Learning Lab 2: Parallel Algorithms of Matrix Multiplication

Matrix multiplication is one of the basic matrix computation problems. This lab considers the sequential matrix multiplication algorithm and the Fox parallel algorithm based on chessboard block scheme of data partitioning.

Lab Objective

The objective of this lab is to develop a parallel program for matrix multiplication. The lab assignments include:

- Exercise 1 – State the matrix multiplication problem.
- Exercise 2 – Code the serial matrix multiplication program.
- Exercise 3 – Develop the parallel matrix multiplication algorithm.
- Exercise 4 – Code the parallel matrix multiplication program.

Estimated time to complete this lab: **90 minutes**.

The lab students are assumed to be familiar with the related sections of the training material: Section 4 “Parallel programming with MPI”, Section 6 “Principles of parallel method development” and Section 8 “Parallel algorithms of matrix multiplication”. Besides, the preliminary lab “Parallel programming with MPI” and Lab 1 “Parallel algorithms of matrix-vector multiplication” are assumed to have been done.

Exercise 1 – Stating the Matrix Multiplication Problem

Multiplying the matrix *A* of size $m \times n$ by the matrix *B* of size $n \times l$ leads to obtaining the matrix *C* of size $m \times l$ with each matrix *C* element defined according to the expression:

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

As it can be seen in (2.1), each element of the result matrix *C* is the scalar product of the corresponding row of the matrix *A* and the column of the matrix *B*:

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

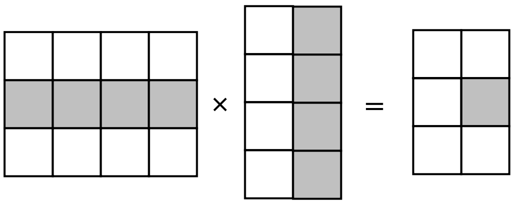


Figure. 2.1. The Element of the Result Matrix *C* is the Result of the Scalar Multiplication of the Corresponding Matrix *A* Row of the Matrix *A* and the Column of the Matrix *B*

Thus, for instance, if the matrix *A*, which consists of 3 rows and 4 columns, is multiplied by the matrix *B*, consisting of 4 rows and 2 columns, we obtain the matrix *C*, which consists of 3 rows and 2 columns:

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

Figure. 2.2. The Example of Matrix Multiplication

Thus, in order to obtain the result matrix *C* $m \times l$ operations of multiplying rows of the matrix *A* by columns of the matrix *B* should be executed. Each operation of this type includes multiplying row and column elements and further summing of the obtained products.

The pseudo-code for implementation of the matrix-vector multiplication may look as follows (hereafter we assume that the matrices participating in multiplication are square, i.e. are of size *Size*×*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];
    }
  }
}
```

Task 2 – Code the Seriial Matrix Multiplication Program

In order to do this Exercise, you should implement the serial matrix multiplication algorithm. The initial variant of the program to be developed is given in the project *SerailMatrixMult*, which contains a part of the source code and provides the necessary project parameters. While doing the Exercise you should add the code to input the matrix size, set the initial data, multiply the matrices and output the result matrix to the given version of the serial program.

Task 1 – Open the project SerialMatrixMult

Open the project *SerialMatrixMult* using the following steps:

- Start the application **Microsoft Visual Studio 2005**, if it has not been started yet,
- Execute the command **Open→Project/Solution** in the menu **File,**
- Choose the folder **c:\MsLabs\SerialMatrixMult** in the dialog window **Open Project,**
- Make the double click on the file **SerialMatrixMult.sln** or execute the command **Open** after choosing the file**.**

After the project has been open in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *SerialMM.cpp*, as it is shown in Figure 2.3. After that, the code, which has to be enhanced, will be opened in the workspace of the Visual Studio.

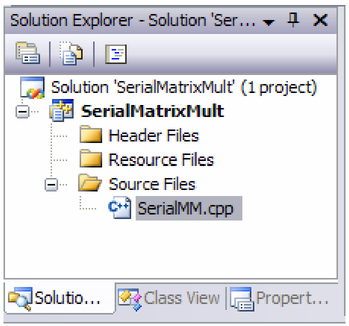


Figure. 2.3. Opening the File SerialMM.cpp

The file *SerialMV.cpp* provides access to the necessary libraries and also contains the initial version of the head function of the program – the function *main*. It provides the possibility to declare the variables and to print out the initial program message.

Let us consider the variables, which are used in the main function of the application. The first two of them (*pAMatrix* and *pBMatrix*) are correspondingly the matrices used in matrix multiplication as arguments. The third variable *pCMatrix* is the matrix to be obtained as a result of multiplication. The variable *Size* defines the matrix size (let us assume that all the matrices are square of *Size*×*Size*).

```
  double* pAMatrix;  // First argument of matrix multiplication
  double* pBMatrix;  // Second argument of matrix multiplication
  double* pCMatrix;  // Result matrix
  int Size;          // Size of matrices
```

As in case of developing the matrix-vector multiplication algorithm, we use one-dimensional arrays, where matrices are stored rowwise. Thus, the element located at the intersection of *i*-th row and *j*-th column of the matrix in one-dimensional array has the index *i*Size+j*.

The program code, which follows the declaration of variables, is the output of the initial message and waiting for pressing any button before the accomplishment of the application execution:

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

Now it is possible to make the first application run. Execute the command **Rebuild Solution** in the menu **Build.** This command makes possible to compile the application. If the application is compiled successfully (in the lower part of the Visual Studio window there is the following message: "`Rebuild All: 1 succeeded, 0 failed, 0 skipped`"), press the key **F5** or execute the command **Start Debugging** of the menu **Debug**.

Right after the code start the following message will appear in the command console:

“`Serial matrix multiplication program`".

Press any key to exit the program.

Task 2 – Input the Matrix Size

In order to input the initial data of the serial matrix multiplication program, we will develop the function *ProcessInitialization*. This function is aimed at inputting the matrix size, allocating the memory for the initial matrices *pAMatrix* and *pBMatrix* and the result matrix *pCMatrix*, setting the element values of the initial matrices. Thus, the function should have the following heading:

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

At the first stage it is necessary to input the matrix size (to set the value of the variable *Size*). The following bold marked code should be added to the function *ProcessInitialization*:

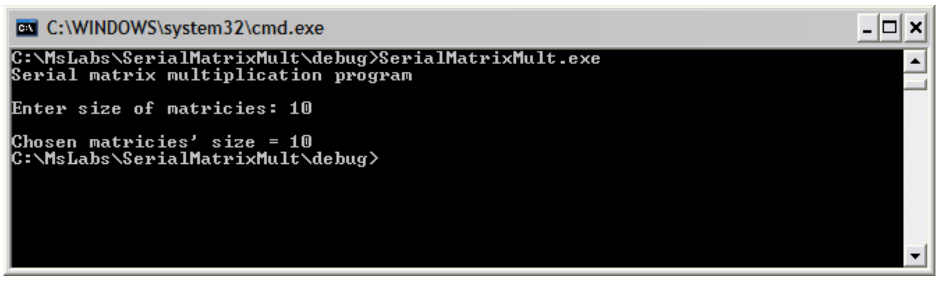
```
// Function for memory allocation and initialization of matrix elements
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
  // Setting the size of matrices
  printf("\nEnter the size of matrices: ");
  scanf("%d", &Size);
  printf("\nChosen matrices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

The user can enter the matrix size, which is read from the standard input stream *stdin* and stored in the integer variable *Size*. After that the value of the variable *Size* is printed (Figure 2.4).

