

Лабораторная работа 2: Параллельные алгоритмы матричного умножения

Цель лабораторной работы	1
Упражнение 1 – Определение задачи матричного умножения.....	2
Упражнение 2 – Реализация последовательного алгоритма матричного умножения	3
Задание 1 – Открытие проекта SerialMatrixMult.....	3
Задание 2 – Ввод размеров матриц.....	4
Задание 3 – Ввод данных	5
Задание 4 – Завершение процесса вычислений	6
Задание 5 – Реализация матричного умножения	7
Задание 6 – Проведение вычислительных экспериментов.....	8
Упражнение 3 – Разработка параллельного алгоритма матричного умножения.....	10
Определение подзадач	10
Выделение информационных зависимостей.....	10
Масштабирование и распределение подзадач по процессорам	12
Упражнение 4 – Реализация параллельного алгоритма умножения матриц	12
Задание 1 – Открытие проекта ParallelMatrixMult.....	12
Задание 2 – Создание виртуальной декартовой топологии	13
Задание 3 – Определение размеров объектов и ввод исходных данных.....	16
Задание 4 – Завершение процесса вычислений	18
Задание 5 – Распределение данных между процессами	19
Задание 6 – Начало реализации параллельного алгоритма матричного умножения ...	22
Задание 7 – Рассылка блоков матрицы A	23
Задание 8 – Циклический сдвиг блоков матрицы B вдоль столбцов процессорной решетки	24
Задание 9 – Умножение матричных блоков	25
Задание 10 – Сбор результатов	26
Задание 11 – Проверка правильности работы программы.....	27
Задание 12 – Проведение вычислительных экспериментов.....	28
Контрольные вопросы.....	30
Задания для самостоятельной работы	30
Приложение 1. Программный код последовательного приложения для матричного умножения	30
Приложение 2 – Программный код параллельного приложения для матричного умножения	32

Операция умножения матриц является одной из основных задач матричных вычислений. В данной лабораторной работе рассматриваются последовательный алгоритм матричного умножения и параллельный алгоритм Фокса (*the Fox algorithm*), основанный на блочной схеме разделения данных.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет умножение двух квадратных матриц. Выполнение лабораторной работы включает:

- Упражнение 1 – Определение задачи матричного умножения
- Упражнение 2 – Реализация последовательного алгоритма матричного умножения
- Упражнение 3 – Разработка параллельного алгоритма матричного умножения
- Упражнение 4 - Реализация параллельного алгоритма умножения матриц

Примерное время выполнения лабораторной работы: **90 минут**.

При выполнении лабораторной работы предполагается знание раздела 4 "Параллельное программирование на основе MPI", раздела 6 "Принципы разработки параллельных методов" и раздела 8 "Параллельные алгоритмы матричного умножения" учебных материалов курса. Кроме того, предполагается, что выполнена ознакомительная лабораторная работа "Параллельное программирование с использованием MPI" и лабораторная работа 1 "Параллельные алгоритмы матрично-векторного умножения".

Упражнение 1 – Определение задачи матричного умножения

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

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \leq i < m, 0 \leq j < l. \quad (2.1)$$

Как следует из (2.1), каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B (рис. 2.1):

$$c_{ij} = (a_i, b_j^T), a_i = (a_{i0}, a_{i1}, \dots, a_{in-1}), b_j^T = (b_{0j}, b_{1j}, \dots, b_{n-1j})^T. \quad (2.2)$$

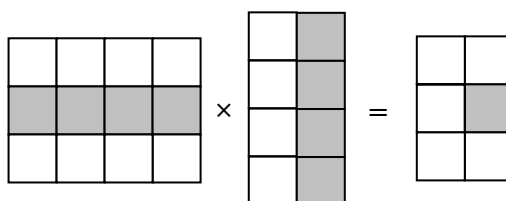


Рис. 2.1. Элемент результирующей матрицы C – это результат скалярного умножения соответствующих строки матрицы A и столбца матрицы B

Так, например, при умножении матрицы A , состоящей из 3 строк и 4 столбцов на матрицу B из 4 строк и 2 столбцов, получается матрица C из 3 строк и 2 столбцов:

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & -1 \\ 2 & 5 \\ -3 & 2 \\ 7 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 3 \\ 5 & -9 \\ -3 & -7 \end{pmatrix}$$

Рис. 2.2. Пример умножения матриц

Тем самым, получение результирующей матрицы C предполагает повторение $m \times l$ однотипных операций по умножению строк матрицы A и столбцов матрицы B . Каждая такая операция включает умножение элементов строки и столбца матриц и последующее суммирование полученных произведений.

Псевдокод для представленного алгоритма умножения матрицы на вектор может выглядеть следующим образом (здесь и далее предполагается, что матрицы, участвующие в умножении, квадратные, то есть имеют размерность $Size \times Size$):

```
// 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];
        }
    }
}
```

Упражнение 2 – Реализация последовательного алгоритма матричного умножения

При выполнении этого упражнения необходимо реализовать последовательный алгоритм матричного умножения. Начальный вариант будущей программы представлен в проекте *SerialMatrixMult*, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера матриц, задание исходных данных, умножения матриц и вывода результатов.

Задание 1 – Открытие проекта SerialMatrixMult

Откройте проект **SerialMatrixMult**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню **File** выполните команду **Open→Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\SerialMatrixMult**,
- Дважды щелкните на файле **SerialMatrixMult.sln** или выбрав файл выполните команду **Open**.

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialMM.cpp**, как это показано на рис. 2.3. После этих действий код, который предстоит в дальнейшем расширить будет открыт в рабочей области Visual Studio.

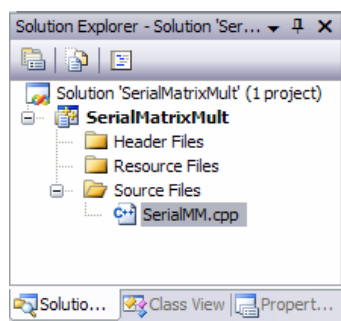


Рис. 2.3. Открытие файла SerialMM.cpp

В файле *SerialMV.cpp* подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции *main*. Эта заготовка содержит объявление переменных и вывод на печать начального сообщения программы.

Рассмотрим переменные, которые используются в основной функции (*main*) нашего приложения. Первые две из них (*pAMatrix* и *pBMatrix*) – это, соответственно, матрицы которые участвуют в матричном умножении в качестве аргументов. Третья переменная *pCMatrix* – матрица, которая должна быть получена в результате умножения. Переменная *Size* определяет размер матриц (предполагаем, что все матрицы квадратные, имеют размерность *Size×Size*).

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

Как и при разработке алгоритмов умножения матрицы на вектор, для хранения матриц используются одномерные массивы, в которых матрицы хранятся построчно. Таким образом, элемент, расположенный на пересечении *i*-ой строки и *j*-ого столбца матрицы, в одномерном массиве имеет индекс *i*Size+j*.

Программный код, который следует за объявлением переменных, это вывод начального сообщения и ожидание нажатия любой клавиши перед завершением выполнения приложения:

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

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет скомпилировать приложение. Если приложение скомпилировано успешно (в нижней части окна Visual Studio появилось сообщение "Rebuild All: 1 succeeded, 0 failed, 0 skipped"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

Сразу после запуска кода, в командной консоли появится сообщение: "Serial matrix multiplication program". Для того, чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод размеров матриц

Для задания исходных данных последовательного алгоритма матричного умножения реализуем функцию *ProcessInitialization*. Эта функция предназначена для определения размера матриц, выделения памяти для исходных матриц *pAMatrix* и *pBMatrix*, и матрицы-результата умножения *pCMatrix*, а также для задания значений элементов исходных матриц. Значит, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, int &Size);
```

На первом этапе необходимо определить размер матриц (задать значение переменной *Size*). В тело функции *ProcessInitialization* добавьте выделенный фрагмент кода:

```
// Function for memory allocation and initialization of matrices' elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, int &Size) {
    // Setting the size of matrices
    printf("\nEnter size of matrices: ");
    scanf("%d", &Size);
    printf("\nChosen matrices' size = %d", Size);
}
```

Пользователю предоставляется возможность ввести размер матриц, который затем считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *Size*. Далее печатается значение переменной *Size* (рис. 2.4).

После строки, выводящей на экран приветствие, добавьте вызов функции инициализации процесса вычислений *ProcessInitialization* в тело основной функции последовательного приложения:

```
void main() {
    double* pAMatrix; // The first argument of matrix multiplication
    double* pBMatrix; // The second argument of matrix multiplication
    double* pCMatrix; // The result matrix
    int Size;          // Size of matrices
    time_t start, finish;
    double duration;

    printf ("Serial matrix multiplication program\n");
    ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);
    getch();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что значение переменной *Size* задается корректно.

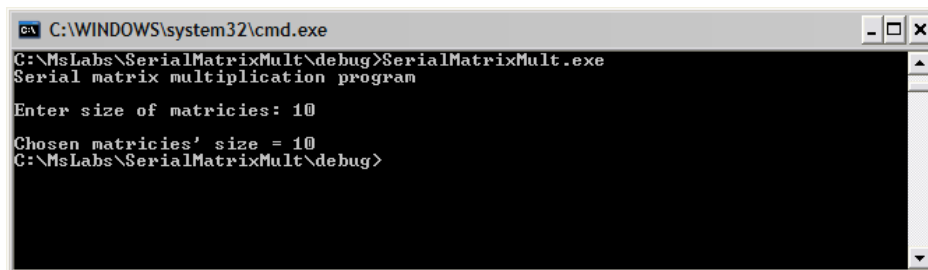


Рис. 2.4. Задание размера объектов

Как и лабораторной работе 1, выполним контроль правильности ввода. Организуем проверку размера матриц и, в случае ошибки (заданный размер является нулевым или отрицательным), продолжим запрашивать размер матриц до тех пор, пока не будет введено положительное число. Для реализации такого поведения поместим фрагмент кода, который производит ввод размера матриц, в цикл с постусловием:

```
// Setting the size of matrices
```

```
do {
    printf("\nEnter size of matrices: ");
    scanf("%d", &Size);
    printf("\nChosen matrices size = %d\n", Size);
    if (Size <= 0)
        printf("\nSize of objects must be greater than 0!\n");
}
while (Size <= 0);
```

Снова скомпилируйте и запустите приложение. Попробуйте ввести неположительное число в качестве размера объектов. Убедитесь в том, что ошибочные ситуации обрабатываются корректно.

