


Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program among the world's research and education centers

Strategic initiative "Achieving leading positions in the field of supercomputer technology and high-performance computing"

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

Parallel Programming for Multiprocessor Distributed Memory Systems

06 Practice
Parallel Algorithms of Matrix Multiplication

With the support of Microsoft

Sysoyev A.V.
Software department

Contents

- Matrix Multiplication Problem Statement
- Serial Implementation
- Parallel Algorithm
- Parallel Program

MATRIX MULTIPLICATION PROBLEM STATEMENT

Step 1. Problem Statement…

- Matrix multiplication

$$C = A \cdot B$$

- or

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

- 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

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

Step 1. Problem Statement

- The pseudo code for the given matrix multiplication algorithm may be as follows (hereafter we assume that the matrices are square)

```
// 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];
  }
```

- Computational complexity is $O(n^3)$

SERIAL IMPLEMENTATION

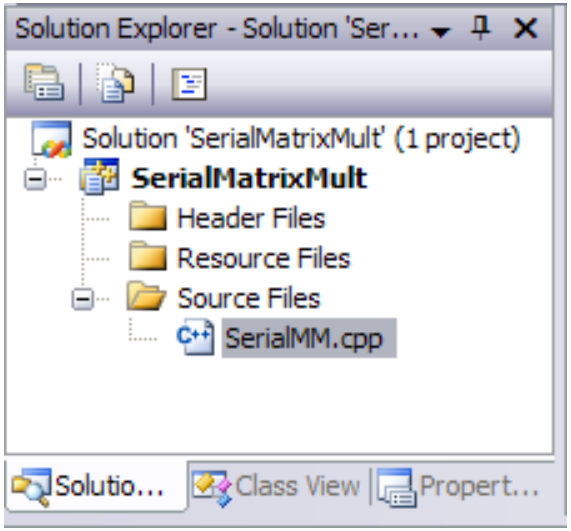
Step 2. Serial Implementation…

- Task 1 – Open the Project SerialMatrixMult
- Task 2 – Input the Matrix Size
- Task 3 – Input the Initial Data
- Task 4 – Terminate the Program Execution
- Task 5 – Implement the Matrix Multiplication
- Task 6 – Carry out the Computational Experiments

Step 2. Serial Implementation…

Task 1 – Open the Project SerialMatrixMult

- Start **Microsoft Visual Studio**
- Open solution SerialMatrixMult.sln from the folder **c:\ParLabs\SerialMatrixMult**
- Open file **SerialMM.cpp** in the window Solution Explorer (Ctrl+Alt+L)



Step 2. Serial Implementation…

Task 1 – Open the Project SerialMatrixMult

- Next variables will be used in the program

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

- The program code, which follows the declarations of the variables, is the output of the initial message and the waiting for pressing any key before the application exit

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

Step 2. Serial Implementation…

Task 2 – Input the Matrix Size

- In order to set the initial data of the matrix multiplication program implement the function **`ProcessInitialization()`**
  - determine the matrix size
  - allocate the memory for the matrices
  - sets the values of the initial matrices elements

```
// Function for process initialization
void ProcessInitialization(double* &pAMatrix,
  double* &pBMatrix, double* &pCMatrix, int &Size);
```

Step 2. Serial Implementation…

Task 2 – Input the Matrix Size

- Determine the sizes of the objects with correct input control

```
// Function for process initialization
void ProcessInitialization(double* &pAMatrix,
  double* &pBMatrix, double* &pCMatrix, int &Size) {
  // Setting the size of matrices
  do {
    printf("\nEnter size of matrices: ");
    scanf("%d", &Size);
    printf("\nChosen matrices' size = %d", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);
}
```

Step 2. Serial Implementation…

Task 2 – Input the Matrix Size

- Add the call of the function **`ProcessInitialization()`** to the **`main()`** function after the initial message line