After the line of the initial message you should add the call of the function, which initializes the process computations *ProcessInitialization* to the main function:

```
void main() {
  double* pAMatrix;  // First argument of matrix multiplication
  double* pBMatrix;  // Second argument of matrix multiplication
  double* pCMatrix;  // 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();
}
```

Compile and run the application. Make sure that the value of the variable *Size* is set correctly.



```
C:\MsLabs\SerialMatrixMult\debug>SerialMatrixMult.exe
Serial matrix multiplication program

Enter size of matricies: 10

Chosen matricies' size = 10
C:\MsLabs\SerialMatrixMult\debug>
```

Figure. 2.4. Setting the Matrix Size

We should control the input correctness as we have done in Lab 1. Let us arrange the check up of the matrix size and in case there is an error (i.e. the size is zero or negative) we will continue to ask for the matrix size until a positive number is entered. In order to implement this behavior we will insert the code, which inputs the matrix size, to the loop with the following condition:

```

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

Compile and run the application again. Try to enter a non-positive number as an matrix size. Make sure that invalid situations are processed correctly.

Task 3 – Input the Initial Data

The initialization function must also provide memory allocation for storing the matrices. Add the bold marked code to the function *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
  pAMatrix = new double [Size*Size];
  pBMatrix = new double [Size*Size];
  pCMatrix = new double [Size*Size];
}
```

Further, it is necessary to set the values of all the matrix elements: the matrices *pAMatrix*, *pBMatrix* and *pCMatrix*. The values of the result matrix elements before the execution of matrix multiplication are equal to zero. In order to set the values of the matrix *A* and matrix *B* elements, we will develop the function *DummyDataInitialization*. The heading and the implementation of the function are given below:

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

As it can be seen from the given code, this function provides setting matrix elements in rather a simple way: the values of all matrix elements are equal to 1. That is in case when the user chooses the matrix size equal to 4, the matrices will be determined as follows:

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

(setting the data by means of a random number generator will be discussed in Task 6).

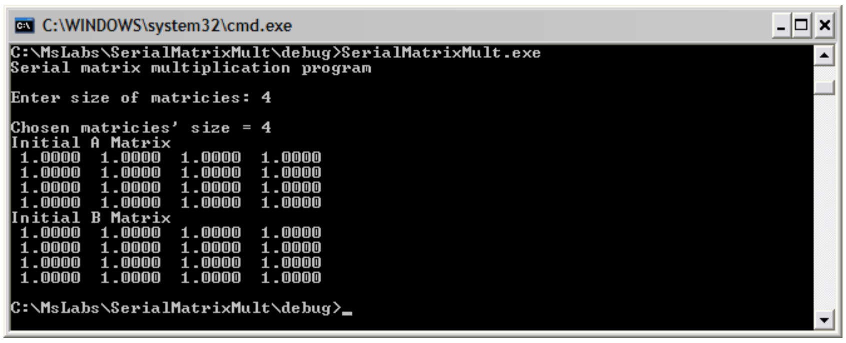
Add the call of the function *DummyDataInitialization* and the procedure of filling the result matrix with zeros to the function *ProcessInitialization* after allocating memory inside:

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

  // Memory allocation
  <…>

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

In order to control the data input we will make use of the formatted matrix output function *PrintMatrix*, which was developed in the course of the execution of Lab 1. The code of the function is available in the project (more details about the function *PrintMatrix* are given in Task 3 of Exercise 2, Lab 1). Let us add the call of the function for printing out the objects *pAMatrix* and *pBMatrix* to the main function of the application:

```
  // 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);
```

Compile and run the application. Make sure that the data input is executed according to the above-described rules (Figure 2.5). Run the application several times setting various matrix sizes



```
C:\MsLabs\SerialMatrixMult\debug>SerialMatrixMult.exe
Serial matrix multiplication program

Enter size of matricies: 4

Chosen matricies' 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>_
```

Figure. 2.5. The Result of Program Execution after Completion of Task 3

Task 4 – Terminate the Program Execution

Let us first develop the function for correct program termination, before implementing the matrix-vector multiplication. For this purpose it is necessary to deallocate the memory, which has been dynamically allocated in the course of the program execution. Let us develop the corresponding function *ProcessTermination*. The memory has been allocated for storing the initial matrices *pAMatrix* and *pBMatrix*, and also for storing the multiplication product *pCMatrix*. These matrices, consequently, should be given to the function *ProcessTermination* as arguments:

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

The function *ProcessTermination* should be called at the end of the main function:

```
  // 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);
```

Compile and run the application. Make sure it is being executed correctly.

Task 5 – Implement the Matrix Multiplication

Let us develop the main computational part of the program. In order to multiply matrices the function *SerialResultCalculation* is used. It gets the initial matrices *pAMatrix* and *pBMatrix*, the size of the matrices *Size*, and the result matrix *pCMatrix* as input parameters.

In accordance with the algorithm given in Exercise 1, the code of the function should be the following:

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

Let us call the function of matrix multiplication computation from the main program. In order to control the correctness of multiplication we will print out the result vector:

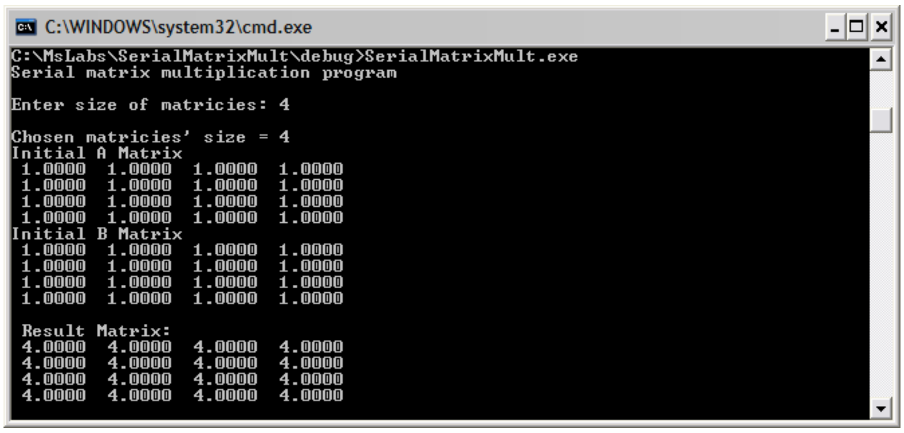
```
  // Memory allocation and initialization of matrix elements
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

  // 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);
```

Compile and run the application. Analyze the results of the matrix multiplication algorithm. If the algorithm is executed correctly, the values of the elements of the result matrix must be equal to the order of the matrix (see Figure 2.6).

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

Figure. 2.6. The Result of Matrix Multiplication

Carry out several computational experiments, changing the matrix sizes.



```
C:\MsLabs\SerialMatrixMult\debug>SerialMatrixMult.exe
Serial matrix multiplication program

Enter size of matricies: 4

Chosen matricies' 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
```

Figure. 2.7. The Result of Matrix Multiplication

Task 6 – Carry out the Computational Experiments

In order to test the speed up of the parallel algorithm functioning, it is necessary to carry out experiments on calculating the sequential algorithm execution time. It is reasonable to analyze the algorithm execution time for considerably large matrices. We will set the elements of large matrices and vectors by means of a random data generator. For this purpose we will develop the function *RandomDataInitialization* (the random number generator is initialized by the current time value):

```
// 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);
    }
}
```

Let us call this function instead of the function *DummyDataInitialization*, which has been developed previously. The function *DummyDataInitialization* generated the data, which made possible to check the algorithm operation correctness easily.

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

  // Memory allocation
  <…>

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

Compile and run the application. Make sure that the data is randomly generated.