Задание 3 – Ввод данных

Функция инициализации процесса вычислений должна осуществлять также выделение памяти для хранения объектов (добавьте выделенный код в тело функции *ProcessInitialization*):

```
// Function for memory allocation and initialization of matrices' elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, int &Size) {
    // Setting the size of matrices
    do {
        <...>
    }
    while (Size <= 0);

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

Далее необходимо задать значения всех элементов исходных объектов: матриц *pAMatrix*, *pBMatrix* и *pCMatrix*. Значения элементов результирующей матрицы до выполнения матричного умножения равны 0. Для задания значений элементов матриц *A* и *B* реализуем еще одну функцию *DummyDataInitialization*. Интерфейс и реализация этой функции представлены ниже:

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

    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++) {
            pAMatrix[i*Size+j] = 1;
            pBMatrix[i*Size+j] = 1;
        }
    }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матриц простым образом: значения всех элементов матриц равны 1. То есть в случае, когда пользователь выбрал размер объектов, равный 4, будут определены следующие матрица и вектор:

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

(задание данных при помощи датчика случайных чисел будет рассмотрено в задании 6).

Вызов функции *DummyDataInitialization* и процедуру заполнения результирующей матрицы нулями необходимо выполнить после выделения памяти внутри функции *ProcessInitialization*:

```
// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, int &Size) {
    // Setting the size of matrices
```

```

do {
    <...>
}
while (Size <= 0);

// Memory allocation
<...>

// Initialization of matrix elements
DummyDataInitialization(pAMatrix, pBMatrix, Size);
for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
}
}

```

Для контроля ввода данных воспользуемся функцией форматированного вывода объектов *PrintMatrix*, которая была разработана при выполнении лабораторной работы 1 и текст которой уже имеется в проекте (подробнее о функции *PrintMatrix* см. задание 3 упражнение 2 лабораторной работы 1). Добавим вызов этой функций для печати объектов *pAMatrix* и *pBMatrix* в основную функцию приложения:

```

// Memory allocation and initialization of matrix elements
ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

// Matrix output
printf ("Initial A Matrix \n");
PrintMatrix(pAMatrix, Size, Size);
printf("Initial B Matrix \n");
PrintMatrix(pBMatrix, Size, Size);

```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным правилам (рис. 2.5). Выполните несколько запусков приложения, задавайте различные размеры матриц.

```

C:\WINDOWS\system32\cmd.exe
C:\MsLabs\SerialMatrixMult\debug>SerialMatrixMult.exe
Serial matrix multiplication program
Enter size of matrices: 4
Chosen matrices' size = 4
Initial A Matrix
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
Initial B Matrix
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
C:\MsLabs\SerialMatrixMult\debug>

```

Рис. 2.5. Результат работы программы при завершении задания 3

Задание 4 – Завершение процесса вычислений

Перед выполнением матрично-векторного умножения сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*. Память выделялась для хранения исходных матриц *pAMatrix* и *pBMatrix*, а также для хранения матрицы - результата умножения *pCMatrix*. Следовательно, эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```

// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
    double* pCMatrix) {
    delete [] pAMatrix;
    delete [] pBMatrix;
    delete [] pCMatrix;
}

```

Вызов функции *ProcessTermination* необходимо выполнить перед завершением той части программы, которая выполняет умножение матрицы на вектор:

```
// Memory allocation and initialization of matrix elements
ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

// Matrix output
printf ("Initial A Matrix \n");
PrintMatrix(pAMatrix, Size, Size);
printf("Initial B Matrix \n");
PrintMatrix(pBMatrix, Size, Size);

// Computational process termination
ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что оно выполняется корректно.

Задание 5 – Реализация матричного умножения

Выполним теперь разработку основной вычислительной части программы. Для выполнения умножения матриц *SerialResultCalculation*, которая принимает на вход исходные матрицы *pAMatrix* и *pBMatrix*, размер этих матриц *Size*, а также указатель на результирующую матрицу *pCMatrix*.

В соответствии с алгоритмом, изложенным в упражнении 1, код этой функции должен иметь следующий вид:

```
// Function for 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];
            }
        }
    }
}
```

Выполним вызов функции вычисления матричного произведения из основной программы. Для контроля правильности выполнения умножения распечатаем результирующую матрицу:

```
// Memory allocation and initialization of matrix elements
ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

// Matrix output
printf ("Initial A Matrix \n");
PrintMatrix(pAMatrix, Size, Size);
printf("Initial B Matrix \n");
PrintMatrix(pBMatrix, Size, Size);

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

// Printing the result matrix
printf ("\n Result Matrix: \n");
PrintMatrix(pCMatrix, Size, Size);

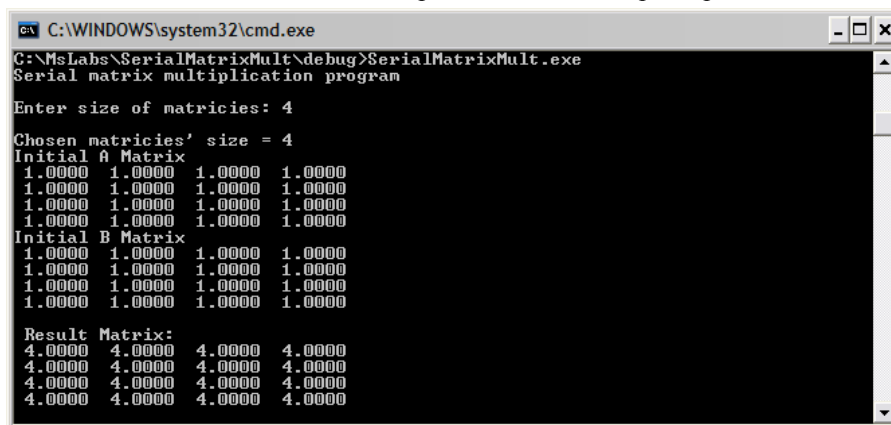
// Computational process termination
ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
```

Скомпилируйте и запустите приложение. 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 (рис. 2.6).

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

Рис. 2.6. Результат матричного умножения

Проведите несколько вычислительных экспериментов, изменяя размеры объектов.



```
C:\WINDOWS\system32\cmd.exe
C:\MsLabs\SerialMatrixMult\debug>SerialMatrixMult.exe
Serial matrix multiplication program
Enter size of matrices: 4
Chosen matrices' size = 4
Initial A Matrix
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
Initial B Matrix
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
Result Matrix:
4.0000 4.0000 4.0000 4.0000
4.0000 4.0000 4.0000 4.0000
4.0000 4.0000 4.0000 4.0000
4.0000 4.0000 4.0000 4.0000
```

Рис. 2.7. Результат выполнения матрично-векторного умножения

Задание 6 – Проведение вычислительных экспериментов

Для последующего тестирования ускорения работы параллельного алгоритма необходимо провести эксперименты по вычислению времени выполнения последовательного алгоритма. Анализ времени выполнения алгоритма разумно проводить для достаточно больших объектов. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов *RandomDataInitialization* (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for random initialization of matrix elements
void RandomDataInitialization (double* pAMatrix, double* pBMatrix,
int Size) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++) {
            pAMatrix[i*Size+j] = rand()/double(1000);
            pBMatrix[i*Size+j] = rand()/double(1000);
        }
}
```

Будем вызывать эту функцию вместо ранее разработанной функции *DummyDataInitialization*, которая генерировала такие данные, что можно было легко проверить правильность работы алгоритма:

```
// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
double* &pCMatrix, int &Size) {
    // Setting the size of matrices
    do {
        <...>
    }
    while (Size <= 0);

    // Memory allocation
    <...>

    // Random initialization of matrix elements
    RandomDataInitialization(pAMatrix, pBMatrix, Size);
    for (int i=0; i<Size*Size; i++) {
        pCMatrix[i] = 0;
    }
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Для определения времени добавьте в получившуюся программу вызовы функций, позволяющие узнать время работы программы или её части. Мы, как и ранее, будем пользоваться функцией:


```
time_t clock(void);
```

Добавим в программный код вычисление и вывод времени непосредственного выполнения умножения матрицы на вектор, для этого поставим замеры времени до и после вызова функции *SerialResultCalculation*:

```
// Matrix multiplication
start = clock();
SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
finish = clock();
duration = (finish-start)/double(CLOCKS_PER_SEC);

// Printing the result matrix
printf ("\n Result Matrix: \n");
PrintMatrix(pCMatrix, Size, Size);

// Printing the time spent by matrix multiplication
printf("\n Time of execution: %f\n", duration);
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими объектами, отключите печать матриц (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу:

Таблица 2.1. Время выполнения последовательного алгоритма умножения матриц

Номер теста	Размер матрицы	Время работы (сек)
1	10	
2	100	
3	500	
4	1000	
5	1500	
6	2000	
7	2500	
8	3000	

Согласно алгоритму вычисления произведения матриц, изложенному в упражнении 1, получение результирующей матрицы предполагает повторение $Size \times Size$ однотипных операций по умножению строк матрицы *pAMatrix* и столбцов матрицы *pBMatrix*. Каждая такая операция включает умножение элементов строки и столбца ($Size$ операций) и последующее суммирование полученных произведений ($Size-1$ операций). Как результат, общее время матричного умножения можно определить при помощи соотношения:

$$T_1 = Size \cdot Size \cdot (2 \cdot Size - 1) \cdot \tau. \quad (2.3)$$

где τ есть время выполнения одной базовой вычислительной операции.

Заполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (2.4). Для вычисления времени выполнения одной операции τ , как и при выполнении лабораторной работы 1, выберем один из экспериментов в качестве образца, время выполнения этого эксперимента поделим на число выполненных операций (число операций может быть вычислено по формуле (2.3)). Таким образом, вычислим время выполнения одной операции. Далее, используя это значение, вычислим теоретическое время выполнения для всех оставшихся экспериментов. Напомним, что время выполнения одной операции, вообще говоря, зависит от размера матриц, участвующих в умножении (см. лабораторную работу 1), поэтому при выборе эксперимента для образца следует ориентироваться на некоторый средний случай.

Вычислите теоретическое время выполнения матричного умножения. Результаты занесите в таблицу:

Таблица 2.2. Сравнение реального времени выполнения последовательного алгоритма умножения матриц со временем, вычисленным теоретически

Время выполнения одной операции τ (сек):			
Номер теста	Размер матрицы	Время работы (сек)	Теоретическое время (сек)
1	10		

2	100		
3	500		
4	1000		
5	1500		
6	2000		
7	2500		
8	3000		

Упражнение 3 – Разработка параллельного алгоритма матричного умножения

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Выполним более подробное рассмотрение данного способа организации вычислений.

Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 7.2 лекционного материала и в упражнении 3 лабораторной работы 1. При таком способе разделения данных исходные матрицы A , B и результирующая матрица C представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера $n \times n$, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен $k \times k$, $k=n/q$). При таком представлении данных операция матричного умножения матриц A и B в блочном виде может быть представлена в виде:

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

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. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы C .

