Learning Lab 1: Parallel Algorithms of Matrix-Vector Multiplication

Matrices and matrix operations are widely used in mathematical modeling of processes, phenomena and systems. Matrix calculations are the basis of many research and engineering computations. The areas of the applications may be computational mathematics, physics, economics etc.

Being time-consuming matrix operations are a classical area for applying parallel computations. On the one hand, the use of high performance multiprocessor systems makes possible to significantly increase the complexity of the problems being solved. On the other hand, due to their simple way of formulation, matrix operations give a good opportunity to demonstrate many techniques and methods of parallel programming.

Lab Objective

The objective of this lab is to develop a parallel program, which performs matrix-vector multiplication. The lab assignments include:

- Exercise 1 – State the matrix-vector multiplication problem,

- Exercise 2 – Code the serial matrix-vector multiplication program,
- Exercise 3 – Develop the parallel matrix-vector multiplication algorithm,
- Exercise 4 – Code the parallel matrix-vector 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 7 "Parallel methods of matrix-vector multiplication". Besides, the preliminary lab "Parallel programming with MPI" is assumed to have been done.

Exercise 1 –State the Matrix-vector Multiplication Problem

As a result of multiplying the matrix A of the dimension $m \times n$ by the vector b, which consists of n elements, we obtain the vector c of the size m. Each i-th element of the vector is the result of scalar multiplications of i-th matrix A row (let us denote this row as a_i) and the vector b (see Figure 1.1):

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j \ , 1 \le i \le m \ . \tag{1.1}$$

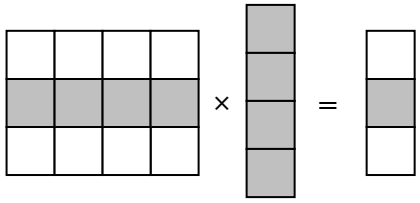

Figure. 1.1. The Element of the Result Vector is the Result of Scalar Matrix Row-Vector Multiplication

Thus, if the matrix composed of 3 rows and 4 columns is multiplied by the vector composed of 4 elements, we obtain the vector of size 3:

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \\ -6 \end{pmatrix}$$

Figure. 1.2. Test Example of Matrix-Vector Multiplication

Therefore, obtaining the result vector c assumes the execution of m operations of the same type of multiplying matrix A rows by the vector b. Each of these operations includes multiplying the elements of the matrix row by the vector b and further summing up of the products.

The pseudo code for the given matrix-vector multiplication algorithm may be as follows:

```
// Serial algorithm of matrix-vector multiplication
for (i = 0; i < m; i++){
  c[i] = 0;
  for (j = 0; j < n; j++){
    c[i] += A[i][j]*b[j]
  }
}
```

Exercise 2 – Code the Serial Matrix-Vector Multiplication Program

In this exercise, you will implement the serial matrix-vector multiplication algorithm. The initial version of the program to be developed is given in the project *SerialMatrixVectorMult*, which contains a part of the initial code and provides the necessary project parameters. The following operations have to be added to the given program version: matrix and vector size input, matrix and vector initialization, matrix-vector multiplication and result output.

Task 1 – Open the Project SerialMatrixVectorMult

Open the project *SerialMatrixVectorMult* using the following steps:

- Start **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\SerialMatrixVectorMult** in the dialog window **Open Project,**
- Make the double click on the file **SerialMatrixVectorMult.sln** or execute the command **Open** after choosing the file**.**

After the project has been opened in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *SerialMV.cpp*, as it is shown in Figure 1.3. After that, the code, which is to be enhanced, will be opened in the workspace of the Visual Studio.