In order to determine the time, add the calls of the functions, which make possible to find out the program execution time or the execution time for a part of the program, to the resulting program. As previously we will use the following function:

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

Let us add the computation and the output of the execution time of matrix-vector multiplication to the program code. For this purpose we will clock in before and after the call of the function *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);
```

Compile and run the application. In order to carry out the computational experiments with large matrices, eliminate matrix printing (transform the corresponding code lines into comment). Carry out the computational experiments and register the results in the following table:

Table 2.1. The Execution Time of the Serial Matrix Multiplication Program

Test Number	Matrix Size	Execution Time (sec)
1	10	
2	100	
3	500	
4	1,000	
5	1,500	
6	2,000	
7	2,500	
8	3,000	

In accordance with the computational algorithm of matrix-vector multiplication given in Exercise 1, obtaining the result matrix requires the *Size*×*Size* operations of multiplying the rows of the matrix *pAMatrix* by the columns of the matrix *pBMatrix*. Each operation of this type includes multiplying the row elements by column elements (*Size* operations) and further summing up of the obtained products (*Size-1* operations). As a result, the total time of matrix multiplication may be determined by means of the following expression:

$$T_1 = Size \cdot Size \cdot (2 \cdot Size - 1) \cdot \tau\ . \tag{2.3}$$

where τ is the execution time of the basic computational operation.

Let us fill out the table of comparison of the experiment execution time to the time, which may be obtained according to the formula (2.3). In order to compute the execution time τ of a single operation, as in the course of execution of Lab 1, we will apply the following technique: choose one of the experiments as a pivot one. Let us divide the execution time of the pivot experiment by the number of the executed operations (the number of the operations may be calculated using formula (2.3)). Thus, we will calculate the execution time of a single operation. Then using this value we will calculate the theoretical execution time for the remaining experiments. It should be noted that the execution time of a single operation depends generally on the size of the matrices involved in multiplication (see Lab 1). That is why choosing the experiment to be used as the pivot one, we should be oriented at some average case.

Calculate the theoretical execution time of matrix multiplication. Write the results in the following table:

Table 2.2. The Comparison of the Experiment Execution Time for the Serial Matrix Multiplication Program to the Execution Time, which has been Computed Theoretically

Basic Computational Operation Execution Time τ (sec):			
Test number	Matrix Size	Execution Time (sec)	Theoretical Execution Time (sec)
1	10		
2	100		
3	500		
4	1,000		
5	1,500		
6	2,000		

7	2,500		
8	3,000		

Exercise 3 – Develop the Parallel Matrix Multiplication Algorithm

The chessboard block matrix presentation along with presenting matrices as sets of rows and columns (stripes) is widely used in the development of parallel methods of matrix multiplication. Let us discuss this method of matrix decomposition in detail.

Subtask Definition

The chessboard block scheme of matrix partitioning is described in detail in subsection 7.2 of the training material and Exercise 3 of Lab 1. In case of this method of data distribution the initial matrices *A, B* and the result matrix *C* are presented as sets of blocks. To simplify the further explanations we will assume all the matrices are square of $n \times n$ size, the number of vertical blocks and the number of horizontal blocks are the same and are equal to *q* (i.e. the size of all block is equal to $k \times k$, $k=n/q$). In case of this data presentation method multiplying the matrices *A* and *B* as blocks may be presented as follows:

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

where each block C_{ij} of matrix *C* is defined in accordance with the expression:

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

In case of chessboard block data partitioning it is natural to define the basic computational subtasks on the basis of the computations performed over the matrix blocks. With regards to this we define the basic subtask as the procedure of computing of the elements of a block of the matrix *C*.

To perform all the necessary computations the basic subtasks should have the corresponding sets of rows of the matrix *A* and columns of the matrix *B*. The allocation of all the necessary data in each subtask will inevitably lead to data doubling and to a considerable increase of the size of memory used. As a result, the computations must be arranged in such a way that the subtasks should contain only a part of the data necessary for computations at any given moment, and the access to the rest of the data should be provided by means of data transmission. One of the possible approaches (*the Fox algorithm*) will be discussed further in this Exercise.

Analysis of Information Dependencies

So, parallel computations for matrix multiplication are based on chessboard block data distribution. Also two conditions take place: 1) basic subtasks is responsible for computation of separate blocks of the matrix *C*; 2) each subtask contains only one block of the matrix *A* and one block of the matrix *B* at each iteration. The indices of the blocks of the matrix *C* contained in the subtasks are used for enumeration of the subtasks. Thus, subtask *(i,j)* computes block C_{ij}. So the set of subtasks forms a square grid, which corresponds to the structure of the block presentation of the matrix *C*.

In accordance with the Fox algorithm each basic subtasks *(i,j)* contains four matrix blocks:

- The block C_{ij} of the matrix *C*, computed by the subtask,
- The block A_{ij} of the matrix *A*, located in the subtask before the beginning of computations,
- Blocks A'_{ij}, B'_{ij} of the matrices *A* and *B*, obtained by the subtask in the course of computations.

Parallel algorithm execution includes:

- The initialization stage. Each subtask *(i,j)* obtains blocks A_{ij}, B_{ij}. All elements of blocks C_{ij} in all subtasks are set to zero;
- The computation stage. At this stage the following operations are carried out at each iteration *l*, $0 \le l < q$:
  - For each row *i*, $0 \le i < q$, the block A_{ij} of subtask*(i,j)* is transmitted to all the subtasks of the same grid row; index *j*, which defines the position of the subtask in the row, is computed according to the following expression:

    $$j = (\,i+l\,)\ mod\ q, \qquad (2.4)$$

    where *mod* is operation of obtaining the remainder of the integer division;

– Blocks A'_{ij}, B'_{ij} obtained by each subtask *(i,j)* as a result of block transmission are multiplied and added to block C_{ij}

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

– Blocks B'_{ij} of each subtask *(i,j)* are transmitted to the subtasks, which are upper neighbors in the grid columns (the first row blocks are transmitted to the last row of the grid).

To illustrate these rules the state of blocks in each subtask in the course of executing iterations of the computation stage is given in Figure 2.8 (for the grid of 2×2).

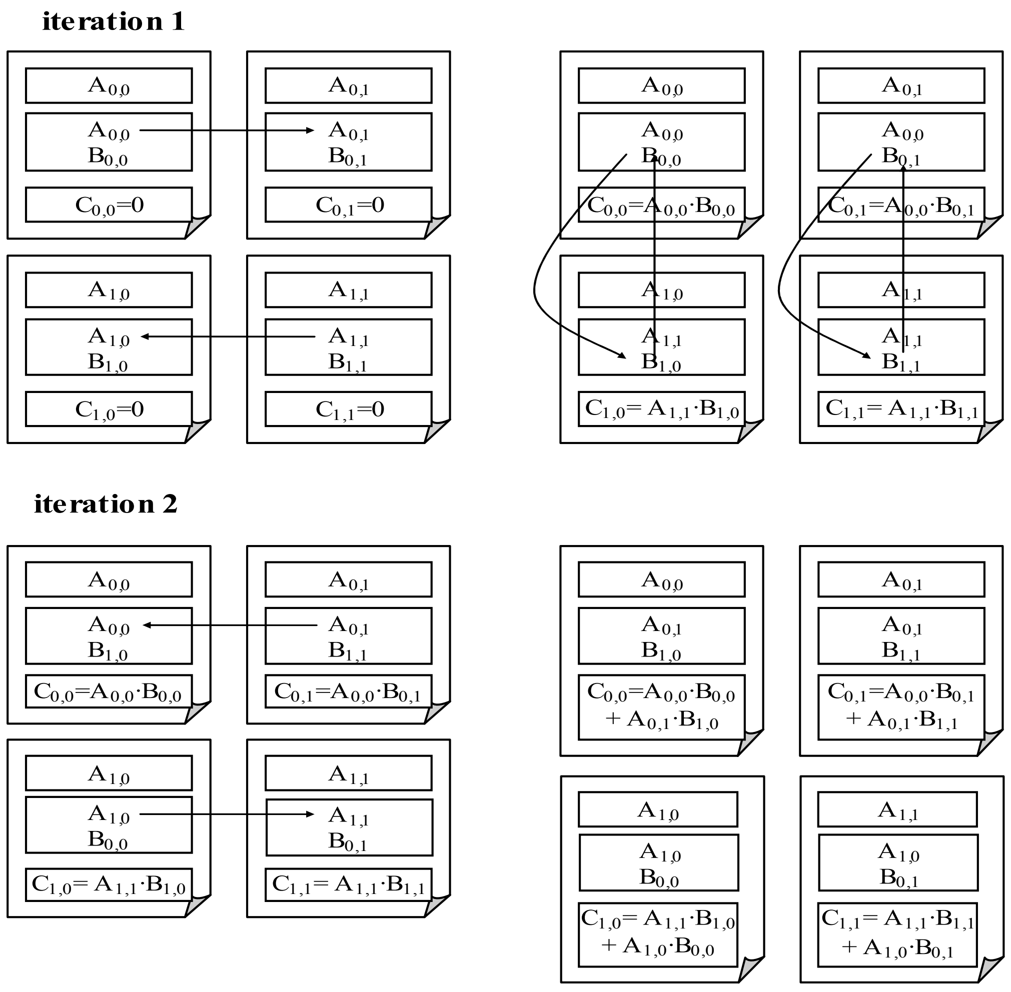


Figure. 2.8. The State of Blocks in Each Subtask in the Course of Execution of the Fox Algorithm Iterations

Scaling and Distributing the Subtasks among the Processors

The number of blocks in this scheme of parallel computations may vary depending on the choice of their sizes. These sizes may be chosen so that the total number of the basic subtasks coincide with the number of processors *p*. Thus, for instance, in the simplest case when the number of processors may be presented as $p=\delta^2$ (i.e. it is a perfect square), the number of blocks in the matrices vertically and horizontally may be chosen equal to δ (i.e. $q=\delta$). This way to define the number of blocks makes the amount of computations in each subtask the same and, thus, perfect balancing of the computational load is achieved. In a more general case, when the number of processors and the sizes of matrices are arbitrary, computational load may not be absolutely equal. Nevertheless, if the choice of parameters is adequate, the computational load may be distributed among the processors equally with adequate accuracy.

To execute the Fox algorithm efficiently the set of available processors should be arranged as a square grid, as basic subtasks in the Fox algorithm are presented as a square grid and blocks are transmitted to rows and columns of the subtask grid in the course of computations. In this case it is possible to immediately map the set of subtasks onto the set of processors locating the basic subtasks (*i,j*) on the processor $P_{i,j}$. The adequate structure of the data transmission network may be provided at the physical level, if the network topology is a grid or a complete graph.

Exercise 4 – Code the Parallel Matrix Multiplication Program

To do this Exercise you should develop the parallel program for matrix multiplication based on the Fox algorithm. This Exercise is aimed at:

- Enhancing the practical knowledge on the development of the parallel matrix computations based on chessboard block data distribution,
- Getting experience in developing more complicated parallel programs
- Getting familiar with the use of communicators and virtual topologies in MPI.

Task 1 – Open the Project ParallelMatrixMult

Open the project **ParallelMatrixMult** using the following steps:

- Start the application **Microsoft Visual Studio 2005**, if it has not been started yet,
- Execute the command **Open→Project/Solution** in the menu **File,**
- Choose the folder **c:\MsLabs\ParallelMatrixMult** in the dialog window **Open Project;**
- Make the double click on the file **ParallelMatrixMult.sln** or select it and execute the command **Open.**

After the project has been open in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *ParallelMM.cpp*, as it is shown in Figure 2.9. After that, the code, which has to be modified, will be opened in the workspace of the Visual Studio.

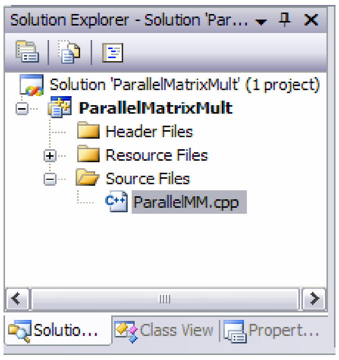


Figure. 2.9. Opening the File ParallelMM.cpp with the Use of the Solution Explorer

The main function of the parallel program to be developed is located in the file *ParallelMV.cpp* and provides access to the necessary libraries and contains the declarations of the necessary variables, the calls of the initialization function and the function terminating the execution environment of MPI program, the function determining the number of available processes and process ranks:

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

void main(int argc, char* argv[]) {
  double* pAMatrix;  // First argument of matrix multiplication
  double* pBMatrix;  // Second argument of matrix multiplication
  double* pCMatrix;  // 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();
}
```