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы A и столбцов матрицы B . Размещение всех требуемых данных в каждой подзадаче неизбежно приведет к дублированию и к значительному росту объема используемой памяти. Как результат, вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы при помощи передачи сообщений. Один из возможных подходов – алгоритм Фокса (*the Fox algorithm*) – подробно рассмотрим в данном упражнении.

Выделение информационных зависимостей

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом в подзадачах на каждой итерации расчетов располагаются только по одному блоку исходных матриц A и B . Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C , т.е. подзадача (i,j) отвечает за вычисление блока C_{ij} – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C .

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче (i,j) располагается 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 подзадачами в ходе выполнения вычислений.

Выполнение параллельного метода включает:

- этап инициализации, на котором каждой подзадаче (i,j) передаются блоки A_{ij} , B_{ij} и обнуляются блоки C_{ij} на всех подзадачах;
- этап вычислений, в рамках которого на каждой итерации l , $0 \leq l < q$, осуществляются следующие операции:
 - для каждой строки i , $0 \leq i < q$, блок A_{ij} подзадачи (i,j) пересылается на все подзадачи той же строки i решетки; индекс j , определяющий положение подзадачи в строке, вычисляется в соответствии с выражением

$$j = (i+l) \bmod q, \quad (2.4)$$

где \bmod есть операция получения остатка от целочисленного деления;

- полученные в результате пересылок блоки A'_{ij} , B'_{ij} каждой подзадачи (i,j) перемножаются и прибавляются к блоку C_{ij}

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

- блоки 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).

Для пояснения приведенных правил параллельного метода на рис. 2.8 приведено состояние блоков в каждой подзадаче в ходе выполнения итераций этапа вычислений (для решетки подзадач 2×2).

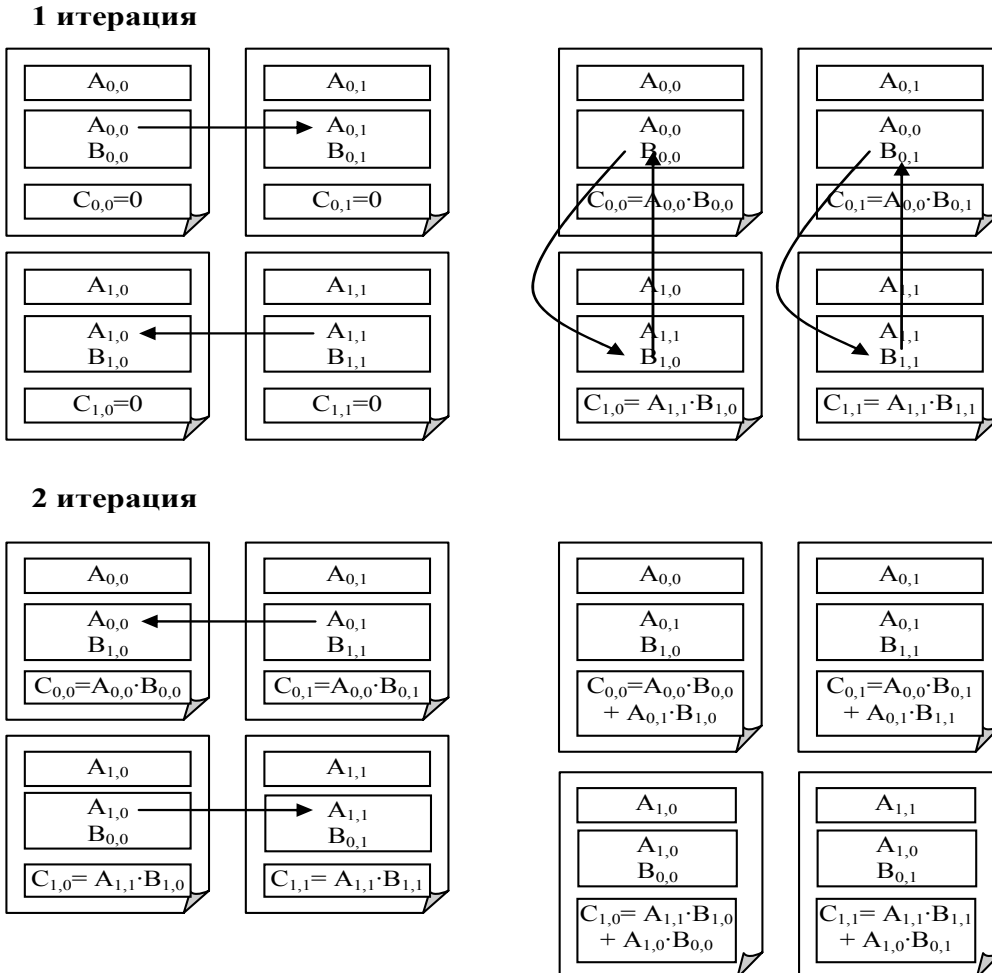


Рис. 2.8.

Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса

Масштабирование и распределение подзадач по процессорам

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 процессоров p . Так, например, в наиболее простом случае, когда число процессоров представимо в виде $p = \delta^2$ (т.е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (т.е. $q = \delta$). 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 процессорами. В более общем случае при произвольных количестве процессоров и размерах матриц балансировка вычислений может отличаться от абсолютно одинаковой, но, тем не менее, при надлежащем выборе параметров может быть распределена между процессорами равномерно в рамках требуемой точности.

Для эффективного выполнения алгоритма Фокса, в котором базовые подзадачи представлены в виде квадратной решетки и в ходе вычислений выполняются операции передачи блоков по строкам и столбцам решетки подзадач, наиболее адекватным решением является организация множества имеющихся процессоров также в виде квадратной решетки. В этом случае можно осуществить непосредственное отображение набора подзадач на множество процессоров – базовую подзадачу (i,j) следует располагать на процессоре $P_{i,j}$. Необходимая структура сети передачи данных может быть обеспечена на физическом уровне, если топология вычислительной системы имеет вид решетки или полного графа.

Упражнение 4 – Реализация параллельного алгоритма умножения матриц

При выполнении этого упражнения Вам будет предложено разработать параллельный алгоритм Фокса для матричного умножения. При работе с этим упражнением Вы

- На практике познакомитесь со схемой организации матричных вычислений на основе блочного разделения данных,
- Получите опыт разработки более сложных параллельных программ,
- Познакомитесь с процедурой создания в MPI коммунитаторов и виртуальных топологий.

Задание 1 – Открытие проекта ParallelMatrixMult

Откройте проект **ParallelMatrixMult**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню File выполните команду **Open→Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\ParallelMatrixMult**,
- Дважды щелкните на файле **ParallelMatrixMult.sln** или подсветите его выполните команду **Open**.

После того, как Вы открыли проект, в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **ParallelMM.cpp**, как это показано на рисунке 2.9. После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

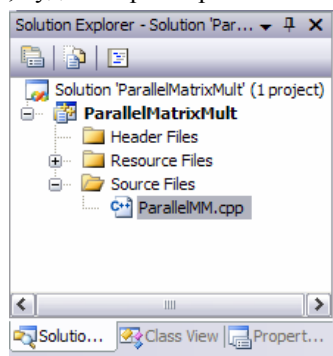


Рис. 2.9. Открытие файла **ParallelMM.cpp** с использованием Solution Explorer

В файле **ParallelMM.cpp** расположена главная функция (*main*) будущего параллельного приложения, которая содержит строки подключения библиотек, объявления необходимых переменных,

вызовы функций инициализации и остановки среды выполнения MPI-программ, функции для определения числа доступных процессов и рангов процессов:

```
int ProcNum = 0;          // Number of available processes
int ProcRank = 0;         // Rank of current process

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 matrix
    int Size;          // Size of matrices
    double Start, Finish, Duration;

    setvbuf(stdout, 0, _IONBF, 0);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    if (ProcRank == 0)
        printf("Parallel matrix multiplication program\n");

    MPI_Finalize();
}
```

Заметим, что переменные *ProcNum* и *ProcRank*, как и при разработке параллельного алгоритма умножения матрицы на вектор (см. описание лабораторной работы 1) были объявлены глобальными.

Также в файле **ParallelMM.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм умножения матриц: *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *PrintMatrix* (подробно о назначении этих функций рассказывается в упражнении 2 данной лабораторной работы). Эти функции можно будет использовать и в параллельной программе. Кроме того, помещены заготовки для функций инициализации процесса вычислений (*ProcessInitialization*) и завершения процесса (*ProcessTermination*).

Скомпилируйте и запустите приложение стандартными средствами Visual Studio. Убедитесь в том, что в командную консоль выводится приветствие: "Parallel matrix multiplication program".

Задание 2 – Создание виртуальной декартовой топологии

Согласно схеме параллельных вычислений, описанной в упражнении 3, для эффективного выполнения алгоритма Фокса необходимо организовать доступные процессы MPI-программы в виртуальную топологию в виде двумерной квадратной решетки. Это возможно только в том случае, когда число доступных процессов является полным квадратом.

Перед тем, как приступить к выполнению параллельного алгоритма проверим, является ли число доступных процессоров полным квадратом: $ProcNum = GridSize \times GridSize$. В случае, когда это условие не выполняется, выведем диагностическое сообщение. Продолжим выполнение приложения только в том случае, когда условие выполняется.

Назовем величину *GridSize* *размером решетки*. Эта величина будет использоваться при разделении и сборе данных, а также при выполнении итераций алгоритма Фокса. Объявим соответствующую глобальную переменную и определим ее значение.

```
int ProcNum = 0;          // Number of available processes
int ProcRank = 0;         // Rank of current process
int GridSize;             // Size of virtual processor grid

void main(int argc, char* argv[]) {
    <...>
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    GridSize = sqrt((double)ProcNum);
    if (ProcNum != GridSize*GridSize) {
        if (ProcRank == 0) {
```

```

        printf ("Number of processes must be a perfect square \n");
    }
}
else {
    if (ProcRank == 0)
        printf("Parallel matrix multiplication program\n");
    // Place the code of the parallel Fox algorithm here
}
MPI_Finalize();
}

```

Реализуем функцию *CreateGridCommunicators*, которая создаст коммуникатор в виде двумерной квадратной решетки, определит координаты каждого процесса в этой решетке, а также создаст коммуникаторы отдельно для каждой строки и каждого столбца.

```

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

```

Для непосредственного создания декартовой топологии (решетки) в MPI предназначена функция:

```

int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims,
    int *periods, int reorder, MPI_Comm *cartcomm),

```

где:

- **oldcomm** - исходный коммуникатор,
- **ndims** - размерность декартовой решетки,
- **dims** - массив длины ndims, задает количество процессов в каждом измерении решетки,
- **periods** - массив длины ndims, определяет, является ли решетка периодической вдоль каждого измерения,
- **reorder** - параметр допустимости изменения нумерации процессов,
- **cartcomm** - создаваемый коммуникатор с декартовой топологией процессов.