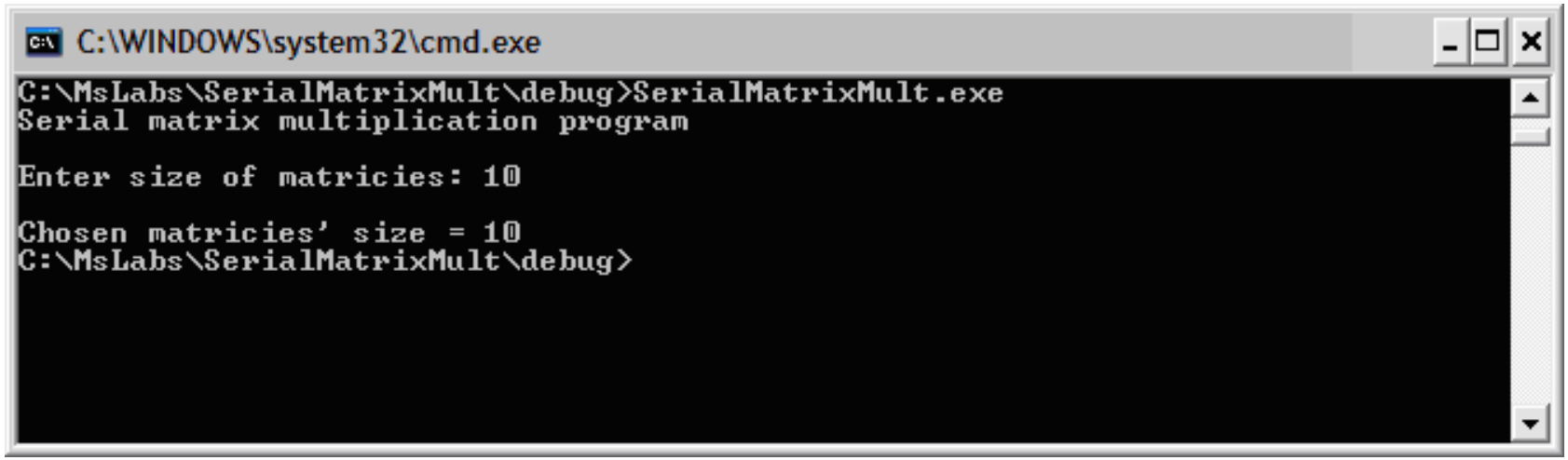
```
void main() {
  double *pAMatrix; // The first argument of multiplication
  double *pBMatrix; // The second argument of multiplication
  double *pCMatrix; // The result matrix
  int Size;         // Size of matrices

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

Step 2. Serial Implementation…

Task 2 – Input the Matrix Size

- Compile and run the application



Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Memory allocation

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

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

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Implement the function to set the matrices elements by the following template

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

- The prototype of the function **`DummyDataInitialization()`**

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

Step 2. Serial Implementation…

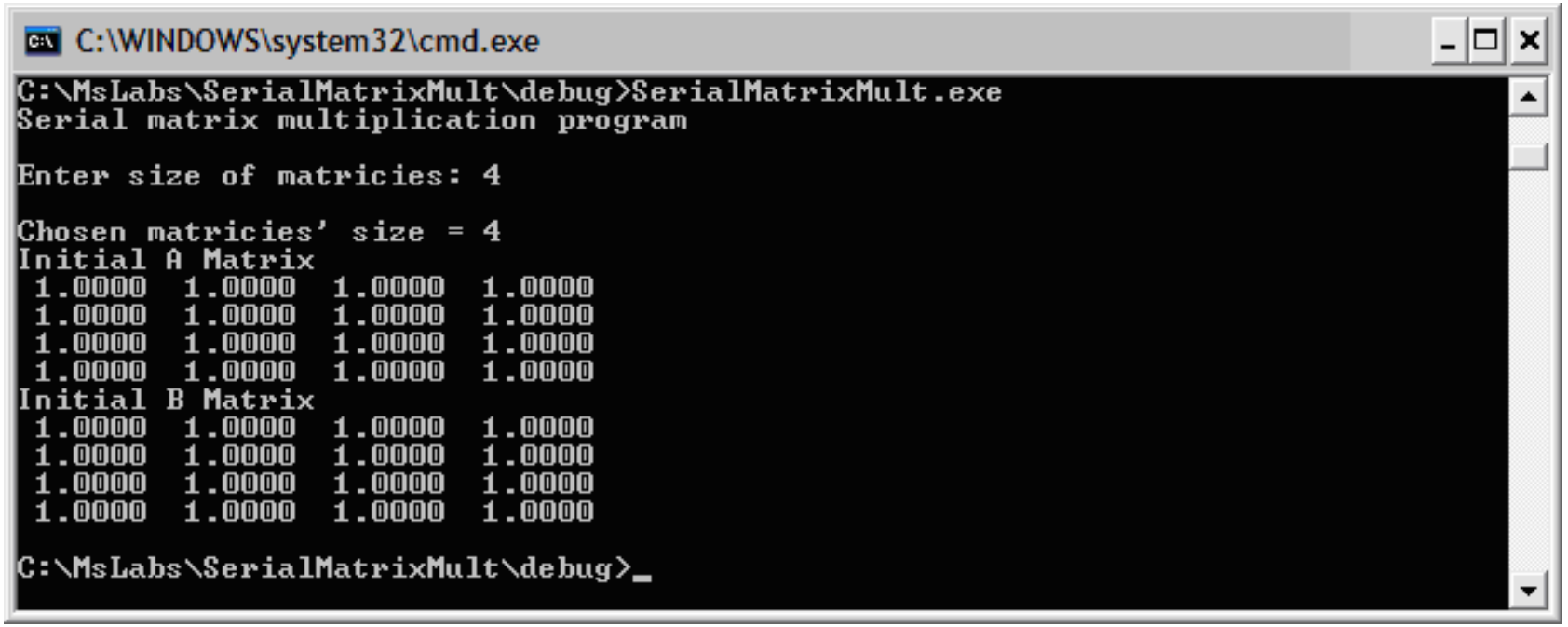
Task 3 – Input the Initial Data

- Call the function **`DummyDataInitialization()`** after allocating memory inside the function **`ProcessInitialization()`**
- Fill the matrix *C* with zeros
- Use of the formatted matrix output function **`PrintMatrix()`**, which was developed in the Practice 05

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Compile and run the application
- Check the correctness of data input



Step 2. Serial Implementation…

Task 4 – Terminate the Program Execution

- The function for correct program termination **`ProcessTermination()`**

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

Step 2. Serial Implementation…

Task 4 – Terminate the Program Execution

- The function **`ProcessTermination()`** should be called at the end of the **`main()`** function

```
// Memory allocation and data initialization
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);
```

Step 2. Serial Implementation…

Task 5 – Implement the Matrix Multiplication

- To multiply the matrices develop the function **`SerialResultCalculation()`**