It should be noted that the variables *ProcNum* and *ProcRank* have been declared global as in case of the development of parallel matrix-vector multiplication program (see Lab 1).

The following functions are copied from the the serial matrix multiplication program**:** *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *PrintMatrix* (the purposes of the functions are considered in detail in Exercise 2 of Lab). These functions may be also used in the parallel program. Besides, the preliminary versions for the functions of the computation initialization (*ProcessInitialization*) and process termination (*ProcessTermination*) are also located there.

Compile and run the applications using the s Visual Studio. Make sure that the inital message "`Parallel matrix multiplication program`" is output into the command console.

Task 2 –Create the Virtual Cartesian Topology

According to the parallel computation scheme described in Exercise 3, it is necessary to arrange the available MPI program processes as a virtual topology in the form of a two-dimensional square grid in order to carry out the Fox algorithm efficiently. It is only possible that the number of the available processes is a perfect square.

Before we start the execution of the parallel algorithm let us check if the number of the available processes is a perfect square, i.e that *ProcNum = GridSize×GridSize*. If it is not so, we will output the diagnostic message. We will continue the execution of the application only if this condition is met.

Let us call the value *GridSize* the size of the grid. This value will be used in data distribution and data collection. It will be also used in the execution of the Fox algorithm iterations. Let us declare the corresponding global variable and set its value.

```
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();
}
```

Let us develop the function *CreateGridCommunicators*, which will create a communicator as a two-dinemsional square grid, determine the coordinates of each process in the grid and create communicators for each row and each column separately.

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

The following function is intended in MPI for creating the Cartesian topology:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims,
  int *periods, int reorder, MPI_Comm *cartcomm),
where:
- oldcomm  - the initial communicator,
- ndims    - the dimension of the Cartesian grid,
- dims     - the array of the length ndims, which sets the number of
             processes  in each grid dimension,
- periods  - the array of the length ndims, which determines whether the
             grid is periodical along each dimension,
- reorder  - the allowability of changes of the process enumeration,
- cartcomm – the communicator being created with the Cartesian process
             topology.
```

So in order to create the Cartesian topology it is necessary to determine two arrays: the first one *DimSize* determines the size of each grid dimension, while the second one *Periodic* determines whether the grid is periodic along each dimension. As we are to create a two-dimensional square grid, both *DimSize* elements must be determined in the following way: $DimSize[0] = DimSize[1] = \sqrt{ProcNum}$. According to the scheme of the parallel computations (Exercise 3), we will have to perform a cyclic shift along the processor grid columns. Therefore, the second dimension of the Cartesian topology has to be periodic. As a result of the execution of the function *MPI_Cart_create* the new communicator will be stored in the variable *cartcomm*. Thus, it is necessary to declare a variable for storing the new communicator and give it to the function *MPI_Cart_create* as an argument. As the grid communicator is widely used in all functions of the parallel application, let us declare the corresponding variable as a global one. All the communicators in the MPI library are of *MPI_Comm* type.

Let us add the call of the function for creating the grid to the function *CreateGridCommunicators*:

```
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
<…>
// Function for creating the 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;

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

In order to determine the Cartesian coordinates of the process according to its rank, we may use the following function:

```
int MPI_Card_coords(MPI_Comm comm,int rank,int ndims,int *coords),
where:
- comm   – the communicator with grid topology,
- rank   - the rank of the process, for which  the Cartesian coordinates
           are determined,
- ndims  - the dimension of the gird,
- coords – the Cartesian process coordinates returned by the function.
```