Итак, для создания декартовой топологии нужно определить два массива: первый *DimSize* определяет размерности решетки, а второй *Periodic* определяет, является ли решетка периодической вдоль каждого измерения. Поскольку нам необходимо создать двумерную квадратную решетку, то оба элемента *DimSize* должны быть определены следующим образом: $DimSize[0] = DimSize[1] = \sqrt{ProcNum}$. Согласно схеме параллельных вычислений (упражнение 3) нам предстоит осуществлять циклический сдвиг вдоль столбцов процессорной решетки, следовательно, второе измерение декартовой топологии должно обязательно быть периодическим. В результате выполнения функции *MPI_Cart_create* новый коммуникатор сохраняется в переменной *cartcomm*. Значит, нужно объявить переменную для хранения нового коммуникатора и передать ее в качестве аргумента функции *MPI_Cart_create*. Поскольку коммуникатор в виде решетки будет широко использоваться во всех функциях параллельного приложения, объявим соответствующую переменную как глобальную. В библиотеке MPI все коммуникаторы имеют тип *MPI_Comm*.

Добавим в тело функции *CreateGridCommunicators* вызов функции создания решетки:

```

int ProcNum = 0;        // Number of available processes
int ProcRank = 0;       // Rank of current process
int GridSize;           // Size of virtual processor grid
MPI_Comm GridComm;      // Grid communicator
<...>
// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators() {
    int DimSize[2]; // Number of processes in each dimension of the grid
    int Periodic[2]; // =1, if the grid dimension should be periodic

    DimSize[0] = GridSize;
    DimSize[1] = GridSize;

    Periodic[0] = 1;
    Periodic[1] = 1;

    // Determination of the size of the virtual grid
    MPI_Dims_create(ProcNum, 2, DimSize);
}

```

```
// Creation of the Cartesian communicator
MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);
}
```

Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int MPI_Cart_coords(MPI_Comm comm,int rank,int ndims,int *coords),
где:
- comm - коммуникатор с топологией решетки,
- rank - ранг процесса, для которого определяются декартовы координаты,
- ndims - размерность решетки,
- coords - возвращаемые функцией декартовы координаты процесса.
```

Поскольку нами была создана двумерная решетка, каждый процесс в этой решетке имеет две координаты, они соответствуют номеру строки и столбца, на пересечении которых расположен данный процесс. Объявим глобальную переменную – массив для хранения координат каждого процесса и определим эти координаты при помощи функции *MPI_Cart_coords*:

```
int GridSize;           // Size of virtual processor grid
MPI_Comm GridComm;      // Grid communicator
int GridCoords[2];      // Coordinates of current processor in grid
<...>
// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators() {
    <...>
    // Creation of the Cartesian communicator
    MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);

    // Determination of the cartesian coordinates for every process
    MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);
}
```

Теперь создадим коммуникаторы для каждой строки и каждого столбца процессорной решетки. Для этого в библиотеке MPI реализованы функции, позволяющие разделить решетку на подрешетки (подробнее об использовании функции *MPI_Cart_sub* см. раздел 4 "Параллельное программирование на основе MPI" лекционных материалов):

```
int MPI_Cart_sub(MPI_Comm comm, int *subdims, MPI_Comm *newcomm),
где:
- comm - исходный коммуникатор с топологией решетки,
- subdims - массив для указания, какие измерения должны остаться
    в создаваемой подрешетке,
- newcomm - создаваемый коммуникатор с подрешеткой.
```

Объявим коммуникаторы для строки и столбца как глобальные переменные и разделим уже созданный коммуникатор *GridComm*:

```
MPI_Comm GridComm;      // Grid communicator
MPI_Comm ColComm;       // Column communicator
MPI_Comm RowComm;       // Row communicator
<...>
// Creation of two-dimensional grid communicator and
// communicators for each row and each column of the grid
void CreateGridCommunicators() {
    int DimSize[2];      // Number of processes in each dimension of the grid
    int Periodic[2];     // =1, if the grid dimension should be periodic
    int Subdims[2];      // =1, if the grid dimension should be fixed

    <...>

    // Determination of the cartesian coordinates for every process
    MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

    // Creating communicators for rows
    Subdims[0] = 0;      // Dimension is fixed
    Subdims[1] = 1;      // Dimension belong to the subgrid
    MPI_Cart_sub(GridComm, Subdims, &RowComm);
}
```

```

// Creating communicators for columns
Subdims[0] = 1; // Dimension belong to the subgrid
Subdims[1] = 0; // Dimension is fixed
MPI_Cart_sub(GridComm, Subdims, &ColComm);
}

```

Вызовем функцию *CreateGridCommunicators* из основной функции параллельного приложения:

```

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 matrix
    int Size;          // Size of matrices
    double Start, Finish, Duration;

    setvbuf(stdout, 0, _IONBF, 0);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    GridSize = sqrt((double)ProcNum);
    if (ProcNum != GridSize*GridSize) {
        if (ProcRank == 0) {
            printf ("Number of processes must be a perfect square \n");
        }
    }
    else {
        if (ProcRank == 0)
            printf("Parallel matrix multiplication program\n");

        // Grid communicator creating
        CreateGridCommunicators();
    }

    MPI_Finalize();
}

```

Скомпилируйте приложение. Если в процессе компиляции были обнаружены ошибки, исправьте их, сверяя свой программный код с кодом, представленным в данном пособии. 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.

Задание 3 – Определение размеров объектов и ввод исходных данных

На следующем этапе разработки параллельного приложения необходимо задать размеры матриц и выделить память для хранения исходных матриц и их блоков. Согласно схеме параллельных вычислений, на каждом процессе в каждый момент времени располагается четыре матричных блока: два блока матрицы *A*, блок матрицы *B* и блок результирующей матрицы *C* (см. упражнение 3). Определим переменные для хранения матричных блоков и размера этих блоков:

```

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 matrix
    int Size;                   // Size of matrices
    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

    double Start, Finish, Duration;
}

```


Для определения размеров матриц и матричных блоков, выделения памяти для их хранения и определения элементов исходных матриц реализуем функцию *ProcessInitialization*.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
    double* &pMatrixAblock, int &Size, int &BlockSize )
```

Начнем с определения размеров. Для простоты, как и ранее, будем предполагать, что все матрицы, участвующие в умножении, квадратные порядка $Size \times Size$. Размер *Size* должен быть таким, чтобы матрицы можно было разделить между процессами равными квадратными блоками, то есть размер *Size* должен быть кратен размеру процессорной решетки *GridSize*.

Для определения размера, как и при выполнении лабораторной работы 1, организуем диалог с пользователем. Если пользователь вводит некорректное число, ему предлагается повторить ввод. Диалог осуществляется только на *ведущем процессе*. Напомним, что ведущим процессом обычно является процесс, который имеет нулевой ранг в рамках коммуникатора *MPI_COMM_WORLD*. Когда размеры матриц корректно определены, значение переменной *Size* рассылается на все процессы:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
    double* &pTemporaryAblock, int &Size, int &BlockSize ) {
    if (ProcRank == 0) {
        do {
            printf("\nEnter size of the initial objects: ");
            scanf("%d", &Size);
            if (Size%GridSize != 0) {
                printf ("Size of matrices must be divisible by the grid size! \n");
            }
        } while (Size%GridSize != 0);
    }
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

После того, как размеры матриц определены, появляется возможность определить размер матричных блоков, а также выделить память для хранения исходных матриц, матрицы результата, матричных блоков (исходные матрицы существуют только на ведущем процессе):

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
    double* &pMatrixAblock, int &Size, int &BlockSize ) {
    <...>
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    BlockSize = Size/GridSize;

    pAblock = new double [BlockSize*BlockSize];
    pBblock = new double [BlockSize*BlockSize];
    pCblock = new double [BlockSize*BlockSize];
    pMatrixAblock = new double [BlockSize*BlockSize];

    if (ProcRank == 0) {
        pAMatrix = new double [Size*Size];
        pBMatrix = new double [Size*Size];
        pCMatrix = new double [Size*Size];
    }
}
```

Для определения элементов исходных матриц будем использовать функцию *DummyDataInitialization*, которая была разработана нами при реализации последовательного алгоритма матричного умножения:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
    double* &pMatrixAblock, int &Size, int &BlockSize ) {
    <...>
```

```

if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    DummyDataInitialization(pAMatrix, pBMatrix, Size);
}
}

```

Блок результирующей матрицы *pCblock* служит для суммирования результатов умножения блоков матриц *A* и *B*. Для того, чтобы суммы накапливались правильно, необходимо первоначально обнулить все его элементы:

```

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
double* &pCMatrix, double* &pAblock, double* &pBblock, double* pCblock,
double* &pMatrixAblock, int &Size, int &BlockSize) {
    <...>
    if (ProcRank == 0) {
        <...>
    }
    for (int i=0; i<BlockSize*BlockSize; i++) {
        pCblock[i] = 0;
    }
}

```

Вызовем функцию *ProcessInitialization* из основной функции параллельного приложения. Для контроля правильности ввода исходных данных воспользуемся функцией форматированного вывода матриц *PrintMatrix*: распечатаем исходные матрицы *A* и *B* на ведущем процессе.

```

void main(int argc, char* argv[]) {
    <...>
    // Memory allocation and initialization of matrix elements
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
        pCblock, pMatrixAblock, Size, BlockSize );
    if (ProcRank == 0) {
        printf("Initial matrix A \n");
        PrintMatrix(pAMatrix, Size, Size);
        printf("Initial matrix B \n");
        PrintMatrix(pBMatrix, Size, Size);
    }
    MPI_Finalize();
}

```

Скомпилируйте и запустите приложение. Убедитесь в том, что диалог для ввода размеров объектов позволяет ввести только корректное значение размеров объектов. Проанализируйте значения элементов исходных матриц. Если данные задаются верно, то все элементы исходных матриц должны быть приравнены 1 (рис. 2.10).

```

C:\WINDOWS\system32\cmd.exe
C:\MsLabs\ParallelMatrixMult\debug>cd c:\MsLabs\ParallelMatrixMult\debug
C:\MsLabs\ParallelMatrixMult\debug>mpiexec -n 4 ParallelMatrixMult.exe
Parallel matrix multiplication program

Enter size of the initial objects: 4
Initial matrix A
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
Initial matrix B
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
C:\MsLabs\ParallelMatrixMult\debug>_

```

Рис. 2.10. Задание исходных данных

Задание 4 – Завершение процесса вычислений

Для того, чтобы на каждом этапе разработки приложение было завершенным, разработаем функцию для корректной остановки процесса вычислений. Для этого необходимо освободить память, выделенную

динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*. На ведущем процессе выделялась память для хранения исходных матриц *pAMatrix* и *pBMatrix* и память для хранения результирующей матрицы *pCMatrix*, на всех процессах выделялась память для хранения четырех матричных блоков *pMatrixAblock*, *pAblock*, *pBblock*, *pCblock*. Все эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
double* pCMatrix, double* pAblock, double* pBblock, double* pCblock,
double* pMatrixAblock) {
    if (ProcRank == 0) {
        delete [] pAMatrix;
        delete [] pBMatrix;
        delete [] pCMatrix;
    }
    delete [] pAblock;
    delete [] pBblock;
    delete [] pCblock;
    delete [] pMatrixAblock;
}
```

Вызов функции остановки процесса вычислений необходимо выполнить непосредственно перед завершением параллельной программы:

```
// Process termination
ProcessTermination(pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
pCblock, pMatrixAblock);
}
MPI_Finalize();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что приложение работает корректно.