```
// Function for matrix multiplication
void SerialResultCalculation(double* pAMatrix,
  double* pBMatrix, double* pCMatrix, int Size) {
  int i, j, k;
  for (i = 0; i < Size; i++)
    for (j = 0; j < Size; j++)
      for (k = 0; k < Size; k++)
        pCMatrix[i*Size+j] +=
          pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}
```

Step 2. Serial Implementation…

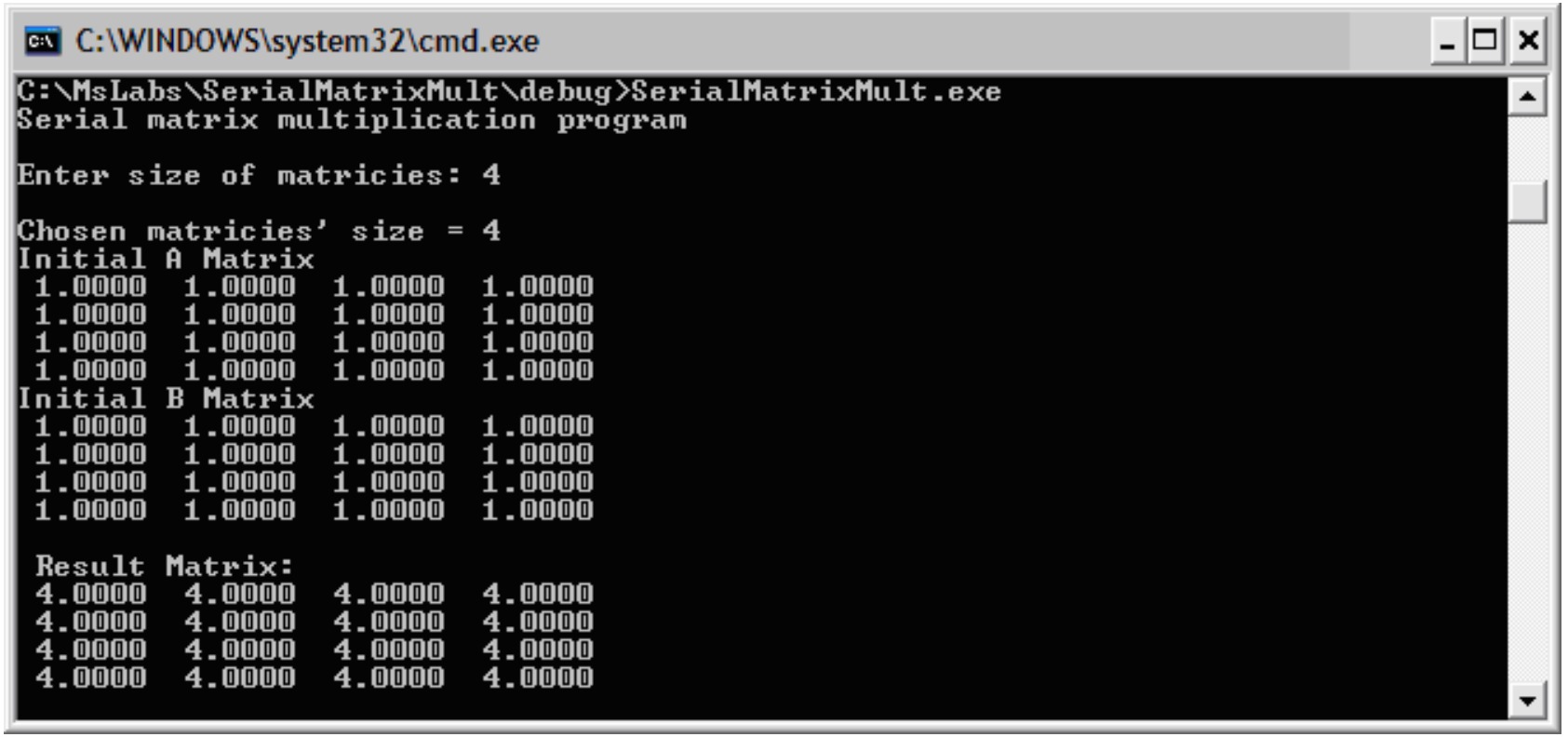
Task 5 – Implement the Matrix Multiplication

- Call the function of matrix multiplication from the **`main()`** function
- In order to control the correctness of the function implementation print out the result matrix
- If the implementation is correct the values of the elements of the result matrix must be equal to the order of the matrix

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

Step 2. Serial Implementation…

Task 5 – Implement the Matrix Multiplication



Step 2. Serial Implementation…

Task 6 – Carry out the Computational Experiments

- Develop the function **`RandomDataInitialization()`** for setting the data with random values (initialize the random generator by the current time value)

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

Step 2. Serial Implementation

Task 6 – Carry out the Computational Experiments

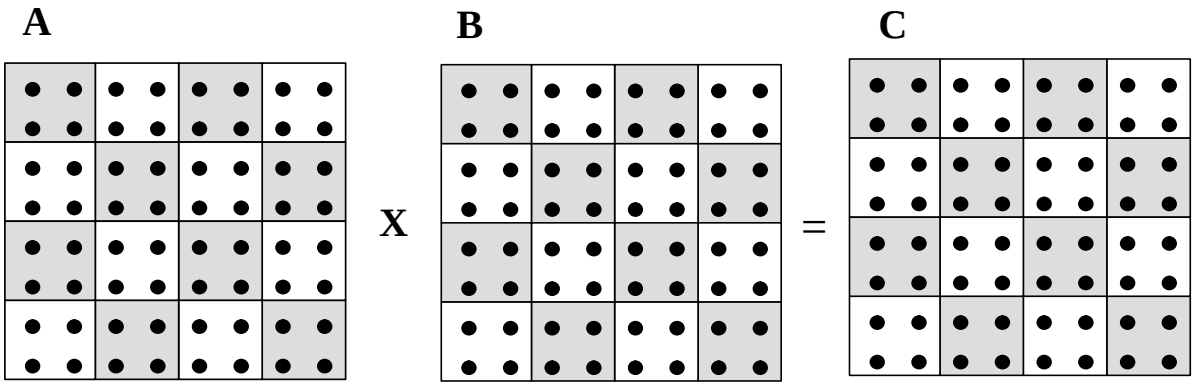
- Call this function instead of the function **`DummyDataInitialization()`**, which has been developed previously
- Add time measurement and printing
- Carry out the computational experiments with large objects
- Fill the table with results of experiments

PARALLEL ALGORITHM

Step 3. Parallel Algorithm (Fox Method)…

Analysis of Information Dependencies

- Let's use the **chessboard block scheme** of matrix partitioning



- In this case **base subtask** computes block C_{ij}

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

Step 3. Parallel Algorithm (Fox Method)…

Analysis of Information Dependencies

- 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

Step 3. Parallel Algorithm (Fox Method)…

Analysis of Information Dependencies

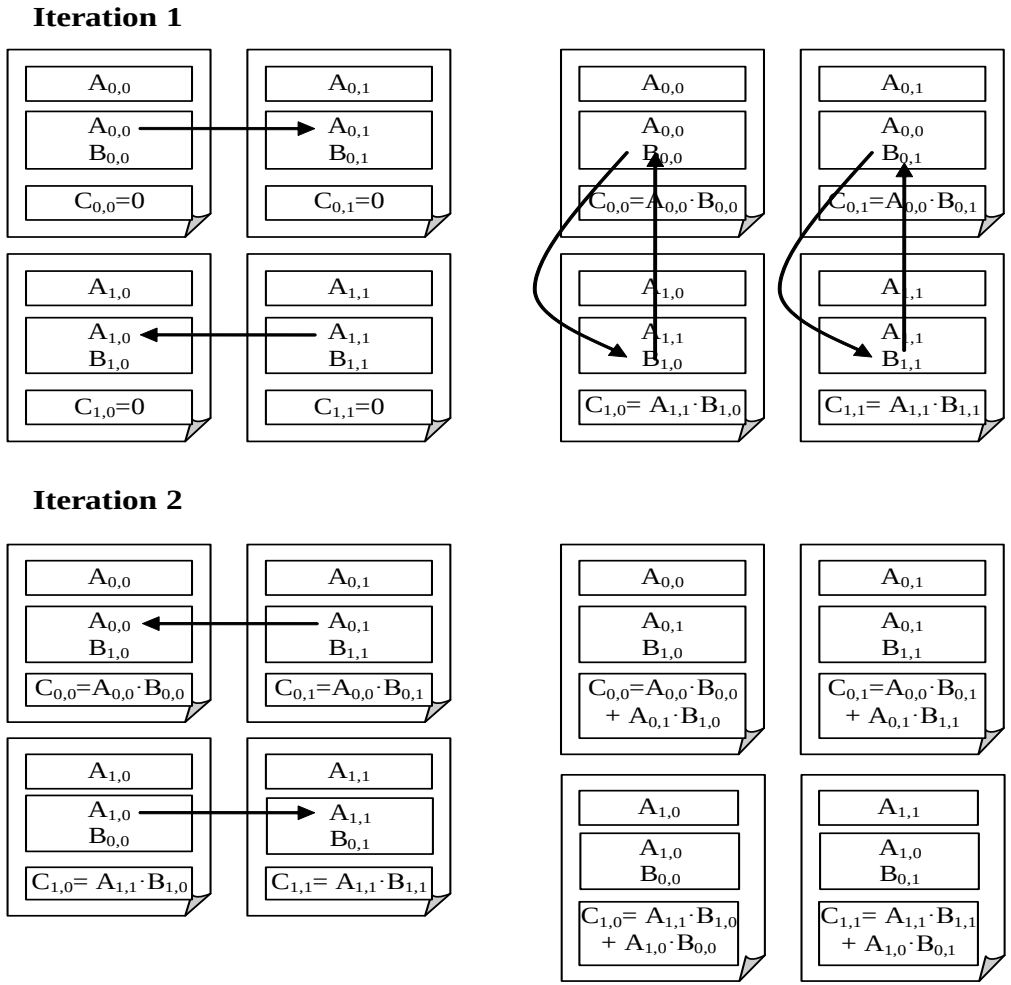
- Parallel algorithm execution includes
  - The initialization stage
    - Each subtask (i, j) obtains blocks A_{ij}, B_{ij}
    - All elements of blocks C_{ij} in all subtasks are set to zero
  - The computation stage
    - The block A_{ij} of subtask (i, j) is transmitted to all the subtasks of the same grid row; index j, which defines the position of the subtask in the row, is computed according to the following expression

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

    - 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}
    - 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)

Step 3. Parallel Algorithm (Fox Method)…

Computation Organization



Step 3. Parallel Algorithm (Fox Method)

Scaling and Distributing the Subtask among the Processors

- The size of blocks may be chosen so that the total number of the basic subtasks coincide with the number of processors p