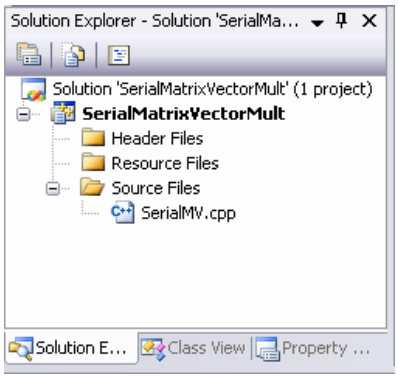

Figure. 1.3. Opening the File SerialMV.cpp

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

Let us consider the variables, which are used in the function *main* of the application. The first two of them (*pMatrix* и *pVector*) are correspondingly the matrix and the vector, which participate in matrix-vector multiplication as arguments. The third variable *pResult* is the vector to be obtained as a result of matrix-vector multiplication. The variable *Size* determines the sizes of the matrix and vector (it is assumed that the matrix *pMatrix* is square and has the dimension of *Size*×*Size*; it is multiplied by the vector of *Size* elements).:

```
double* pMatrix;  // First argument - initial matrix
double* pVector;  // Second argument - initial vector
double* pResult;  // Result vector for matrix-vector multiplication
int Size;         // Sizes of initial matrix and vector
```

It should be noted that the matrix *pMatrix* is stored rowwise in an one-dimensional array. Thus, the element located at the intersection of *i*-th row and *j*-th column of the matrix has the index *i*Size+j* in the one-dimensional array.

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-vector multiplication program\n");
getch();
```

Now it is possible to make the first application execution. Select the command **Rebuild Solution** in the menu **Build.** This command makes possible to compile and build 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 program start the following message will appear in the command console:

"`Serial matrix-vector multiplication program`".

In order to exit the program, press any key.

Task 2 – Input the Matrix and Vector Size

In order to set the initial data of the matrix-vector multiplication program we should implement the function *ProcessInitialization*. It is assigned for determining the sizes of the objects, allocating the memory for the objects

involved in multiplication (the initial matrix *pMatrix* and the vector *pVector*, and the result vector *pResult*). The function also sets the values of the initial matrix and vector elements. Thus, the function should have the following heading:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int &Size);
```

At the first stage it is necessary to determine object sizes (to set the value of the variable *Size*). The following code should be added to the function *ProcessInitialization*:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int &Size) {
  // Setting the size of the initial matrix and vector
  printf("\nEnter the size of the initial objects: ");
  scanf("%d", &Size);
  printf("\nChosen objects size = %d", Size);
}
```

The user is provided with the opportunity to input these object sizes (the sizes of the matrix and the vector), which will further be read from the standard input stream *stdin* and stored in the integer variable *Size*. Further the value of the variable *Size* is printed (Figure 1.4).

Add the call of the function *ProcessInitialization* to the main function after the initial message line:

```
void main() {
  double* pMatrix;  // First argument - initial matrix
  double* pVector;  // Second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;         // Sizes of initial matrix and vector

  printf ("Serial matrix-vector multiplication program\n");
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

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

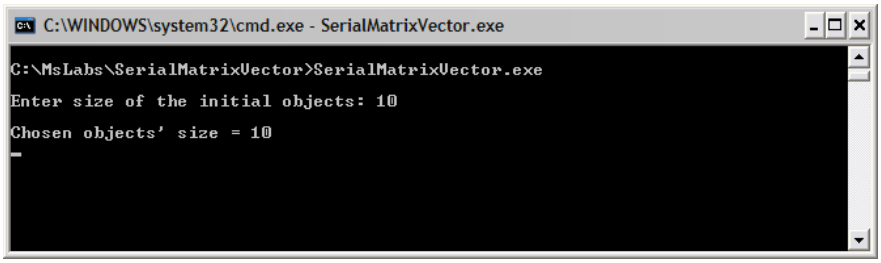

Figure. 1.4. Object Size Setting

Now let us consider the problem of correct input control. Thus, for instance, if the user is trying to enter a non-positive number as the object size, the application must either terminate the execution or continue to ask for the object size until a positive number is entered. Let us implement the second option. For this purpose the code fragment, which enters the object size, is placed in the post-conditioned loop:

```
  // Setting the size of the initial matrix and vector
  do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects 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 nonpositive number as the object size. Make sure that the invalid situation is processed correctly.

Task 3 – Input the Initial Data

The initialization function must also provide memory allocation for object storage (add the selected code to the function *ProcessInitialization*):

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int Size) {
  // Setting the size of the initial matrix and vector
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];
}
```