As we have created a two-dimensional grid, each process in the grid has two coordinates, which correspond to the row number and the column number. The process is located at the intersection of these row and column. Let us declare the global variable, i.e. the array for storing the coordinates of each process, and define the coordinates by means of the function *MPI_Cart_coords*:

```
int GridSize;         // Size of virtual processor grid
MPI_Comm GridComm;    // Grid communicator
int GridCoords[2];    // Coordinates of current processor in grid
<…>
// Function for creating the 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);
}
```

Let us create communicators for each process grid row and column. For this purpose the MPI function, which make possible to divide the grid into subgrids, can be used (more detailed information of the use of the function *MPI_Cart_sub* is given in Section 4 “Parallel programming with MPI” of the training material):

```
int MPI_Card_sub(MPI_Comm comm, int *subdims, MPI_Comm *newcomm),
where:
- comm    - the initial communicator with a grid topology,
- subdims – the array for defining, which dimensions should remain in the
            subgrid to be created,
- newcomm – the communicator with the subgrid, which is being created.
```

Let us declare the communicators for the row and the column as global variables and divide the already created communicator *GridComm* as follows:

```
MPI_Comm GridComm;    // Grid communicator
MPI_Comm ColComm;     // Column communicator
MPI_Comm RowComm;     // Row communicator
<…>
// Function for creating the 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);
}
```

Let us call the function *CreateGridCommunicators* from the main function of the parallel application:

```
void main(int argc, char* argv[]) {
  double* pAMatrix;  // First argument of matrix multiplication
  double* pBMatrix;  // Second argument of matrix multiplication
  double* pCMatrix;  // 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();
}
```

Compile the application. If you find errors, correct them, comparing your code to the code given in the manual. Run the application several times changing the number of the available processes. Make sure that if the available number of processes is not a perfect square, the diagnostic message is output and the application terminates its operation.

Task 3 – Input the Initial Data

At the following stage of the parallel application development, it is necessary to set the matrix sizes and to allocate memory for storing the initial matrices and their blocks. According to the parallel computation scheme four matrix blocks are located on each process at any given moment of time: two blocks of the matrix *A*, a block of the matrix *B* and a block of the result matrix *C* (see Exercise 3). Let us define the variables for storing the matrix blocks and the sizes of the blocks:

```
void main(int argc, char* argv[]) {
  double* pAMatrix;       // First argument of matrix multiplication
  double* pBMatrix;       // Second argument of matrix multiplication
  double* pCMatrix;       // Result matrix
  int Size;               // Size of matrices
  int BlockSize;          // Sizes of matrix blocks
  double *pMatrixAblock; // Initial block of matrix A
  double *pAblock;        // Current block of matrix A
  double *pBblock;        // Current block of matrix B
  double *pCblock;        // Block of result matrix C

  double Start, Finish, Duration;
```

Let us develop the function *ProcessInitialization* in order to input the matrix sizes, set the matrix block sizes, allocate memory for storing them and initialize the matrix elements.

```
// 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 );
```

Let us start with inputting the sizes. For simplicity, we will assume, as previously, that all the matrices involved in multiplication, are square of the order *Size*×*Size*. The size *Size* must provide the matrix distribution among the processes in equal square blocks, i.e. the size *Size* must be divisible by the processor grid size *GridSize*.

In order to input the size we will implement the dialogue with the user as it was done in case of Lab 1. If the user enters an incorrect number he is suggested repeating the input. The dialogue is carried out only on the root process. It should be noted that the root process is usually the process, which has the rank 0 in the communicator *MPI_COMM_WORLD*. When the matrix size is entered correctly, the value of the variable *Size* is broadcast to all the processes:

```
// 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 the size of the matrices: ");
      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);
}
```

After all matrix sizes are set, it is possible to determine the size of matrix blocks and to allocate memory for storing the initial matrices, the result matrix, the matrix blocks (the initial matrices are available only on the root process):

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

In order to determine the elements of the initial matrices we will use the function *DummyDataInitialization*, which was developed in the course of the realization of the serial program of matrix multiplication:

```
// 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];
    DummyDataInitialization(pAMatrix, pBMatrix, Size);
  }
}
```

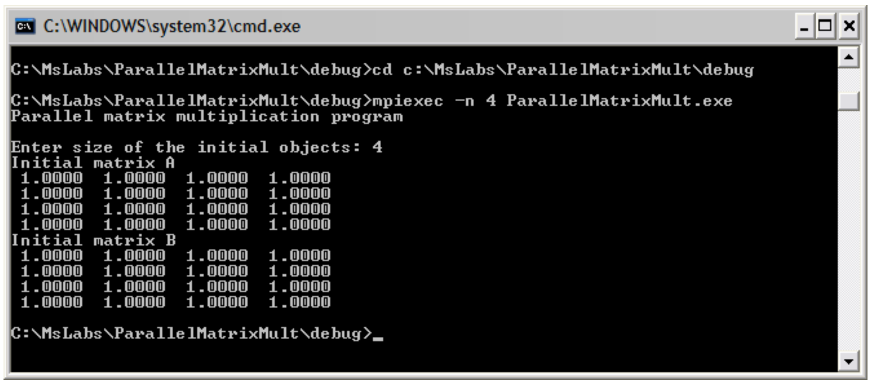
The block of the result matrix *pCblock* serves for summing the products of the block multiplication. In order to store the sums correctly it is necessary to set all its elements of this block to zero initially:

```
// 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;
  }
}
```

Let us call the function *ProcessInitialization* from the main function of the parallel application. In order to control the correctness of the initial data input, we will use the function of the formatted matrix output *PrintMatrix*: let us print out the initial matrices *A* and *B* on the root process.

```
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();
}
```

Compile and run the application. Make sure that the dialogue for the input of matrix sizes makes possible to enter only the correct matrix size value. Analyze the value of the initial matrix elements. If the data is set correctly, all the initial matrix elements must be equal to 1 (see Figure 2.10).



```
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>_
```

Figure. 2.10. Initial Data Setting

Task 4 – Terminate the Parallel Program

In order to terminate the application at each stage of development, we should develop the function of correct termination. For this purpose we should deallocate the memory, which has been allocated dynamically in the course of the program execution. Let us develop the corresponding function *ProcessTermination*. The memory for storing the initial matrices *pAMatrix* and *pBMatrix* and for storing the result *matrix pCMatrix*, was allocated on the root process; besides, memory was allocated on all the processes for storing the four matrix blocks *pMatrixAblock*, *pAblock*, *pBblock*, *pCblock*. All these objects must be given to the function *ProcessTermination* as arguments:

```
// 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;
}
```

The call of the process termination function must be executed immediately before the call of the termination of the parallel program:

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

Compile and run the application. Make sure that it operates correctly.

Task 5 – Distribute the Data among the Processes

In accordance with the parallel computation scheme, the initial matrices to be multiplied are located on the root process. The root process, i.e. the process with rank 0, is located in the upper left hand corner of the processor grid.

It is necessary to distribute the matrices blockwise among the processes so that the blocks A_{ij} and B_{ij} have to be located on the process placed at the intersection of the *i*-th row and *j*-th column of the processor grid. The matrices and the matrix blocks are stored in one-dimensional arrays rowwise. The matrix block is not stored by a continuous sequence of elements in the matrix storage array. Thus, it is impossible to perform blockwise distribution using the standard data types from the library MPI.

To arrange the transmission of blocks within the same communication operation, it is possible to form a derived data type by means of MPI. This approach is intended to be as an assignment for homework. In this lab let us use the following two-stage scheme of data distribution. At the first stage the matrix is divided into horizontal stripes. Each of the stripes contains *BlockSize* rows. These rows are distributed among the processes, which compose the left column of the processor grid (Figure 2.11).