- To execute the Fox algorithm efficiently the set of available processors should be arranged as a square grid
- 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 $P^{i,j}$.

PARALLEL PROGRAM

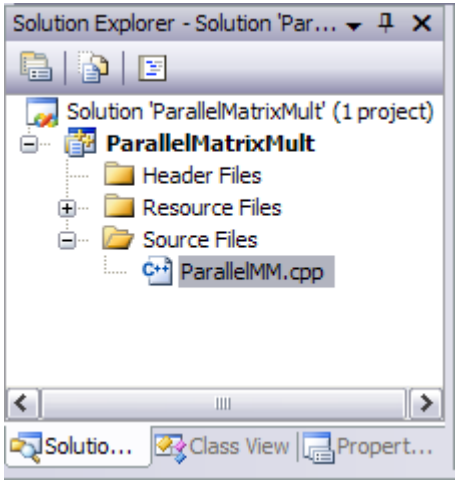
Step 4. Parallel Program…

- Task 1 – Open the Project ParallelMatrixMult
- Task 2 – Create the Virtual Cartesian Topology
- Task 3 – Input the Initial Data
- Task 4 – Terminate the Parallel Program
- Task 5 – Distribute the Data among the Processes
- Task 6 – Implement the Parallel Matrix Multiplication Program
- Task 7 – Broadcast the Blocks of the Matrix A
- Task 8 – Cyclic Shift the Blocks of the Matrix B along the Processor Grid Columns
- Task 9 – Implement the Matrix Block Multiplication
- Task 10 – Gather the Results
- Task 11 – Test the Parallel Program Correctness
- Task 12 – Carry out Computational Experiments

Step 4. Parallel Program…

Task 1 – Open the Project ParallelMatrixMult

- Start **Microsoft Visual Studio**
- Open solution ParallelMatrixVectorMult.sln from the folder **c:\ParLabs\ParallelMatrixMult**
- Open file **ParallelMM.cpp** in the window Solution Explorer (Ctrl+Alt+L)



Step 4. Parallel Program…

Task 1 – Open the Project ParallelMatrixMult

- The project contains the following functions
  - **`DummyDataInitialization()`** – simple data initialization
  - **`RandomDataInitialization()`** – random data initialization
  - **`SerialResultCalculation()`** – serial algorithm implementation
  - **`PrintMatrix()`** – matrix printing
- **`main()`** function contains declarations of variables **`ProcNum`**, **`ProcRank`**, **`pAMatrix`**, **`pBMatrix`**, **`pCMatrix`**, **`Size`**
- The environment of the MPI program is initialized, number of processes **`ProcNum`** and the rank of each process **`ProcRank`** is determined
- Compile and run the applications. Make sure that the initial message "`Parallel matrix multiplication program`" is output into the command console

Step 4. Parallel Program…

Task 2 – Create the Virtual Cartesian Topology

- According to the parallel computation scheme, it is necessary to arrange the available MPI program processes as a virtual topology in the form of a two-dimensional square grid
- It is only possible if the number of the available processes is a perfect square (**`ProcNum`** = **`GridSize`** × **`GridSize`**)
- Declare global variable **`GridSize`**
- In the function **`main()`** check whether **`ProcNum`** is a perfect square
- If no, print a diagnostic message
- If yes, compute value of the variable **`GridSize`**

Step 4. Parallel Program…

Task 2 – Create the Virtual Cartesian Topology

- Develop the function **`CreateGridCommunicators()`**
  - Create a communicator as a two-dimensional square grid
  - Determine the coordinates of each process in the grid
  - Create communicators for each row and each column separately

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

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

Step 4. Parallel Program…

Task 2 – Create the Virtual Cartesian Topology

- 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);
```