Further, it is necessary to set the values of all the initial object elements: the matrix *pMatrix* and the vector *pVector*. For this purpose the function *DummyDataInitialization* should be developed:

```
// Function for simple definition of matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

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

As it can be seen from the given code, this function provides setting the matrix and vector elements in rather a simple way: the value of matrix element coincides with the number of the row, in which it is located, and all the vector elements are equal to 1. That is in case when the user chooses the object size equal to 4, the following matrix and vector will be determined:

$$pMatrix = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix},\ pVector = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

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

The function *DummyDataInitialization* must be called after allocating memory inside the function *ProcessInitialization*:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int Size) {
  // Setting the size of the initial matrix and vector
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  <…>

  // Setting the values of the matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}
```

Let us develop two more functions, which help to control data input. These are the functions of the formatted object output: *PrintMatrix* and *PrintVector*. The arguments of *PrintMatrix* is the matrix *pMatrix*, the number of rows *RowCount* and the number of columns *ColCoun*. The arguments of *PrintVector* is the vector *pVector* and the number of elements *Size*.

```
// 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 formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
  printf("\n");
}
```

Add the call of these functions to the main application function:

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix: \n");
  PrintMatrix (pMatrix, Size, Size);
  printf ("Initial Vector: \n");
  PrintVector (pVector, Size);
```

Compile and run the application. Make sure that the data input is executed according to the above-described rules (Figure 1.5). Run the application several times setting various object sizes.

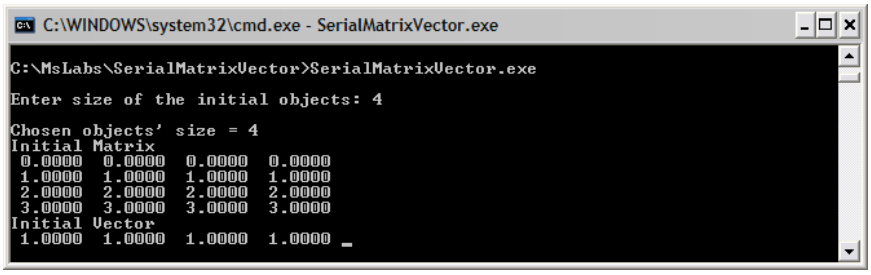

Figure. 1.5. The Result of Program Execution after Completion of Task 3

Task 4 – Terminate the Program Execution

Let us first develop the function for correct program termination, before executing 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 implement the corresponding function *ProcessTermination*. The memory has been allocated for storing the initial matrix *pMatrix* and the vector *pVector*, and also for storing the multiplication product *pResult*. These objects, consequently, should be given to the function *ProcessTermination* as arguments:

```
// Function for computational process termination
void ProcessTermination(double* pMatrix,double* pVector,double* pResult) {
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}
```

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

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix: \n");
  PrintMatrix (pMatrix, Size, Size);
  printf ("Initial Vector: \n");
  PrintVector (pVector, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

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

Task 5 – Implement the Matrix-Vector Multiplication

Let us develop the main computational part of the program. In order to multiply the matrix by the vector we will develop the function *ResultCalculation*, which gets the initial matrix *pMatrix* and the vector *pVector*, the size *Size* and the result vector *pResult*.

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

```
// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
    for (j=0; j<Size; j++)
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}
```

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

```
  // Memory allocation and initialization of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix: \n");
  PrintMatrix (pMatrix, Size, Size);
  printf ("Initial Vector: \n");
  PrintVector (pVector, Size);

  // Matrix-vector multiplication
  ResultCalculation(pMatrix, pVector, pResult, Size);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Compile and run the application. Analyze the results of the matrix-vector multiplication. If the program is executed correctly, the result vector must have the following structure: the *i*-th element of the result vector has to be equal to the product of vector *Size* by the number of the element *i*. Thus, if the object size *Size* is equal to 4, the result vector *pResult* must be the following: *pResult* = (0, 4, 8, 12). Carry out several computational experiments, changing the object sizes.