Задание 5 – Распределение данных между процессами

Согласно схеме параллельных вычислений, исходные матрицы, которые нужно перемножить, расположены на ведущем процессе. Ведущий процесс – процесс с рангом 0, расположен в левом верхнем углу процессорной решетки.

Нужно распределить матрицы построчно 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. Матрицы и матричные блоки хранятся в одномерных массивах построчно. Блок матрицы не хранится непрерывной последовательностью элементов в массиве хранения матрицы, следовательно для распределения по блокам невозможно выполнить с использованием стандартных типов данных библиотеки MPI.

Для организации передачи блоков в рамках одной и той же коммуникационной операции можно сформировать средствами MPI производный тип данных. Оставив такой подход для самостоятельной проработки, применим в данной лабораторной работе следующую двухэтапную схему распределения данных. На первом этапе матрица разделяется на горизонтальные полосы, каждая из которых содержит *BlockSize* строк. Эти полосы распределяются на процессы, составляющие нулевой столбец процессорной решетки (рис. 2.11).

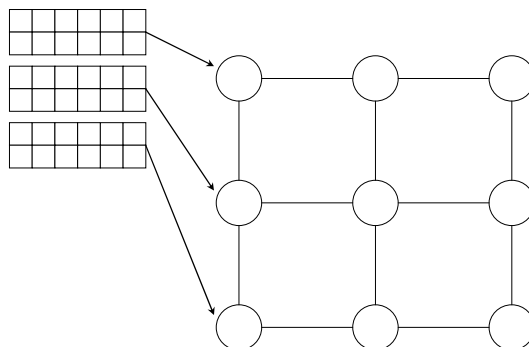


Рис. 2.11. Первый этап распределения данных

Далее каждая полоса разделяется на блоки между процессами, составляющими строки процессорной решетки. Заметим, что распределение полосы на блоки будет осуществляться последовательно через распределение строк полосы при помощи функции *MPI_Scatter* (рис. 2.12).

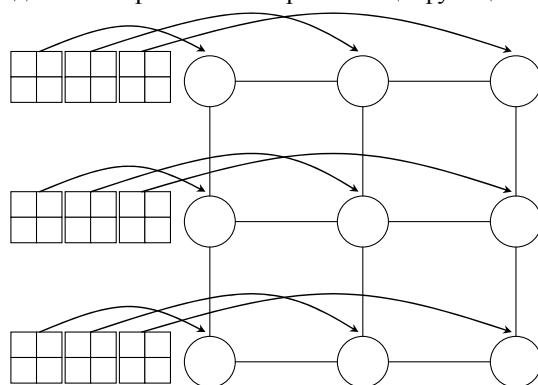


Рис. 2.12. Второй этап разделения данных

Для блочного разделения матрицы между процессами процессорной решетки реализуем функцию *CheckerboardMatrixScatter*.

```
// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter(double* pMatrix, double* pMatrixBlock,
    int Size, int BlockSize);
```

Эта функция принимает в качестве аргументов матрицу, которая хранится на ведущем процессе *pMatrix*, указатель на буфер хранения матричного блока на каждом из процессов параллельного приложения *pMatrixBlock*, размер матрицы *Size* и размер матричного блока *BlockSize*.

На первом этапе необходимо разделить матрицу горизонтальными полосами между процессами, составляющими нулевой столбец решетки процессов. Для этого воспользуемся функцией *MPI_Scatter* в рамках коммуникатора *ColComm*. Заметим, что в параллельном приложении ранее было создано *GridSize* коммуникаторов *ColComm*. Для того, чтобы определить именно тот коммуникатор, который соответствует нулевому столбцу процессорной решетки, воспользуемся значениями, записанными в массиве *GridCoords*. Функцию *MPI_Scatter* будем вызывать только в тех процессах, у которых значение второй координаты равно 0 (то есть процесс расположен в нулевом столбце).

```
// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter(double* pMatrix, double* pMatrixBlock,
    int Size, int BlockSize) {
    double * pMatrixRow = new double [BlockSize*Size];
    if (GridCoords[1] == 0) {
        MPI_Scatter(pMatrix, BlockSize*Size, MPI_DOUBLE, pMatrixRow,
            BlockSize*Size, MPI_DOUBLE, 0, ColComm);
    }
}
```

Отметим, что для временного хранения горизонтальной полосы матрицы используется буфер *pMatrixRow*.

На втором этапе необходимо распределить каждую строчку горизонтальной полосы матрицы вдоль строк процессорной решетки. Снова воспользуемся функцией *MPI_Scatter* в рамках коммуникатора *RowComm*. После выполнения этих действий освободим выделенную память:

```
// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter(double* pMatrix, double* pMatrixBlock,
    int Size, int BlockSize) {
    double * pMatrixRow = new double [BlockSize*Size];
    if (GridCoords[1] == 0) {
        MPI_Scatter(pMatrix, BlockSize*Size, MPI_DOUBLE, MatrixRow,
            BlockSize*Size, MPI_DOUBLE, 0, ColComm);
    }

    for (int i=0; i<BlockSize; i++) {
        MPI_Scatter(&pMatrixRow[i*Size], BlockSize, MPI_DOUBLE,
            &(pMatrixBlock[i*BlockSize]), BlockSize, MPI_DOUBLE, 0, RowComm);
    }
    delete [] pMatrixRow;
```

```
}
```

Для выполнения алгоритма Фокса необходимо поблочно разделить матрицу A (блоки матрицы сохраняются в переменной *pMatrixABlock*) и матрицу B (блоки сохраняются в переменной *pBblock*). Реализуем функцию *DataDistribution*, которая осуществляет разделение указанных матриц:

```
// 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);
}
```

Вызовем функцию распределения данных из главной функции параллельного приложения.

```
void main(int argc, char* argv[]) {
    <...>
    // Memory allocation and initialization of matrix elements
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pABlock, pBblock,
        pCblock, pMatrixABlock, Size, BlockSize );
    if (ProcRank == 0) {
        printf("Initial matrix A \n");
        PrintMatrix(pAMatrix, Size, Size);
        printf("Initial matrix B \n");
        PrintMatrix(pBMatrix, Size, Size);
    }

    // Data distribution among the processes
    DataDistribution(pAMatrix, pBMatrix, pMatrixABlock, pBblock, Size,
        BlockSize);

    MPI_Finalize();
}
```

Для контроля правильности распределения исходных данных снова воспользуемся отладочной печатью. Реализуем функцию, которая будет последовательно распечатывать содержимое матричного блока на всех процессах, назовем эту функцию *TestBlocks*.

```
// Test printing of the matrix block
void TestBlocks (double* pBlock, int BlockSize, char str[]) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (ProcRank == 0) {
        printf("%s \n", str);
    }
    for (int i=0; i<ProcNum; i++) {
        if (ProcRank == i) {
            printf ("ProcRank = %d \n", ProcRank);
            PrintMatrix(pBlock, BlockSize, BlockSize);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
}
```

Вызовем функцию проверки распределения исходных данных из главной функции параллельной программы:

```
// Data distribution among the processes
DataDistribution(pAMatrix, pBMatrix, pMatrixABlock, pBblock, Size,
    BlockSize);
TestBlocks(pMatrixABlock, BlockSize, "Initial blocks of matrix A");
TestBlocks(pBblock, BlockSize, "Initial blocks of matrix B");
}
```

Скомпилируйте приложение. Если в процессе компиляции были обнаружены ошибки, исправьте их, сверяя свой код с программным кодом, представленным в данном пособии. Запустите приложение. Убедитесь в том, что данные распределяются верно (рис. 2.13):

```

C:\WINDOWS\system32\cmd.exe
C:\MsLabs\ParallelMatrixMult\debug>mpirun -n 4 ParallelMatrixMult.exe
Parallel matrix multiplication program
Enter size of the initial objects: 4
Initial matrix A
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
Initial matrix B
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000
Initial blocks of Matrix A
ProcRank = 0
1.0000 1.0000
1.0000 1.0000
ProcRank = 1
1.0000 1.0000
1.0000 1.0000
ProcRank = 2
1.0000 1.0000
1.0000 1.0000
ProcRank = 3
1.0000 1.0000
1.0000 1.0000
Initial blocks of Matrix B
ProcRank = 0
1.0000 1.0000
1.0000 1.0000
ProcRank = 1
1.0000 1.0000
1.0000 1.0000
ProcRank = 2
1.0000 1.0000
1.0000 1.0000
ProcRank = 3
1.0000 1.0000
1.0000 1.0000
C:\MsLabs\ParallelMatrixMult\debug>

```

Рис. 2.13. Распределение исходных данных в случае, когда приложение запускается на 4 процессах и размер матриц равен 4

Измените задание исходных данных. Для определения элементов исходных матриц вместо функции *DummyDataInitialization* примените функцию *RandomDataInitialization*. Скомпилируйте и запустите приложение. Убедитесь в том, что матрицы корректно распределяются между процессами.

Задание 6 – Начало реализации параллельного алгоритма матричного умножения

За выполнение параллельного алгоритма Фокса матричного умножения отвечает функция *ParallelResultCalculation*. В качестве аргументов ей необходимо передать все матричные блоки и размер этих блоков:

```
void ParallelResultCalculation(double* pAblock, double* pMatrixAblock,
double* pBblock, double* pCblock, int BlockSize);
```

Согласно схеме параллельных вычислений, описанной в упражнении 3, для выполнения матричного умножения с помощью алгоритма Фокса необходимо выполнить *GridSize* итераций, каждая из которых состоит из выполнения трех действий:

- рассылка блока матрицы *A* по строке процессорной решетки (для выполнения этого шага реализуем функцию *ABlockCommunication*),
- выполнение умножения матричных блоков (для выполнения умножения матричных блоков можно воспользоваться функцией *SerialResultCalculation*, которая была реализована в ходе разработки последовательного алгоритма умножения матриц),
- циклический сдвиг блоков матрицы *B* вдоль столбца процессорной решетки (функция *BBlockCommunication*).