- **oldcomm** – the initial communicator
- **ndims** – the Cartesian grid dimension
- **dims** – the array of **ndims** length, it defines the number of processes in each dimension of the grid
- **periods** – the array of **ndims** length, which defines whether the grid is periodical along each dimension
- **reorder** – the parameter for pointing out if the process ranks can be reordered
- **cartcomm** – the communicator being created with the Cartesian process topology

Step 4. Parallel Program…

Task 2 – Create the Virtual Cartesian Topology

- Call the function `CreateGridCommunicators()` from the main function of the parallel application
- Compile application
- Execute application several times with different number of 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

Step 4. Parallel Program…

Task 3 – Input the Initial Data

- Declare variables necessary for parallel matrix multiplication (Fox algorithm)

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

Step 4. Parallel Program…

Task 3 – Input the Initial Data

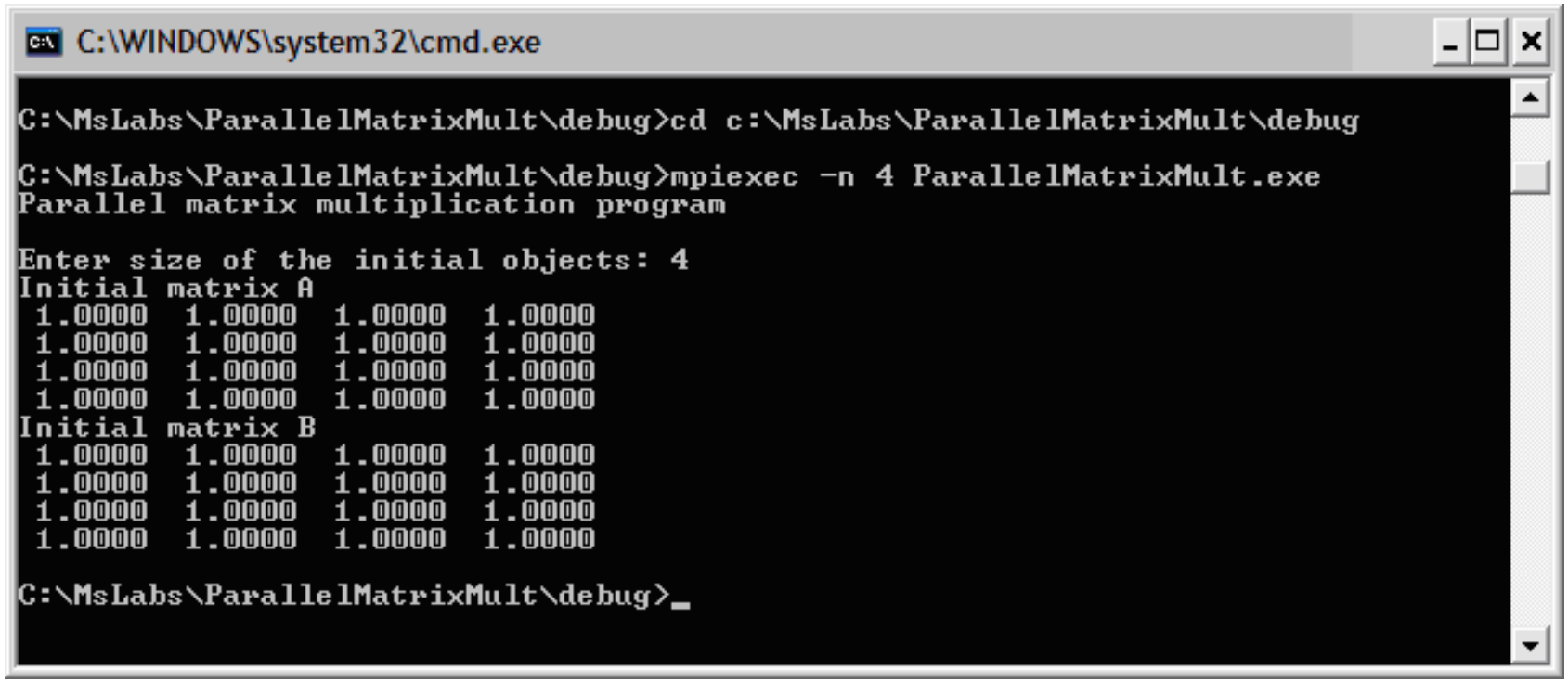
- Develop the function **`ProcessInitialization()`** in order to input the matrix sizes, set the matrix block sizes, allocate memory for storing them and initialize the matrix elements

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

- Call the function **`ProcessInitialization()`** from the main function of application
- To control the correctness of the initial data input use the function of the formatted matrix output **`PrintMatrix()`**
- Compile the application and run

Step 4. Parallel Program…

Task 3 – Input the Initial Data



Step 4. Parallel Program…

Task 4 – Terminate the Parallel Program

- Modify the function for correct program termination **`ProcessTermination()`**
- Deallocate the memory for storing the initial matrices **`pAMatrix`**, **`pBMatrix`** and **`pCMatrix`** (on the root process), and the memory for storing the blocks **`pMatrixAblock`**, **`pAblock`**, **`pBblock`**, **`pCblock`**

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

Step 4. Parallel Program…

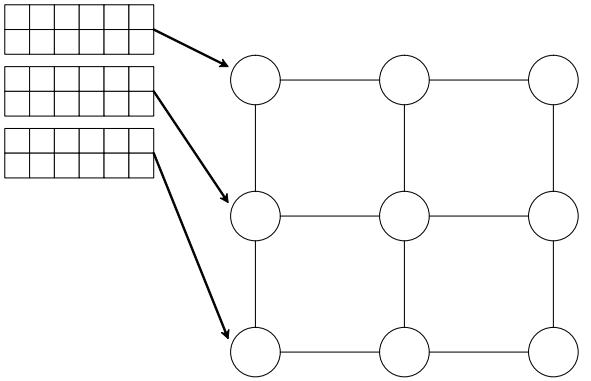
Task 5 – Distribute the Data among the Processes

