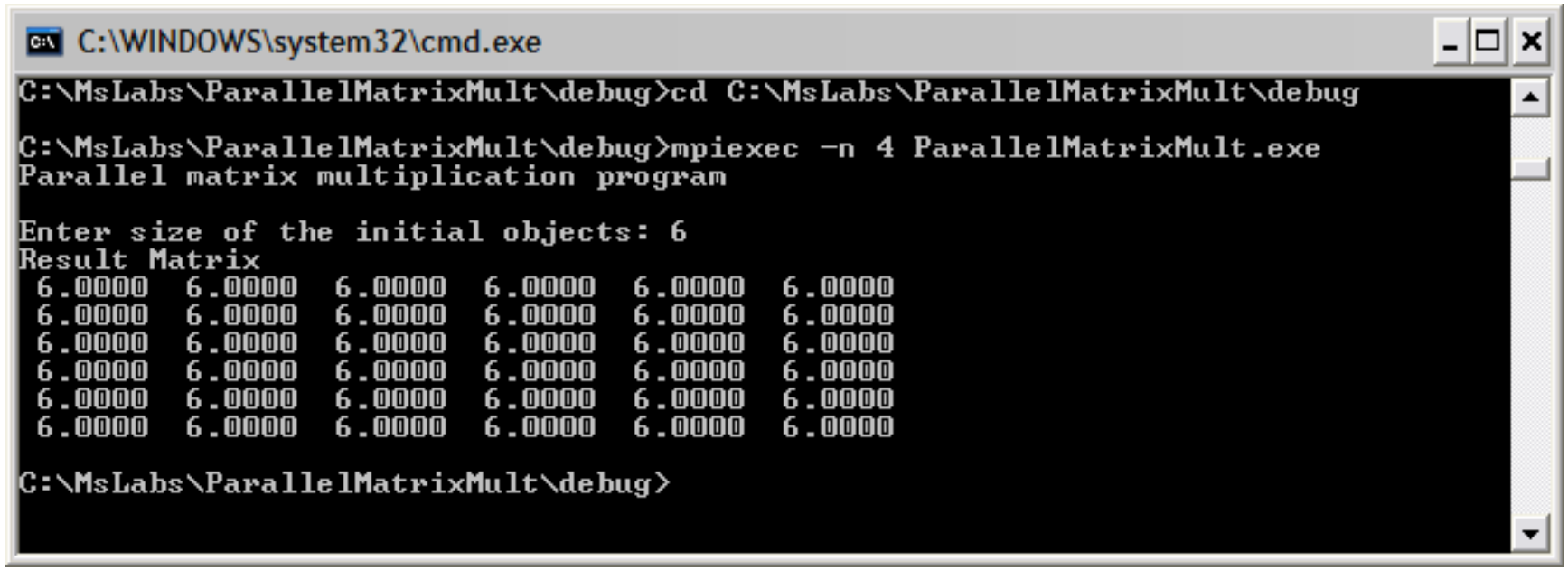
- **Задание 10** – Сбор результатов:…
  - Функция, выполняющая сбор данных:

```
// Function for gathering the result matrix
void ResultCollection (double* pCMatrix, double* pCblock,
  int Size, int BlockSize);
```

  - Реализуйте функцию *ResultCollection*,
  - Добавьте вызов функции *ResultCollection* в функцию *main*,
  - После выполнения сбора данных, распечатайте результирующую матрицу на нулевом процессе,
  - Скомпилируйте и запустите приложение. Убедитесь в том, что приложение работает корректно.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 10** – Сбор результатов:



Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 11** – Проверка правильности работы программы:…
  - Функция *TestResult* сравнивает результаты работы последовательного и параллельного алгоритмов путем поэлементного сравнения полученных матриц

```
void TestResult (double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size),
где
- pAMatrix, pBMatrix – исходные матрицы,
- pCMatrix – результирующая матрица, полученная при помощи
             параллельного алгоритма,
- Size     – порядок матриц.
```

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 11** – Проверка правильности работы программы:…
  - Результатом работы функции *TestResult* является печать диагностического сообщения,
  - Используя эту функцию, можно проверять результат работы параллельного алгоритма независимо от того, насколько велики исходные матрицы и при любых значениях исходных данных.

Упражнение 4: Реализация параллельного алгоритма умножения матриц…

- **Задание 11** – Проверка правильности работы программы:
  - Разработайте функцию для задания элементов исходных матриц при помощи датчика случайных чисел *RandomDataInitialization*,
  - Замените вызов функции *DummyDataIniialization* на *RandomDataInitialization* внутри функции *ProcessInitialization*,
  - Добавьте вызов функции *TestResult* в функцию *main*,
  - Удалите функции отладочной печати,
  - Скомпилируйте и запустите приложение. Убедитесь в том, что реализованный параллельный алгоритм Фокса матричного умножения работает правильно.

Упражнение 4: Реализация параллельного алгоритма умножения матриц

- **Задание 12** – Проведение вычислительных экспериментов:
  - Добавьте вычисление и вывод времени выполнения матричного умножения,
  - Протестируйте работоспособность приложения,
  - Проведите вычислительные эксперименты,
  - Измерьте времена работы умножения матриц при различных количествах исходных данных и различном числе процессов,
  - Определите получаемое ускорение,
  - Вычислите теоретическое время работы параллельного алгоритма,
  - Заполните таблицу результатов вычислений

Заключение

- Рассмотрен один из способов параллельного выполнения матричного умножения – алгоритм Фокса, основанный на блочном разбиении матрицы
- Разработаны приложения, реализующие последовательный и параллельный алгоритмы умножения матриц
- Проведены вычислительные эксперименты, проведено сравнение последовательного и параллельного алгоритмов матричного умножения

Темы заданий для самостоятельной работы

- Измените разработанную реализацию алгоритма Фокса, использовав для рассылки и сборки блоков матриц производный тип данных MPI (см. раздел 4 "Параллельное программирование на основе MPI").
- Изучите параллельный алгоритм умножения матриц, основанный на ленточном разделении матрицы. Разработайте программу, реализующую этот алгоритм.
- Изучите параллельный алгоритм Кэннона умножения матриц, основанный на блочном разделении матрицы. Разработайте программу, реализующую этот алгоритм.

Литература

- **Воеводин В.В., Воеводин Вл.В. (2002).** Параллельные вычисления. – СПб.: БХВ-Петербург.
- **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. (1997). Scalapack Users' Guide (Software, Environments, Tools). Soc for Industrial & Applied Math.
- **Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V. (1999).** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc for Industrial & Applied Math/
- **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.

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)

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

Лабутин Д.Ю., ассистент (система ПараЛаб)

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. (раздел 11, лабораторные работы 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.