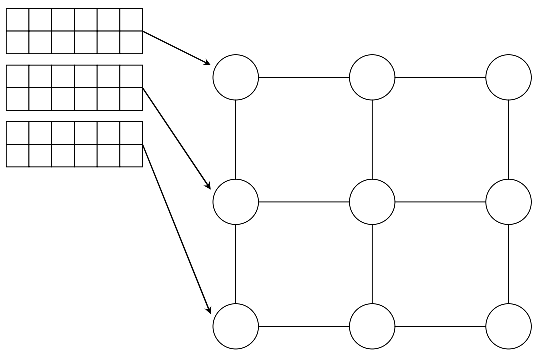


Figure. 2.11. The First Stage of Data Distribution

Each stripe is further divided into blocks among the processes, which compose the processor grid rows. It should be noted that the distribution of the stripes into blocks will be executed sequentially by means of distributing the rows of the stripe with the use of the function *MPI_Scatter* (Figure 2.12).

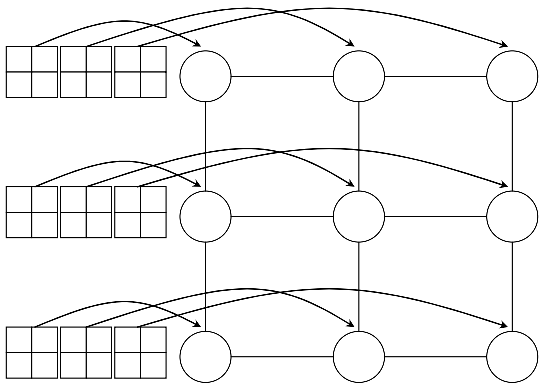


Figure. 2.12. The Second Stage of Data Distribution

In order to distribute the matrix among the processor of grid processes blockwise, we will realize the function *CheckerboardMatrixScatter*.

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

This function has the matrix *pMatrix*, which is stored on the root process, as arguments. It also gets the array *pMatrixBlock* for storing the matrix block on each of the parallel application processes, the matrix size *Size* and the matrix block size *BlockSize* as arguments.

At the first stage it is necessary to divide the matrix among the processes, which compose the left column of the process grid, in horizontal stripes. For this purpose we will use the function *MPI_Scatter* in the communicator *ColComm*. It should be noted that we have already created *GridSize* communicators *ColComm* in the parallel application. In order to determine the communicator, which corresponds to the left column of the processor grid we will use the values recorded in the *GridCoords*. The function *MPI_Scatter* will be called only in the processes, which have the value of the second coordinate equal to zero (i.e. the process is located in the left column).

```
// 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);
  }
}
```

It should be noted that for temporary storage of the horizontal matrix stripe we will use the array *pMatrixRow*.

At the second stage it is necessary to distribute each row of the horizontal matrix stripe along the rows of the processor grid. Let us again use the function *MPI_Scatter* in the communicator *RowComm*. After the execution of these operations, we will deallocate the previously allocated memory:

```
// 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);
  }

  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;
}
```

In order to execute the Fox algorithm it is necessary to distribute blockwise the matrix *A* (the matrix blocks are stored in the variable *pMatrixABlock*) and the matrix *B* (the matrix blocks are stored in the variable *pBblock*). Let us develop the function *DataDistribution*, which provides the distribution of these matrices:

```
// 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);
}
```

Let us call the function of data distribution from the main parallel application function.

```
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();
}
```

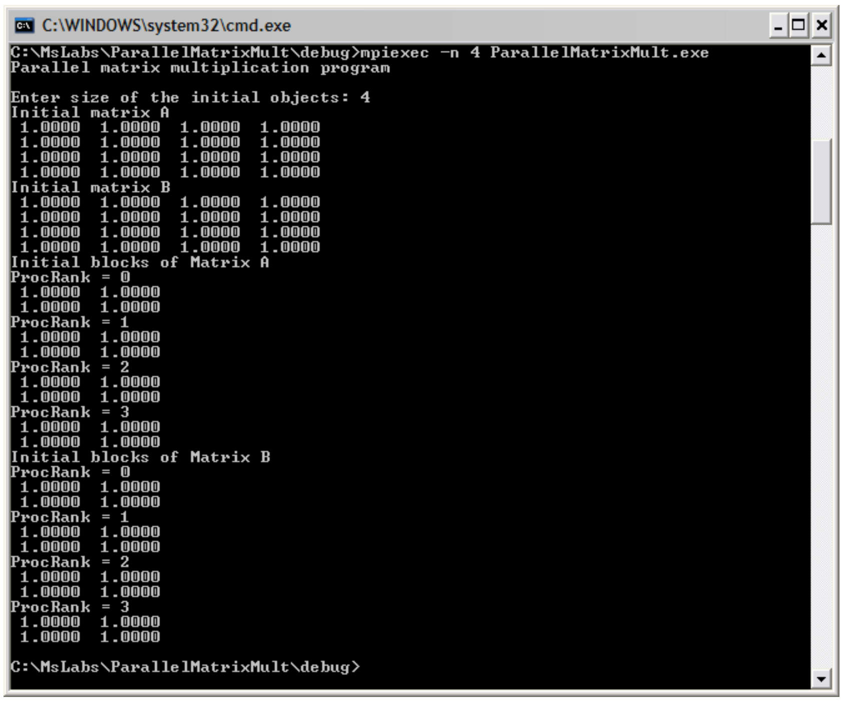
In order to control the correctness of initial data distribution we will make use of debugging print. Let us develop the function, which will print sequentially the matrix blocks on all the processes. Let us call the function *TestBlocks*.

```
// 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);
  }
}
```

Let us call the function of printing the distributed data from the function *main* :

```
    // 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");
  }
```

Compile the application. If you find errors in the process of compiling, correct them, comparing your code to the code given in the manual. Run the application. Make sure that the data is distributed correctly (Figure 2.13):


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
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>
```

Figure. 2.13. The Initial Data Distribution in Case if the Application is Run Using Four Processes and the Matrix Size is Equal to Four

Change the initial data setting. In order to define the initial matrix elements, use the function *RandomDataInitialization* instead of the function *DummyDataInitialization.* Compile and run the application. Make sure that the matrices are distributed among the processes correctly.

Task 6 – Code the Parallel Matrix Multiplication Program

The function *ParallelResultCalculation* executes the parallel Fox algorithm of matrix multiplication. The matrix blocks and their sizes must be given to the function as its arguments:

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

According to the scheme of parallel computations described in Exercise 3, it is necessary to carry out *GridSize* iterations in order to execute matrix multiplication with the use of Fox algorithm. Each of the iterations consists of the execution of the following operations:

- The broadcast of the matrix *A* block along the processor grid row (to execute the step we should develop the function *ABlockCommunication*),
- The multiplication of matrix blocks (to carry out the multiplication of matrix blocks we may use the function *SerialResultCalculation*, which was implemented in the course of the development of the serial matrix multiplication program),
- The cyclic shift of the matrix *B* blocks along the column of the processor grid (the function *BBlockCommunication*).

Thus, the code executing the Fox algorithm of matrix multiplication is the following:

```
// Function for parallel 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 );
  }
}
```

Let us consider these operations in detail in the following exercises of the lab.

Task 7 – Broadcast the Blocks of the Matrix *A*

So at the beginning of each algorithm iteration *iter* a process is selected for each processor grid row, which will send its block of the matrix *A* to the processes of the corresponding grid row. The number of the process *Pivot* in the row is determined according to the following expression:

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

where *i* is the number of the processor grid row, for which we determine the number of the broadcasting process (the number of the row for each process can be determined by the first value in the array *GridCoords*) and *mod* is the operation of calculating the remainder of the division. Thus, the process, which has the value of the second coordinate *GridCoords* coinciding with *Pivot* is the sending process at each iteration. After the number of the sending process has been determined, it is necessary to broadcast the block of the matrix *A* along the row. Let us do it with the use of the function *MPI_Bcast* in the communicator *RowComm*. Here we will need an additional block of matrix *A*: the first block *pMatrixAblock* stores the block, which was located on this process before the beginning of the computations, the block *pAblock* stores the matrix block, which participates in multiplication at this algorithm iteration. Before broadcasting the block *pMatrixAblock* is copied to the array *pAblock*, and then the array *pAblock* is broadcast to the row processes.