- The initial matrices to be multiplied are located on the root process (with rank 0). It 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

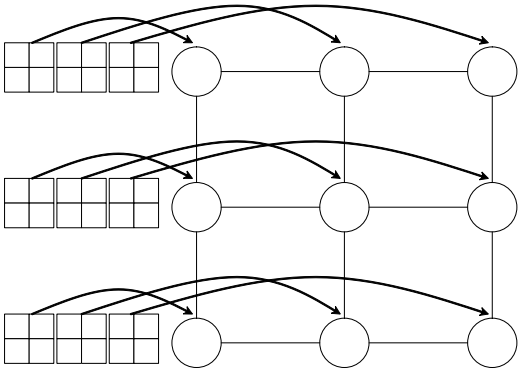
Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

- The First Stage of Data Distribution
  - 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



- The Second Stage of Data Distribution
  - Each stripe is further divided into blocks among the processes, which compose the processor grid rows



Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

- To distribute the matrix among the processor of grid processes blockwise, implement the function **`CheckerboardMatrixScatter()`**

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

- Use the function **`MPI_Scatter()`** to distribute the horizontal stripes among the processes, which compose the left column of the process grid, and to distribute each row of the horizontal stripe along the rows of the processor grid

Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

- 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`**)
- Develop the function **`DataDistribution()`**

```
// 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);
}
```

Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

- To control the correctness of initial data distribution implement the debugging print 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);
  }
}
```

Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

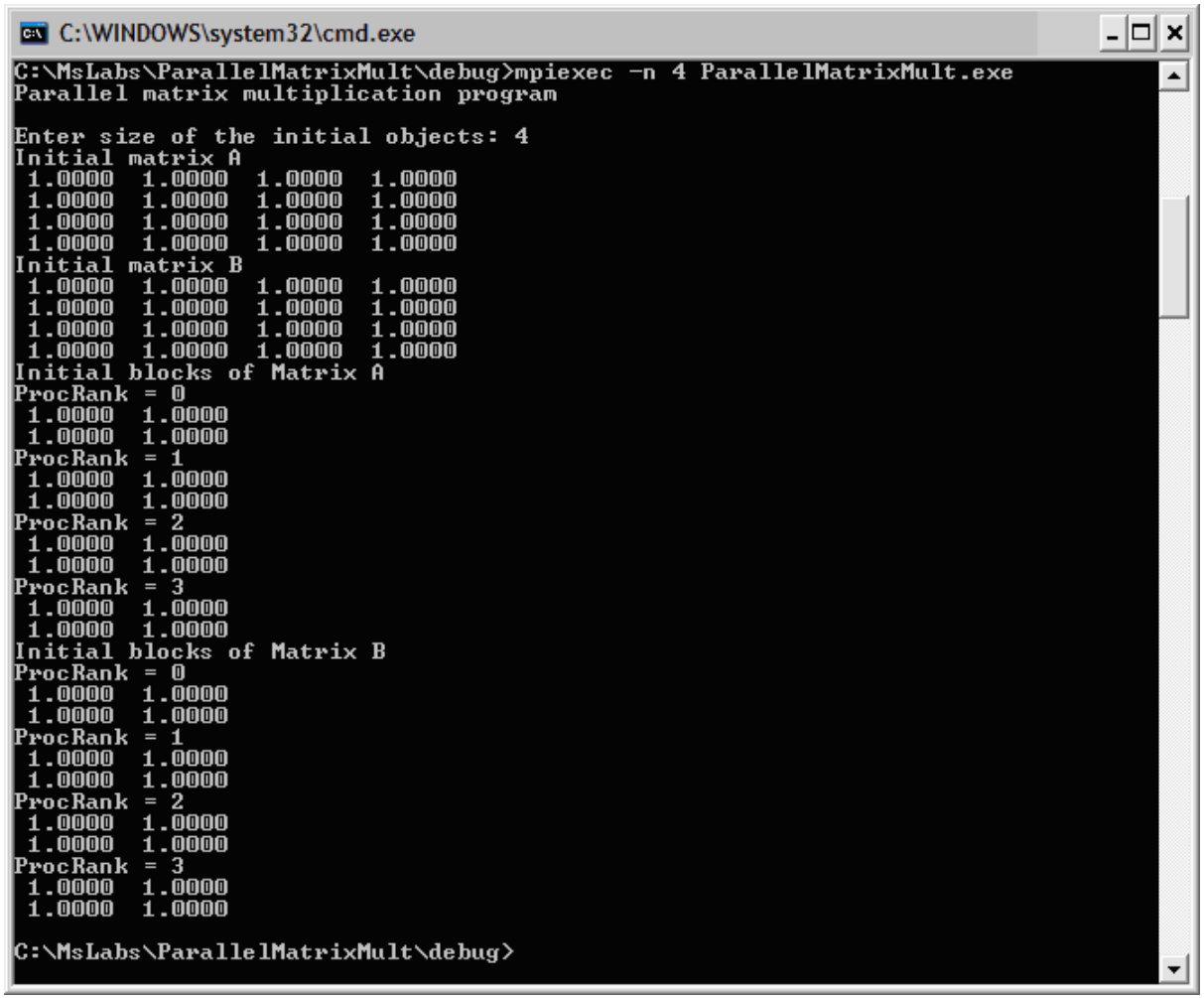
- Call the function of data distribution from the function **`main()`**
- Call the function of printing the distributed data from the function **`main()`**

```
  // 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 and run

Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes



Step 4. Parallel Program…

Task 6 – Implement the Parallel Matrix Multiplication Program

- Implement the **`ParallelResultCalculation()`** function to execute the parallel Fox algorithm of matrix multiplication
- According to the scheme of parallel computations, it is necessary to carry out **`GridSize`** iterations
- Each of the iterations consists of the execution of the following operations:
  - The broadcast of the matrix *A* block along the processor grid row
  - The multiplication of matrix blocks
  - The cyclic shift of the matrix *B* blocks along the column of the processor grid