Значит, код, выполняющий алгоритм Фокса матричного умножения, имеет следующий вид:

```
// Execution of the Fox method
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 blocks of matrix B in process grid columns
    }
}
```

```

    BblockCommunication ( pBblock, BlockSize, ColComm );
}
}

```

Рассмотрим этапы более подробно в ходе отдельных упражнений лабораторной работы.

Задание 7 – Рассылка блоков матрицы A

Итак, в начале каждой итерации *iter* алгоритма для каждой строки процессной решетки выбирается процесс, который будет рассылать свой блок матрицы *A* по процессам соответствующей строки решетки. Номер этого процесса *Pivot* в строке определяется в соответствии с выражением:

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

где *i* – номер строки процессорной решетки, для которой определяется номер рассылającego процесса (для каждого процесса номер строки, в которой он расположен, можно определить, обратившись к первому значению в массиве *GridCoords*), а операция *mod* есть операция вычисления остатка от деления. Таким образом, на каждой итерации рассылającym назначается процесс, у которого значение второй координаты *GridCoords* совпадает с *Pivot*. После того, как номер рассылającego процесса определен, необходимо выполнить широковещательную рассылку блока матрицы *A* по строке. Сделаем это при помощи функции *MPI_Bcast* в рамках коммуникатора *RowComm*. Здесь нам потребуется использование дополнительного блока матрицы *A*: первый блок *pMatrixAblock* хранит тот блок матрицы, который был помещен на данный процесс перед началом вычислений, а блок *pAblock* хранит тот блок матрицы, который принимает участие в умножении на данной итерации алгоритма. Перед выполнением широковещательной рассылки содержимое блока *pMatrixAblock* копируется в буфер *pAblock*, а затем буфер *pAblock* рассылается на все процессы строки.

```

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

    // Defining the leading process of the process grid row
    int Pivot = (GridCoords[0] + iter) % GridSize;

    // Copying the transmitted block in a separate memory buffer
    if (GridCoords[1] == Pivot) {
        for (int i=0; i<BlockSize*BlockSize; i++)
            pAblock[i] = pMatrixAblock[i];
    }

    // Block broadcasting
    MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot, RowComm);
}

```

Оценим правильность выполнения этапа рассылки блоков матрицы *A*. Для этого добавим вызов функции *ParallelResultCalculation* в главную функцию параллельного приложения. Внутри функции *ParallelResultCalculation* прокомментируем вызовы не реализованных пока функций перемножения матричных блоков (*SerialResultCalculation*) и циклического сдвига блоков матрицы *B* (*BblockCommunication*).

Напомним, что сейчас для генерации значений исходных матриц используется функция *RandomDataInitialization* (мы использовали такое задание для проверки правильности выполнения этапа распределения данных). Внутри функции *ParallelResultCalculation* после выполнения рассылки блоков матрицы *A* распечатаем значения, которые хранятся в блоках *pAblock* на всех процессорах:

```

// Execution of the Fox method
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);
        if (ProcRank == 0)
            printf(("Iteration number %d \n", iter);
        TestBlocks(pAblock, BlockSize, "Block of A matrix");

        // Block multiplication
        // BlockMultiplication ( pAblock, pBblock, pCblock, BlockSize );
    }
}

```

```

// Cyclic shift of blocks of matrix B in process grid columns
// BblockCommunication ( pBblock, BlockSize, ColComm );
}
}

```

Скомпилируйте и запустите приложение на 9 процессорах. Проверьте правильность выполнения рассылки блоков матрицы *A*. Для этого сравните блоки, расположенные на процессорах на каждой итерации алгоритма Фокса с выводом, выполненным после выполнения функции *DataDistribution*. Номер блока, который расположен на всех процессорах строки *i* должен вычисляться по формуле (2.4).

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

После выполнения умножения матричных блоков нужно осуществить циклический сдвиг блоков матрицы *B* вдоль столбцов процессорной решетки (рис. 2.14).

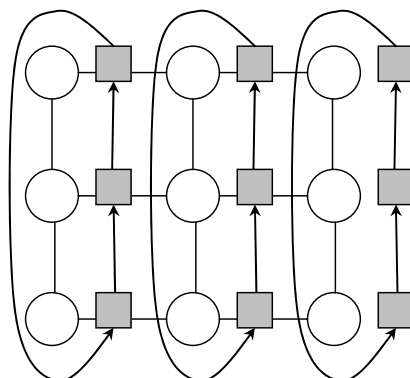


Рис. 2.14. Циклический сдвиг блоков матрицы *B* вдоль столбцов процессорной решетки

Такой сдвиг можно выполнить несколькими разными способами. Наиболее очевидный подход состоит в организации последовательностей передачи и приема матричных блоков при помощи функций *MPI_Send* и *MPI_Receive* (подробнее об этих функциях можно узнать из раздела 4 лекционного материала). Сложность здесь состоит в том, чтобы организовать эту последовательность таким образом, чтобы не возникло тупиковых ситуаций, то есть таких ситуаций, когда один процесс ждет приема сообщения от другого процесса, тот, в свою очередь, от третьего, и так далее.

Достижение эффективного и гарантированного одновременного выполнения операций передачи и приема данных может быть обеспечено при помощи функции *MPI*:

```

int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype stype,int dest,
    int stag, void *rbuf,int rcount,MPI_Datatype rtype,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** - структура данных с информацией о результате выполнения операции.

Как следует из описания, функция *MPI_Sendrecv* передает сообщение, описываемое параметрами (*sbuf, scount, stype, dest, stag*), процессу с рангом *dest* и принимает сообщение в буфер, определяемый параметрами (*rbuf, rcount, rtype, source, rtag*), от процесса с рангом *source*.

В функции *MPI_Sendrecv* для передачи и приема сообщений применяются разные буфера. В случае же, когда сообщения имеют одинаковый тип, в *MPI* имеется возможность использования единого буфера:

```

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);

```

Используем эту функцию для организации циклического сдвига блоков *pBblock* матрицы *B*. Каждый процесс посылает сообщение предыдущему процессу того же столбца процессорной решетки и принимает сообщение от следующего процесса. Процесс, расположенный в нулевой строке процессорной решетки посылает свой блок процессу, расположенному в последней строке (строке с номером *GridSize-1*).


```
// Cyclic shift of matrix B blocks in the process grid columns
void BblockCommunication (double *pBblock, int BlockSize,
    MPI_Comm ColumnComm) {
    MPI_Status Status;
    int NextProc = GridCoords[0] + 1;
    if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
    int PrevProc = GridCoords[0] - 1;
    if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;

    MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
        NextProc, 0, PrevProc, 0, ColumnComm, &Status);
}
```

Оценим правильность выполнения этого этапа. Внутри функции *ParallelResultCalculation* раскомментируем вызов функции циклического сдвига блоков матрицы *B* (*BblockCommunication*). Удалим отладочную печать блоков матрицы *A*. После выполнения рассылки блоков матрицы *B* распечатаем значения, которые хранятся в блоках *pBblock* на всех процессорах:

```
// Execution of the Fox method
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 blocks of matrix B in process grid columns
        BblockCommunication ( pBblock, BlockSize, ColComm );
        if (ProcRank == 0)
            printf(("Iteration number %d \n", iter);
            TestBlocks(pAblock, BlockSize, "Block of B matrix");

    }
}
```

Скомпилируйте и запустите приложение на 9 процессорах. Проверьте правильность выполнения рассылки блоков матрицы *B* (на каждой следующей итерации блоки матрицы *B* должны смещаться на 1 вверх вдоль столбца процессорной решетки). Для этого сравните блоки, расположенные на процессах на каждой итерации алгоритма Фокса с выводом, выполненным после выполнения функции *DataDistribution*.

Задание 9 – Умножение матричных блоков

После того, как блоки матрицы *A* разосланы, необходимо выполнить перемножение блоков *pAblock* и *pBblock*, результат этого умножения прибавить к блоку *pCblock*. Для умножения матричных блоков на каждом процессе необходимо выполнить последовательный алгоритм матричного умножения над матрицами *pAblock* и *pBblock* размера $BlockSize \times BlockSize$ и для этого может быть использована функция *SerialResultCalculation*, разработанная при реализации последовательного алгоритма умножения матриц (упражнение 1).

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

После выполнения *GridSize* итераций алгоритма Фокса на каждом процессе располагается блок результирующей матрицы. Для анализа правильности выполнения алгоритма еще до выполнения сбора данных распечатаем полученные блоки результирующей матрицы *C* при помощи функции *TestBlocks* (удалим из кода параллельного приложения отладочную печать результатов первоначальной рассылки блоков и выполнения итераций алгоритма):

```
void main(int argc, char* argv[]) {
    <...>
    // Memory allocation and initialization of matrix elements
```

```

ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
    pCblock, pMatrixAblock, Size, BlockSize );

DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock,
    Size, BlockSize);

// TestBlocks(pMatrixAblock, BlockSize, "Initial blocks of matrix A");
// TestBlocks(pBblock, BlockSize, "Initial blocks of matrix B");

// Execution of Fox method
ParallelResultCalculation(pAblock, pMatrixAblock, pBblock, pCblock,
    BlockSize);
TestBlocks(pCblock, BlockSize, "Result blocks");

// Process Termination
ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
    pCblock, pMatrixAblock);

MPI_Finalize();
}

```

Для определения элементов исходных матриц внутри функции *ProcessInitialization* будем снова использовать функцию *DummyDataInitialization*. Тогда блоки результирующей матрицы, расположенные на всех процессах, должны состоять из элементов, равных порядку исходных матриц *Size* (рис. 2.15).

Скомпилируйте и запустите приложение. Задавайте различные размеры исходных матриц. Убедитесь в том, что в результате выполнения итераций алгоритма Фокса получаются корректные результаты.