```
// Broadcasting blocks of the matrix A 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);
}
```

Let us control the correctness of this stage execution. For this purpose let us add the call of the function *ParallelResultCalculation* to the main function of the parallel application. To provide the program compilation transform calls of the functions of matrix block multiplication (*SerialResultCalculation*) and the cyclic shift of matrix block *B* (*BblockCommunication*), which have not been realized yet into comment (in the function *ParallelResultCalculation*).

It should be noted that at this moment the function *RandomDataInitialization* is being used to generate the initial matrix values (we use this setting to check the correctness of the data distribution stage execution). After broadcasting the blocks of the matrix *A*, we will print out the values stored in the blocks *pAblock* on all the processors:

```
// Function for parallel 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 );
  }
}
```

Compile and run the application using 9 processors. Check the correctness broadcasting the blocks of the matrix *A*. For this purpose compare the blocks located on the processes at each iteration of the Fox algorithm to the output executed after the accomplishment of the function *DataDistribution*. The number of the block, which is located on all the processors of the row *i* must be calculated according to formula (2.4).

Task 8 – Cyclic Shift the Blocks of the Matrix B along the Processor Grid Columns

In the course of matrix block multiplication it is necessary to perform the cyclic shift of blocks of the matrix *B* along the processor grid columns (Figure 2.14).

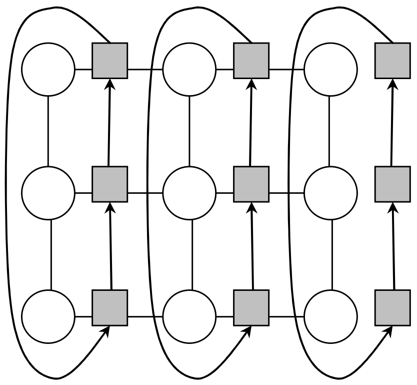


Figure. 2.14. The Cyclic Shift of Blocks of the Matrix *B* along the Processor Grid

This shift may be performed in several ways. The most evident approach is to arrange the sequential sending and receiving the matrix blocks by means of the functions *MPI_Send* and *MPI_Receive* (detailed information about the functions may be obtained in Section 4 of the training material). The problem is that it is hard to perform this sequence so as to avoid deadlock situations, i.e. for instance, avoid the situations when all the processes start blocking receiving operations at the same time.

The efficient and guaranteed simultaneous execution of sending and receiving operations may be achieved by means of the function *MPI_Sendreceive*:

```
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),
where
- sbuf, scount, stype, dest,   stag – the parameters of the sent message,
- rbuf, rcount, rtype, source, rtag – the parameters of the received message,
- comm   - the communicator within which the data transmission is performed,
- status – the structure of the data, which contains the control information
           of the function execution result.
```

As you can see from the description of the function *MPI_Sendrecv* sends the messages described by the parameters (*sbuf, scount, stype, dest, stag*) to the process with the rank *dest* and receives a message to the buffer defined by the parameters (*rbuf, rcount, rtype, source, rtag*), from the process with the rank *source*.

There are different buffers for sending and receiving messages in the function *MPI_Sendrecv.* In case if the messages are of the same type, MPI provides an opportunity to use the single buffer:

```
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);
```

Let us use this function to perform the cyclic shift of the blocks of the matrix. Each process sends a message to the previous process of the same processor grid column and receives a message from the next process. The processes located in the first row of the processor grid sends its block to the process located in the last row (the row with the number *GridSize-1*).

```
// Function for cyclic shifting the blocks of the matrix B
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);
}
```

Let us test the correctness of this stage execution. Let us restore the call of the function for cyclic shifting of the blocks of the matrix *B* (*BblockCommunication*) in the function *ParallelResultCalculation* (delete the comment signs in the calling line). Also let us eliminate the printing ofthe blocks of the matrix *A*. After the execution of shifting the blocks of the matrix *B*, let us add printing out the blocks *pBblock* on all the processors:

```
// Function for parallel 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”);

  }
}
```

Compile and run the application using 9 processes. Check the correctness of shifting blocks of the the matrix *B* (blocks must be shifted by 1 upward along the processor grid column at each iteration). For this purpose compare the blocks located on processes at each iteration of the Fox algorithm with the blocks located on processes after the execution of the function *DataDistribution*.

Task 9 – Implement the Matrix Block Multiplication

After we have sent the matrix *A* blocks, it is necessary to execute the multiplication of the block *pAblock* by the block *pBblock*. Then we should add the product to the block *pCblock*. In order to multiply the matrix blocks on each of the process, it is necessary to execute the serial matrix multiplication algorithm for the blocks *pAblock* and *pBblock* of size *BlockSize*×*BlockSize*. For this purpose we can use the function *SerialResultCalculation*, which was developed in the course of the implementation of the serial matrix multiplication algorithm (Exercise 1).

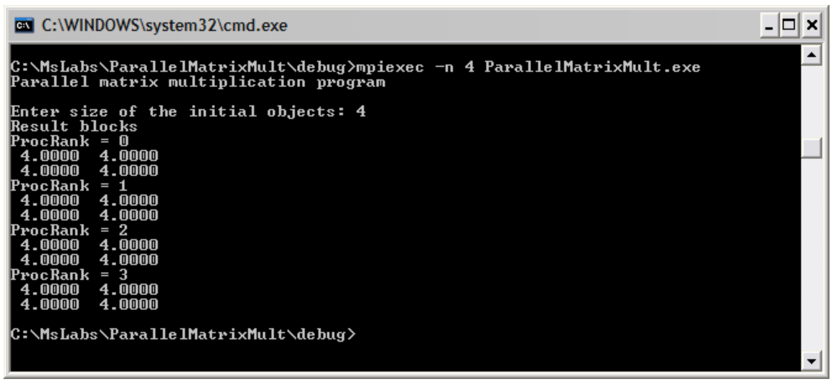
```
// Function for block multiplication
void BlockMultiplication(double* pAblock, double* pBblock,
  double* pCblock, int Size) {
  SerialResultCalculation(pAblock, pBblock, pCblock, Size);
}
```

After the execution of *GridSize* iterations of the Fox algorithm, the block of the result matrix is located on each process. To check the correctness of the algorithm execution before gathering data, we will print out the obtained blocks of the result matrix *C* by means of the function *TestBlocks* (let us delete the debugging print of the results of the initial block broadcast and the debugging print of algorithm iteration execution from the parallel application code):

```
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);

  // 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();
}
```

In order to determine the initial matrices elements in the function *ProcessInitialization* we will again use the function *DummyDataInitialization*. The blocks of the result matrix, located on all the processes, must consist of the elements, which are equal to the value *Size* of the initial matrices (Figure 2.15).

Compile and run the application. Set different sizes of the initial matrices. Make sure that the results of the Fox algorithm execution are correct.