Step 4. Parallel Program…

Task 6 – Implement the Parallel Matrix Multiplication Program

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

Step 4. Parallel Program…

Task 7 – Broadcast the Blocks of the Matrix A

- The number of the process *Pivot* in the row is determined according to the following expression

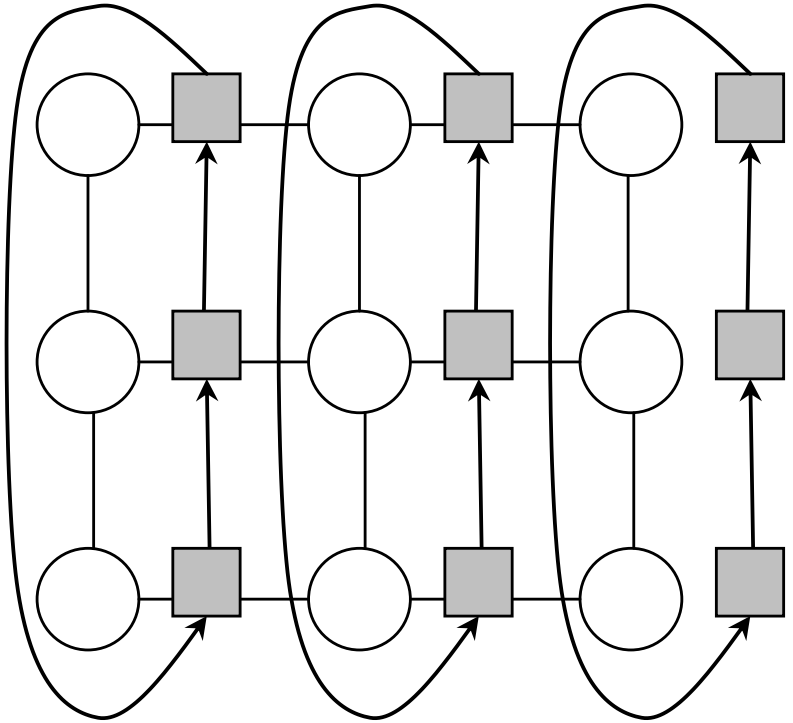
$$Pivot = (i + iter) \bmod GridSize,$$

  where *i* is the number of the processor grid row
- Implement the function **`ABlockCommunication()`** that
  - Determine **`Pivot`**
  - Copy the transmitted block **`pMatrixAblock`** in a memory buffer **`pAblock`**
  - Broadcast **`pAblock`**

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

Step 4. Parallel Program…

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



Step 4. Parallel Program…

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

- To perform the cyclic shift of blocks of the matrix B along the processor grid columns implement the function **`BblockCommunication()`**

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

Step 4. Parallel Program…

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

- To perform the cyclic shift of blocks of the matrix B along the processor grid columns implement the function **`BblockCommunication()`**

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

- The efficient execution of shifting can be implemented using the function **`MPI_Sendrecv_replace()`**

```
int MPI_Sendrecv_replace(void *buf, int count,
  MPI_Datatype type, int dest, int stag, int source,
  int rtag, MPI_Comm comm, MPI_Status* status);
```

Step 4. Parallel Program…

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

```
// 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);
}
```

Step 4. Parallel Program…

Task 9 – Implement the Matrix Block Multiplication

- Implement the function **`BlockMultiplication()`** to execute the multiplication of the block **`pAblock`** by the block **`pBblock`**

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

Step 4. Parallel Program…

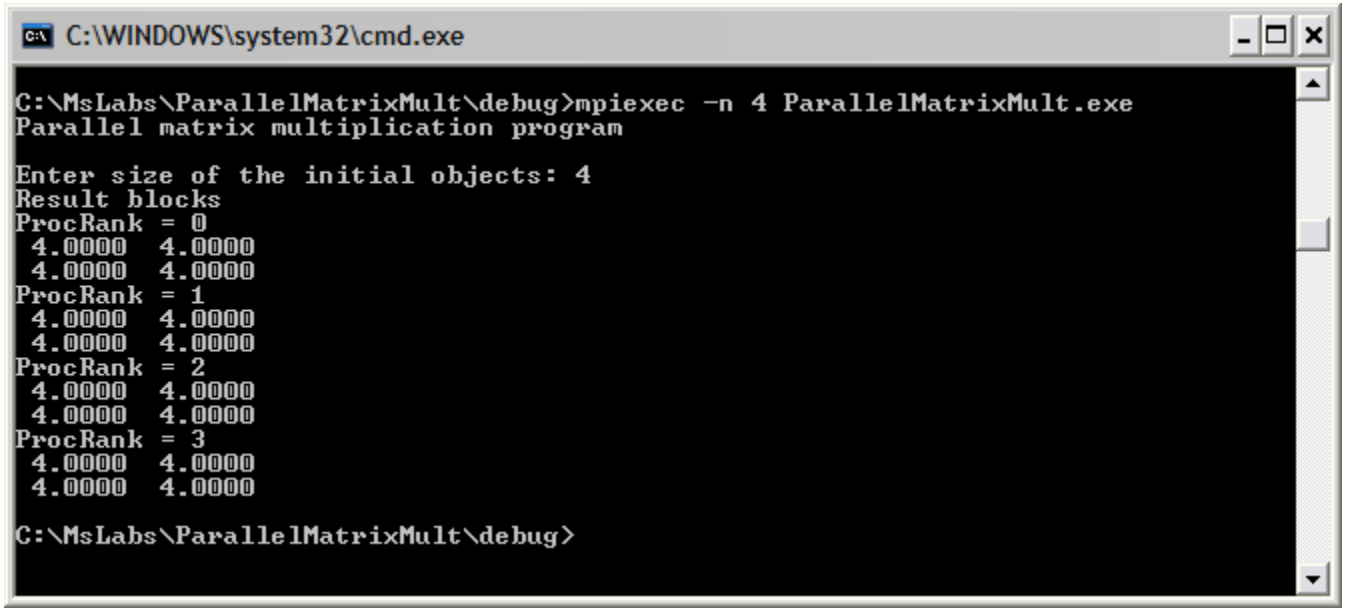
Task 9 – Implement the Matrix Block Multiplication