```

C:\WINDOWS\system32\cmd.exe
C:\MsLabs\ParallelMatrixMult\debug>mpiexec -n 4 ParallelMatrixMult.exe
Parallel matrix multiplication program
Enter size of the initial objects: 4
Result blocks
ProcRank = 0
4.0000 4.0000
4.0000 4.0000
ProcRank = 1
4.0000 4.0000
4.0000 4.0000
ProcRank = 2
4.0000 4.0000
4.0000 4.0000
ProcRank = 3
4.0000 4.0000
4.0000 4.0000
C:\MsLabs\ParallelMatrixMult\debug>

```

Рис. 2.15. Анализ частичных результатов, полученных при помощи алгоритма Фокса

Задание 10 – Сбор результатов

Процедура сбора результатов повторяет процедуру распределения исходных данных, разница состоит в том, что этапы необходимо выполнять в обратном порядке. Сначала необходимо осуществить сбор полос результирующей матрицы из блоков, расположенных на процессорах одной строки процессорной решетки. Далее нужно собрать матрицу из полос, расположенных на процессорах, составляющих столбец процессорной решетки.

Для сбора результирующей матрицы будем использовать функцию *MPI_Gather* библиотеки MPI. Эта функция собирает данные со всех процессов в коммуникаторе на одном процессе. Действия, выполняемые этой функцией, противоположны действиям функции *MPI_Scatter*. Функция *MPI_Gather* имеет следующий интерфейс:

```

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.

```

Процедуру сбора результирующей матрицы *C* реализуем непосредственно в функции *ResultCollection*:

```
// Function for gathering the result matrix
void ResultCollection (double* pCMatrix, double* pCblock, int Size,
int BlockSize) {
    double * pResultRow = new double [Size*BlockSize];
    for (int i=0; i<BlockSize; i++) {
        MPI_Gather( &pCblock[i*BlockSize], BlockSize, MPI_DOUBLE,
        &pResultRow[i*Size], BlockSize, MPI_DOUBLE, 0, RowComm);
    }

    if (GridCoords[1] == 0) {
        MPI_Gather(pResultRow, BlockSize*Size, MPI_DOUBLE, pCMatrix,
        BlockSize*Size, MPI_DOUBLE, 0, ColComm);
    }
    delete [] pResultRow;
}
}
```

Добавим вызов функции *ResultCollection* вместо вызова функции тестирования частичных результатов при помощи отладочной печати (*TestBlocks*). Для контроля правильности сбора данных и работы алгоритма в целом, распечатаем результирующую матрицу *pCMatrix* на ведущем процессе с использованием функции *PrintMatrix*.

```
void main(int argc, char* argv[]) {
    <...>
    // Execution of Fox method
    ParallelResultCalculation(pAblock, pMatrixAblock, pBblock, pCblock,
    BlockSize);
    // TestBlocks(pCblock, BlockSize, "Result blocks");

    ResultCollection(pCMatrix, pCblock, Size, BlockSize);
    if (ProcRank == 0) {
        printf("Result matrix \n");
        PrintMatrix(pCMatrix, Size, Size);
    }

    // Process Termination
    ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
    pCblock, pMatrixAblock);

    MPI_Finalize();
}
```

Скомпилируйте и запустите приложение. Оцените правильность работы приложения. Напомним, что если исходные данные генерируются при помощи функции *DummyDataInitialization*, то все элементы результирующей матрицы должны быть равны ее порядку *Size* (рис. 2.16).

```
C:\WINDOWS\system32\cmd.exe
C:\MsLabs\ParallelMatrixMult\debug>cd C:\MsLabs\ParallelMatrixMult\debug
C:\MsLabs\ParallelMatrixMult\debug>mpiexec -n 4 ParallelMatrixMult.exe
Parallel matrix multiplication program
Enter size of the initial objects: 6
Result Matrix
6.0000 6.0000 6.0000 6.0000 6.0000 6.0000
6.0000 6.0000 6.0000 6.0000 6.0000 6.0000
6.0000 6.0000 6.0000 6.0000 6.0000 6.0000
6.0000 6.0000 6.0000 6.0000 6.0000 6.0000
6.0000 6.0000 6.0000 6.0000 6.0000 6.0000
6.0000 6.0000 6.0000 6.0000 6.0000 6.0000
C:\MsLabs\ParallelMatrixMult\debug>
```

Рис. 2.16. Результат работы алгоритма Фокса

Задание 11 – Проверка правильности работы программы

Теперь, после выполнения функции сбора, необходимо проверить правильность выполнения алгоритма. Для этого разработаем функцию *TestResult*, которая сравнит результаты последовательного и параллельного алгоритмов. Для выполнения последовательного алгоритма можно использовать функцию *SerialResultCalculation*, разработанную в упражнении 2. Результат работы этой функции сохраним в матрице *pSerialResult*, а затем поэлементно сравним эту матрицу с матрицей *pCMatrix*, полученной при

помощи параллельного алгоритма Фокса. Получение каждого элемента результирующей матрицы требует выполнения последовательности умножений и сложений дробных чисел. Порядок выполнения этих действий может повлиять на наличие и величину машинной погрешности. Поэтому в данном случае нельзя проверять элементы двух матриц на равенство. Введем допустимую величину расхождения результатов последовательного и параллельного алгоритма *Accuracy*. Матрицы будем считать равными в том случае, когда соответствующие элементы отличаются не более чем на величину допустимой погрешности *Accuracy*.

Функция *TestResult* должна иметь доступ к исходным матрицам *pAMatrix* и *pCMatrix*, а значит может быть выполнена только на ведущем процессе:

```
void TestResult(double* pAMatrix, double* pBMatrix, double* pCMatrix,
int Size) {
    double* pSerialResult; // Result matrix of serial multiplication
    double Accuracy = 1.e-6; // Comparison accuracy
    int equal = 0; // =1, if the matrices are not equal
    int i; // Loop variable

    if (ProcRank == 0) {
        pSerialResult = new double [Size*Size];
        for (i=0; i<Size*Size; i++) {
            pSerialResult[i] = 0;
        }
        SerialResultCalculation(pAMatrix, pBMatrix, pSerialResult, Size);
        for (i=0; i<Size*Size; i++) {
            if (fabs(pSerialResult[i]-pCMatrix[i]) >= Accuracy)
                equal = 1;
        }
        if (equal == 1)
            printf("The results of serial and parallel algorithms "
                "are NOT identical. Check your code.");
        else
            printf("The results of serial and parallel algorithms "
                "are identical.");
        delete [] pSerialResult;
    }
}
```

Результатом работы этой функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельного алгоритма независимо от того, насколько велики исходные объекты при любых значениях исходных данных.

Закомментируйте вызовы функций, использующих отладочную печать, которые ранее использовались для контроля правильности выполнения этапов параллельного приложения. Вместо функции *DummyDataInitialization*, которая генерирует матрицы простого вида, вызовите функцию *RandomDataInitialization*, которая генерирует исходные матрицы при помощи датчика случайных чисел. Скомпилируйте и запустите приложение. Задавайте различные объемы исходных данных. Убедитесь в том, что приложение работает правильно.

Задание 12 – Проведение вычислительных экспериментов

Определим время выполнения параллельного алгоритма. Для этого добавим в программный код замеры времени. Поскольку параллельный алгоритм включает этап распределения данных, вычисления блока частичных результатов на каждом процессе и сбора результата, то отсчет времени должен начинаться непосредственно перед вызовом функции *DataDistribution*, и останавливаться сразу после выполнения функции *ResultCollection*:

```
<...>
Start = MPI_Wtime();
DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock,
    Size, BlockSize);

// Execution of the Fox method
ParallelResultCalculation(pAblock, pMatrixAblock, pBblock, pCblock,
    BlockSize);

ResultCollection(pCMatrix, pCblock, Size, BlockSize);
```

```

Finish = MPI_Wtime();
Duration = Finish-Start;

TestResult(pAMatrix, pBMatrix, pCMatrix, Size);
if (ProcRank == 0) {
    printf("Time of execution = %f\n", Duration);
}

ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
    pCblock, pMatrixAblock);

MPI_Finalize();

```

Очевидно, что таким образом будет распечатано то время, которое было затрачено на выполнение вычислений нулевым процессом. Возможно, что время выполнения алгоритма другими процессами немного от него отличается. Но на этапе разработки параллельного алгоритма мы особое внимание уделили равномерной загрузке (*балансировке*) процессов, поэтому теперь у нас есть основания полагать, что время выполнения алгоритма другими процессами несущественно отличается от приведенного.

Добавьте выделенный фрагмент кода в тело основной функции приложения. Скомпилируйте и запустите приложение. Заполните таблицу:

Таблица 2.3. Время выполнения параллельного алгоритма Фокса матричного умножения и достигнутое ускорение

Номер теста	Размер матриц	Последовательный алгоритм	Параллельный алгоритм			
			4 процесса		9 процессов	
			Время	Ускорение	Время	Ускорение
1	10					
2	100					
3	500					
4	1000					
5	1500					
6	2000					
7	2500					
8	3000					

В графу "Последовательный алгоритм" внесите время выполнения последовательного алгоритма, замеренное при проведении тестирования последовательного приложения в упражнении 2. Для того, чтобы вычислить ускорение, разделите время выполнения последовательного алгоритма на время выполнения параллельного алгоритма. Результат поместите в соответствующую графу таблицы.

Для того, чтобы оценить время выполнения параллельного алгоритма, реализованного согласно вычислительной схеме, приведенной в упражнении 3, можно воспользоваться следующим соотношением:

$$T_p = q[(n^2 / p) \cdot (2n / q - 1) + (n^2 / p)] \cdot \tau + (q \log_2 q + (q - 1))(\alpha + w(n^2 / p) / \beta) \quad (2.5)$$

(подробный вывод этой формулы приведен в разделе 8 "Параллельные алгоритмы матричного умножения" учебных материалов курса). Здесь n – размер объектов, p – количество процессов, q – размер процессорной решетки, τ – время выполнения одной скалярной операции (значение было нами вычислено при тестировании последовательного алгоритма), α – латентность а β – пропускная способность сети передачи данных. В качестве значений латентности и пропускной способности следует использовать величины, полученные при выполнении лабораторной работы "Выполнение заданий под управлением Microsoft Compute Cluster Server 2003".

Вычислите теоретическое время выполнения параллельного алгоритма по формуле (2.5). Результаты занесите в таблицу 2.4:

Таблица 2.4. Сравнение реального времени выполнения параллельного алгоритма со временем, вычисленным теоретически

Номер теста	Размер матриц	4 процесса		9 процессов	
		Модель	Эксперимент	Модель	Эксперимент
1	10				
2	100				

3	500				
4	1000				
5	1500				
6	2000				
7	2500				
8	3000				

Контрольные вопросы

- Насколько сильно отличаются время, затраченное на выполнение последовательного и параллельного алгоритма? Почему?
- Получилось ли ускорение при матрице размером 10 на 10? Почему?
- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

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

Приложение 1. Программный код последовательного приложения для матричного умножения

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix, double* pBMatrix, int Size) {
    int i, j; // Loop variables

    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++) {
            pAMatrix[i*Size+j] = 1;
            pBMatrix[i*Size+j] = 1;
        }
}

// Function for random initialization of matrix elements
void RandomDataInitialization (double* pAMatrix, double* pBMatrix,
    int Size) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++) {
            pAMatrix[i*Size+j] = rand()/double(1000);
            pBMatrix[i*Size+j] = rand()/double(1000);
        }
}

// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, int &Size) {
    // Setting the size of matrices
    do {
```

```

    printf("\nEnter size of matrices: ");
    scanf("%d", &Size);
    printf("\nChosen matrices' size = %d\n", Size);
    if (Size <= 0)
        printf("\nSize of objects must be greater than 0!\n");
}
while (Size <= 0);

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

// Initialization of matrix elements
DummyDataInitialization(pAMatrix, pBMatrix, Size);
for (int i=0; i<Size*Size; i++) {
    pCMatrix[i] = 0;
}
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
    int i, j; // Loop variables
    for (i=0; i<RowCount; i++) {
        for (j=0; j<ColCount; j++)
            printf("%7.4f ", pMatrix[i*RowCount+j]);
        printf("\n");
    }
}

// Function for 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];
    }
}

// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
    double* pCMatrix) {
    delete [] pAMatrix;
    delete [] pBMatrix;
    delete [] pCMatrix;
}

void main() {
    double* pAMatrix; // The first argument of matrix multiplication
    double* pBMatrix; // The second argument of matrix multiplication
    double* pCMatrix; // The result matrix
    int Size; // Size of matrices
    time_t start, finish;
    double duration;

    printf("Serial matrix multiplication program\n");
    // Memory allocation and initialization of matrix elements
    ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

    // Matrix output
    printf ("Initial A Matrix \n");