```
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>
```

Figure. 2.15. The Blocks of the Result Matrix Computed Using the Fox Algorithm

Task 10 – Gather the Results

The procedure of gathering the results repeats the procedure of initial data distribution. The difference consists in the fact that all the stages must be executed in the reverse order. First, it is necessary to gather the blocks located on the processes of one process grid row into stripes of the result matrix. Then it is necessary to gather the stripes located on the left process grid column into the matrix.

In order to gather the result matrix we will use the function *MPI_Gather* of the MPI library. This function gathers the data from all the processes in the communicator onto one process. The function operations are opposite to the operations of the function *MPI_Scatter*. The function *MPI_Gather* has the following heading:

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype stype,
  void *rbuf,int rcount,MPI_Datatype rtype, int root, MPI_Comm comm),
where
- sbuf, scount, stype – the parameters of the message being sent,
- rbuf, rcount, rtype – the parameters of the message to be recieved,
- root – the rank of the process, which executes the data gather,
- comm – the communicator, in which the data transfer is executed.
```

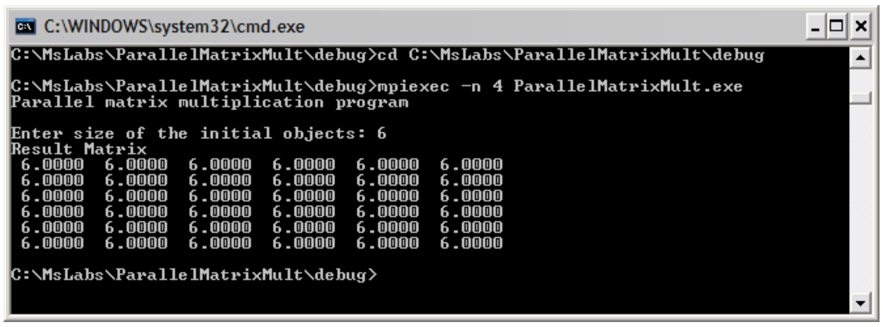
We will implement the procedure of gathering the result matrix *C* in the function *ResultCollection*:

```
// 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;
}
```

Let us add the call of the function *ResultCollection* instead of the call of the function of testing the partial results by means of the debugging print (*TestBlocks*). In order to control the correctness of data gather and the program execution in general we will print the result matrix *pCMatrix* on the root process using the function *PrintMatrix*.

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

  // Gathering the result matrix
  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();
}
```

Compile and run the application. Test the correctness of the program execution. It should be noted that if the initial data is generated by means of the function *DummyDataInitialization*, all the elements of the result matrix should be equal to the value *Size* (Figure 2.16).



```
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>
```

Figure. 2.16. The Test Result of the Matrix Multiplication

Task 11 – Test the Parallel Program Correctness

After we have executed the function of gathering, it is necessary to check the correctness of the program execution. For this purpose we will develop the function *TestResult*, which will compare the results of the serial and parallel algorithms. In order to execute the serial algorithm it is possible to use the function *SerialResultCalculation*, which was developed in Exercise 2. The result of this function will be stored in the matrix *pSerialResult*. Then we will compare element by element this matrix to the matrix *pCMatrix*, which was obtained by means of the parallel program. In order to obtain each element of the result matrix, it is necessary to execute serial multiplication and summation of real numbers. The order of executing these operations can influence the machine inaccuracy of computations and its value. That is why it is impossible to check of the matrix elements are identical or not. Let us introduce the allowed divergence value of the serial and parallel algorithm results – the value *Accuracy*. The matrices are assumed to be the same if the corresponding elements differ by no more than the value of the allowed error *Accuracy*.

The function *TestResult* must have the access to the initial matrices *pAMatrix*, *pBMatrix* and *pCMatrix*, and therefore, can be executed only on the root process:

```
// Function for testing the matrix multiplication result
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;
  }
}
```

The results of the function execution are the print of the diagnostic message. It is possible to check the results of the parallel program execution using this message regardless of the initial object size in case of any values of the initial data.

Transform into comment the calls of the functions, using the debugging print. Those function calls have been previously used for checking the correctness of parallel program execution. Instead of the function *DummyDataInitialization*, which generates matrices of simple type, call the function *RandomDataInitialization*, which generates the matrix by means of the random data generator. Compile and run the application. Set various amounts of the initial data. Make sure that the application is functioning properly.

Task 12 – Carry out Computational Experiments

Let us determine the parallel algorithm execution time. For this purpose add clocking to the program code. As the parallel program includes the stage of data distribution, the computation of the result block on each process and result gather, the timing should start immediately before the call of the function *DataDistribution* and stop right after the execution of the function *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();
```

It is obvious that this way we will print the time spent on the execution of the calculations done by the root process (the process with the rank 0). The execution time for other processes may slightly differ from it. At the stage of developing the parallel algorithm we paid special attention to the equal load (balancing) of the processes. Therefore, now we have good reason to believe that the algorithm execution time for the other processes differs from that of the root process insignificantly.

Add the marked code fragment to the body of the main application function. Compile and run the application. Fill out the table:

Table 2.3. Execution Time of the Fox Algorithm of Matrix Multiplication and the Obtained Speed Up

Test Number	Matrix Size	Serial Algorithm	Parallel Algorithm			
			4 processors		9 processors	
			Time	Speed up	Time	Speed up
1	10					
2	100					
3	500					

4	1,000					
5	1,500					
6	2,000					
7	2,500					
8	3,000					

Give the serial algorithm execution time in the column "Serial algorithm". The time must be measured in the course of testing the serial application in Exercise 2. In order to calculate the speed up, divide the serial algorithm execution time by the parallel algorithm execution time. Place the results in the corresponding column of the table.

In order to estimate the execution time of the parallel algorithm implemented according to the computational scheme, which was given in Exercise 3, you may use the following expression:

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

(the detailed derivation of the formula is given in Section 8 "Parallel Algorithms of Matrix Multiplication" of the training material). Here n is the matrix size, p is the number of processes, q is the size of the processor grid, τ is the execution time for a basic computational operation (this value has been computed in the course of testing the serial algorithm), α is the latency, and β is the bandwidth of the data transmission network. The values obtained in the course of carrying out the Compute Cluster Server Lab 2 "Carrying out Jobs under Microsoft Compute Cluster Server 2003" should be used as the latency and the bandwidth.

Calculate the theoretical execution time for the parallel algorithm according to formula (2.5). Tabulate the results in the following way (Table 2.4):

Table 2.4. The Comparison of the Parallel Experiment Execution Time to the Theoretically Calculated Execution Time

Test Number	Matrix Sizes	4 processors		9 processors	
		Model	Experiment	Model	Experiment
1	10				
2	100				
3	500				
4	1,000				
5	1,500				
6	2,000				
7	2,500				
8	3,000				

Discussions

- How great is the difference between the execution time of the serial and the parallel algorithms? Why?
- Has there any speed up been obtained in case when the matrix size was 10 x 10? Why?
- Are the theoretical and the experiment execution time values congruent? What may be the cause of incongruity?

Exercises

1. Modify the developed Fox algorithm implementation using the derived MPI data type for broadcasting and gathering matrix blocks (see Section 4 "Parallel programming with MPI").
2. Study the parallel algorithm of matrix multiplication based on block striped matrix partitioning. Develop a program implementation of this algorithm.
3. Study the Cannon parallel algorithm of matrix multiplication based on chessboard block matrix partitioning. Develop a program implementation of this algorithm.

Appendix 1. The Program Code of the Serial Application for Matrix Multiplication

```
#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 the 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;  // First argument of matrix multiplication
  double* pBMatrix;  // Second argument of matrix multiplication
  double* pCMatrix;  // 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);
}
```

Appendix 2. The Program Code of Parallel Application for Matrix Multiplication

```
#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);
}

// Function for creating the 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 the size of matrices: ");
      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);

  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 blocks of the matrix A 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);
}

// Function for cyclic shifting the blocks of the matrix B
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);
}
```

```
// Function for parallel 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);
  }
}

// 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);
  }
}

// Function for testing the matrix multiplication result
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;  // First argument of matrix multiplication
  double* pBMatrix;  // Second argument of matrix multiplication
  double* pCMatrix;  // Result matrix
  int Size;          // Size of matrices
  int BlockSize;     // Sizes of matrix blocks
  double *pAblock;   // Initial block of matrix A
  double *pBblock;   // Initial block of matrix B
  double *pCblock;   // Block of result matrix C
  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 communcators
    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 the Fox method
    ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
      pCblock, BlockSize);

    // Gathering the result matrix
    ResultCollection(pCMatrix, pCblock, Size, BlockSize);

    TestResult(pAMatrix, pBMatrix, pCMatrix, Size);

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

  MPI_Finalize();
}
```