- Modify the function **`ParallelResultCalculation()`**
  - Call the function **`BlockMultiplication()`**
  - Comment debugging print
- Print out blocks of result matrix on each process by means of the function **`TestBlocks()`** in the function **`main()`**
- Compile the application and run

Step 4. Parallel Program…

Task 9 – Implement the Matrix Block Multiplication

- If the function **`DummyDataInitialization()`** is called 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



Step 4. Parallel Program…

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
  - to gather the blocks located on the processes of one process grid row into stripes of the result matrix
  - to gather the stripes located on the left process grid column into the matrix

Step 4. Parallel Program…

Task 10 – Gather the Results

- To gather the data use the function **`MPI_Gather()`**

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype,
               void *rbuf, int rcount, MPI_Datatype rtype,
               int root, MPI_Comm comm);
```

- **`sbuf, scount, stype`** – the parameters of the transmitted message
- **`rbuf, rcount, rtype`** – the parameters of the received message
- **`root`** – the rank of the process, on which the result must be obtained
- **`comm`** – the communicator, within of which the operation is executed

Step 4. Parallel Program…

Task 10 – Gather the Results

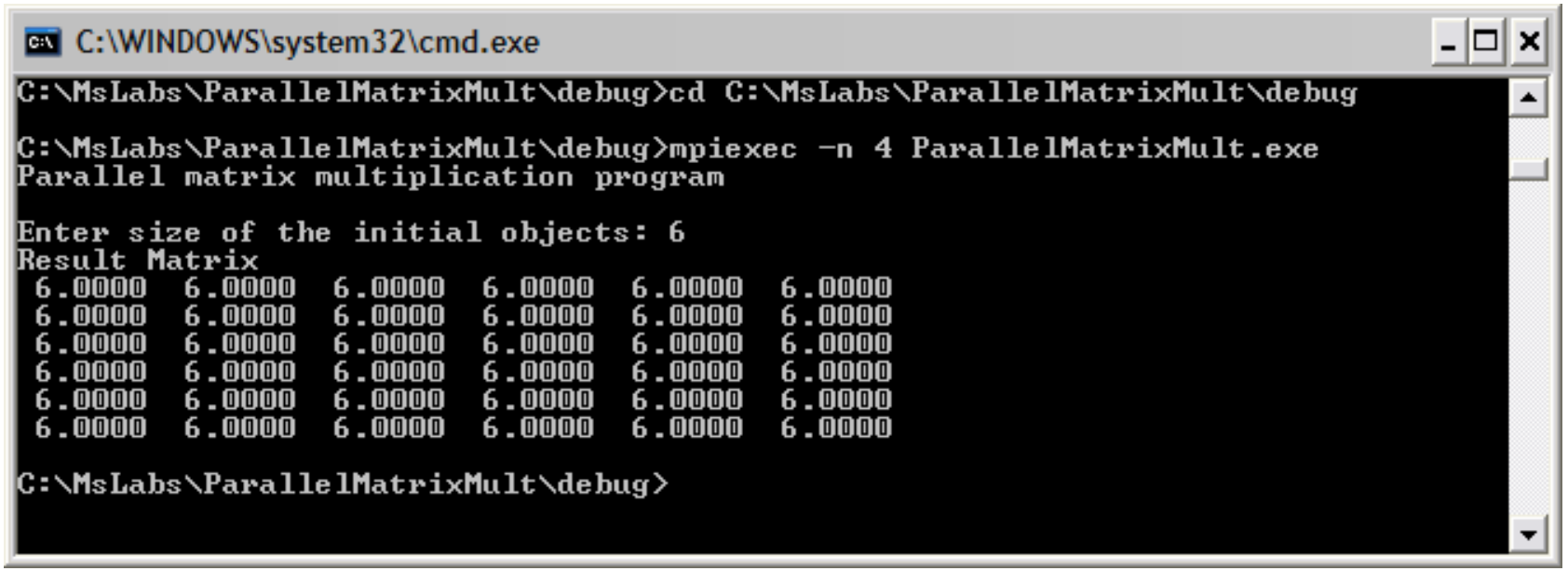
- Implement the procedure of gathering the result matrix *C* in the function **`ResultCollection()`**

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

- Call the function **`ResultCollection()`** form the function main
- Print the result matrix *C* after gathering on the root process
- Compile the application and run

Step 4. Parallel Program…

Task 10 – Gather the Results



Step 4. Parallel Program…

Task 11 – Test the Parallel Program Correctness

- To check the correctness of the parallel program execution implement the function **`TestResult()`**, which will compare the results of the serial and parallel algorithms element by element

```
void TestResult(double* pAMatrix, double* pBMatrix,
  double* pCMatrix, int Size);
```

- The function **`TestResult()`** should print the diagnostic message
- 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

Step 4. Parallel Program…

Task 11 – Test the Parallel Program Correctness

- Implement the function **`RandomDataInitialization()`**
- Call the function **`RandomDataInitialization()`** instead of **`DummyDataInitialization()`** in the function **`ProcessInitialization()`**
- Call the function **`TestResult()`** to the function **`main()`**
- Comment debugging print
- Compile the application and run
- Check that implemented Fox algorithm works correctly

Step 4. Parallel Program

Task 12 – Carry out the Computational Experiments

- Add time measurement of matrix multiplication execution
- Carry out the computational experiments with large objects
- Fill the table with results of experiments
- Determine the given speedup

Summary

- Block parallel method (Fox algorithm) of matrix multiplication is considered
- Serial and parallel methods of matrix multiplication are implemented
- Computational experiments are performed, comparison of serial and parallel algorithms is made

Exercises

- Modify the developed Fox algorithm implementation using the derived MPI data type for broadcasting and gathering matrix blocks
- Study the parallel algorithm of matrix multiplication based on block striped matrix partitioning. Develop a program implementation of this algorithm
- Study the Cannon parallel algorithm of matrix multiplication based on chessboard block matrix partitioning. Develop a program implementation of this algorithm

References

1. Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V. (1999). Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc for Industrial & Applied Math.
2. Quinn, M.J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
3. Kumar V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc. (2nd edn., 2003)
4. Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.
5. Foster, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.