```

```

PrintMatrix(pAMatrix, Size, Size);
printf("Initial B Matrix \n");
PrintMatrix(pBMatrix, Size, Size);

// Matrix multiplication
start = clock();
SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);
finish = clock();
duration = (finish-start)/double(CLOCKS_PER_SEC);

// Printing the result matrix
printf ("\n Result Matrix: \n");
PrintMatrix(pCMatrix, Size, Size);

// Printing the time spent by matrix multiplication
printf("\n Time of execution: %f\n", duration);

// Computational process termination
ProcessTermination(pAMatrix, pBMatrix, pCMatrix);
}

```

Приложение 2 – Программный код параллельного приложения для матричного умножения

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <mpi.h>

int ProcNum = 0;      // Number of available processes
int ProcRank = 0;     // Rank of current process
int GridSize;         // Size of virtual processor grid
int GridCoords[2];    // Coordinates of current processor in grid
MPI_Comm GridComm;    // Grid communicator
MPI_Comm ColComm;     // Column communicator
MPI_Comm RowComm;     // Row communicator

/// Function for simple initialization of matrix elements
void DummyDataInitialization (double* pAMatrix, double* pBMatrix, int Size){
    int i, j; // Loop variables

    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++) {
            pAMatrix[i*Size+j] = 1;
            pBMatrix[i*Size+j] = 1;
        }
}

// Function for random initialization of matrix elements
void RandomDataInitialization (double* pAMatrix, double* pBMatrix,
    int Size) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++) {
            pAMatrix[i*Size+j] = rand()/double(1000);
            pBMatrix[i*Size+j] = rand()/double(1000);
        }
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {

```



```

    int i, j; // Loop variables
    for (i=0; i<RowCount; i++) {
        for (j=0; j<ColCount; j++)
            printf("%7.4f ", pMatrix[i*ColCount+j]);
        printf("\n");
    }
}

// Function for 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];
    }
}

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

// Creation of two-dimensional grid communicator
// and communicators for each row and each column of the grid
void CreateGridCommunicators() {
    int DimSize[2]; // Number of processes in each dimension of the grid
    int Periodic[2]; // =1, if the grid dimension should be periodic
    int Subdims[2]; // =1, if the grid dimension should be fixed

    DimSize[0] = GridSize;
    DimSize[1] = GridSize;
    Periodic[0] = 0;
    Periodic[1] = 0;

    // Creation of the Cartesian communicator
    MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);

    // Determination of the cartesian coordinates for every process
    MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

    // Creating communicators for rows
    Subdims[0] = 0; // Dimensionality fixing
    Subdims[1] = 1; // The presence of the given dimension in the subgrid
    MPI_Cart_sub(GridComm, Subdims, &RowComm);

    // Creating communicators for columns
    Subdims[0] = 1;
    Subdims[1] = 0;
    MPI_Cart_sub(GridComm, Subdims, &ColComm);
}

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
    double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
    double* &pTemporaryAblock, int &Size, int &BlockSize ) {
    if (ProcRank == 0) {
        do {
            printf("\nEnter size of the initial objects: ");
            scanf("%d", &Size);

```

```

        if (Size%GridSize != 0) {
            printf ("Size of matricies must be divisibile by the grid size!\n");
        }
    }
    while (Size%GridSize != 0);
}
MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

BlockSize = Size/GridSize;

pAblock = new double [BlockSize*BlockSize];
pBblock = new double [BlockSize*BlockSize];
pCblock = new double [BlockSize*BlockSize];
pTemporaryAblock = new double [BlockSize*BlockSize];

for (int i=0; i<BlockSize*BlockSize; i++) {
    pCblock[i] = 0;
}
if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    DummyDataInitialization(pAMatrix, pBMatrix, Size);
    //RandomDataInitialization(pAMatrix, pBMatrix, Size);
}
}

// Function for checkerboard matrix decomposition
void CheckerboardMatrixScatter(double* pMatrix, double* pMatrixBlock,
    int Size, int BlockSize) {
    double * MatrixRow = new double [BlockSize*Size];
    if (GridCoords[1] == 0) {
        MPI_Scatter(pMatrix, BlockSize*Size, MPI_DOUBLE, MatrixRow,
            BlockSize*Size, MPI_DOUBLE, 0, ColComm);
    }

    for (int i=0; i<BlockSize; i++) {
        MPI_Scatter(&MatrixRow[i*Size], BlockSize, MPI_DOUBLE,
            &(pMatrixBlock[i*BlockSize]), BlockSize, MPI_DOUBLE, 0, RowComm);
    }
    delete [] MatrixRow;
}

// Data distribution among the processes
void DataDistribution(double* pAMatrix, double* pBMatrix, double*
    pMatrixAblock, double* pBblock, int Size, int BlockSize) {
    // Scatter the matrix among the processes of the first grid column
    CheckerboardMatrixScatter(pAMatrix, pMatrixAblock, Size, BlockSize);
    CheckerboardMatrixScatter(pBMatrix, pBblock, Size, BlockSize);
}

// Function for gathering the result matrix
void ResultCollection (double* pCMatrix, double* pCblock, int Size,
    int BlockSize) {
    double * pResultRow = new double [Size*BlockSize];
    for (int i=0; i<BlockSize; i++) {
        MPI_Gather( &pCblock[i*BlockSize], BlockSize, MPI_DOUBLE,
            &pResultRow[i*Size], BlockSize, MPI_DOUBLE, 0, RowComm);
    }

    if (GridCoords[1] == 0) {
        MPI_Gather(pResultRow, BlockSize*Size, MPI_DOUBLE, pCMatrix,
            BlockSize*Size, MPI_DOUBLE, 0, ColComm);
    }
}

```

```

    }
    delete [] pResultRow;
}

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

    // Defining the leading process of the process grid row
    int Pivot = (GridCoords[0] + iter) % GridSize;

    // Copying the transmitted block in a separate memory buffer
    if (GridCoords[1] == Pivot) {
        for (int i=0; i<BlockSize*BlockSize; i++)
            pAblock[i] = pMatrixAblock[i];
    }

    // Block broadcasting
    MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot, RowComm);
}

// Cyclic shift of matrix B blocks in the process grid columns
void BblockCommunication (double *pBblock, int BlockSize) {
    MPI_Status Status;
    int NextProc = GridCoords[0] + 1;
    if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
    int PrevProc = GridCoords[0] - 1;
    if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;

    MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
        NextProc, 0, PrevProc, 0, ColComm, &Status);
}

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 blocks of matrix B in process grid columns
        BblockCommunication(pBblock, BlockSize);
    }
}

// Test printing of the matrix block
void TestBlocks (double* pBlock, int BlockSize, char str[]) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (ProcRank == 0) {
        printf("%s \n", str);
    }
    for (int i=0; i<ProcNum; i++) {
        if (ProcRank == i) {
            printf ("ProcRank = %d \n", ProcRank);
            PrintMatrix(pBlock, BlockSize, BlockSize);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
}

void TestResult(double* pAMatrix, double* pBMatrix, double* pCMatrix,
    int Size) {
    double* pSerialResult;    // Result matrix of serial multiplication

```

```

double Accuracy = 1.e-6;    // Comparison accuracy
int equal = 0;              // =1, if the matrices are not equal
int i;                      // Loop variable

if (ProcRank == 0) {
    pSerialResult = new double [Size*Size];
    for (i=0; i<Size*Size; i++) {
        pSerialResult[i] = 0;
    }
    BlockMultiplication(pAMatrix, pBMatrix, pSerialResult, Size);
    for (i=0; i<Size*Size; i++) {
        if (fabs(pSerialResult[i]-pCMatrix[i]) >= Accuracy)
            equal = 1;
    }
    if (equal == 1)
        printf("The results of serial and parallel algorithms are NOT "
               "identical. Check your code.");
    else
        printf("The results of serial and parallel algorithms are "
               "identical.");
}
}

// Function for computational process termination
void ProcessTermination (double* pAMatrix, double* pBMatrix,
    double* pCMatrix, double* pAblock, double* pBblock, double* pCblock,
    double* pMatrixAblock) {
    if (ProcRank == 0) {
        delete [] pAMatrix;
        delete [] pBMatrix;
        delete [] pCMatrix;
    }
    delete [] pAblock;
    delete [] pBblock;
    delete [] pCblock;
    delete [] pMatrixAblock;
}

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 matrix
    int Size;            // Size of matrices
    int BlockSize;       // Sizes of matrix blocks on current process
    double *pAblock;     // Initial block of matrix A on current process
    double *pBblock;     // Initial block of matrix B on current process
    double *pCblock;     // Block of result matrix C on current process
    double *pMatrixAblock;
    double Start, Finish, Duration;

    setvbuf(stdout, 0, _IONBF, 0);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    GridSize = sqrt((double)ProcNum);
    if (ProcNum != GridSize*GridSize) {
        if (ProcRank == 0) {
            printf ("Number of processes must be a perfect square \n");
        }
    }
    else {

```

```

if (ProcRank == 0)
    printf("Parallel matrix multiplication program\n");

// Creating the cartesian grid, row and column communicators
CreateGridCommunicators();

// Memory allocation and initialization of matrix elements
ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
    pCblock, pMatrixAblock, Size, BlockSize );

DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
    BlockSize);

// Execution of Fox method
ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
    pCblock, BlockSize);

ResultCollection(pCMatrix, pCblock, Size, BlockSize);

TestResult(pAMatrix, pBMatrix, pCMatrix, Size);

// Process Termination
ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
    pCblock, pMatrixAblock);
}

MPI_Finalize();
}

```