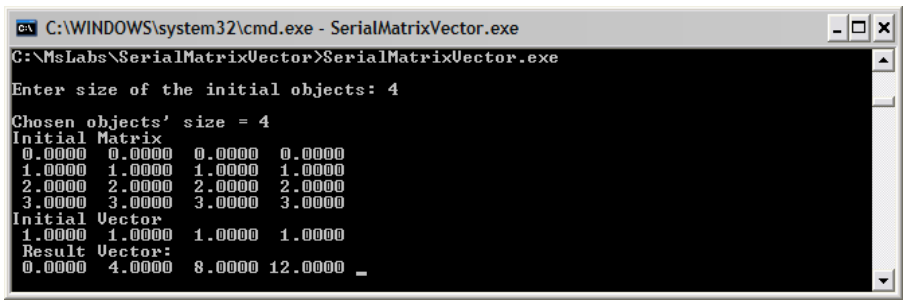

Figure. 1.6. The Results of Matrix-Vector Multiplication

Task 6 – Carry out the Computational Experiments

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

```
// Function for random setting  of the matrix and vector elements
void RandomDataInitialization (double* pMatrix,double* pVector,int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++)
      pMatrix[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 correctness of the program calculations easily.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int Size) {
  // Setting the size of the initial matrix and vector
  <…>

  // Memory allocation
  <…>

  // Random definition of matrix and vector elements
  RandomDataInitialization(pMatrix, pVector, Size);
}
```

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. We will use the following function

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

This function returns the number of processor ticks, which have passed since the system was started. Consequently, if this function is called twice, before and after the fragment under consideration, it is possible to compute its operation time. For instance, this fragment will calculate the duration of the function *f ()* operation.

```
time_t t1, t2;
t1 = clock();
f();
t2 = clock();
double duration = (t2-t1)/double(CLOCKS_PER_SEC);
```

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 compute time before and after the call of the function *ResultCalculation*:

```
// Matrix-vector multiplication
start = clock();
ResultCalculation(pMatrix, pVector, pResult, Size);
finish = clock();
duration = (finish-start)/double(CLOCKS_PER_SEC);

// Printing the result vector
printf ("\n Result Vector: \n");
PrintVector(pResult, Size);
// Printing the time spent by matrix-vector multiplication
printf("\n Time of execution: %f", duration);
```

Compile and run the application. In order to carry out the computational experiments with large objects, eliminate the matrix and vector outprint (comment on the corresponding code lines). Carry out the computational experiments and register the results in the following table:

Test Number	Matrix Size	Execution Time (sec)
1	10	
2	100	
3	1,000	
4	2,000	
5	3,000	
6	4,000	
7	5,000	
8	6,000	
9	7,000	
10	8,000	
11	9,000	
12	10,000	

In accordance with the computational algorithm of matrix-vector multiplication given in Exercise 1, obtaining the result vector implies the *Size* operations of multiplying the rows of the matrix *pMatrix* by the vector *pVector*. Each operation of this type includes multiplying the matrix row and vector elements (*Size* operations) and further summing up of the obtained products (*Size-1* operations). The total number of the necessary scalar operations is the following value:

$$N = Size \cdot (2 \cdot Size - 1). \tag{1.2}$$

In order to estimate the execution time for the parallel algorithm, it is necessary to know the duration τ of a single scalar operation execution. Thus, in order to compute the time of the algorithm execution, it is necessary to multiply the number of the executed operations by the time of a single scalar operation execution:

$$T_1 = N \cdot \tau = Size \cdot (2 \cdot Size - 1) \cdot \tau . \tag{1.3}$$

Let us fill out the table of comparison of the real execution time to the time, which may be obtained according to the formula (1.3). In order to compute the execution time of a single operation we will apply the following technique: choose one of the experiments as a pivot one. For instance, the experiment on matrix-vector multiplication of size 5000 can be considered as this pivot case. The experiment execution will be divisible by the number of the executed operations (the number of the operations may be calculated using formula (1.2)). Thus, we will calculate the execution time of a single scalar operation. Then using this value we will calculate the theoretical execution time for the remaining experiments. The results should be presented in the form of the following table:

Basic Computational Operation Execution Time τ (sec):			
Test Number	Matrix size	Execution Time (sec)	Theoretical time (sec)
1	10		
2	100		

3	1,000		
4	2,000		
5	3,000		
6	4,000		
7	5,000		
8	6,000		
9	7,000		
10	8,000		
11	9,000		
12	10,000		

It should be noted that the single scalar operation execution time depends generally on the size of the objects involved in multiplication. This dependence can be explained by the computer architecture properties. If the objects are very small, they can be wholly located in cache memory of the processor, and the access to the memory is fast. If the algorithm operates with medium size objects, which can be entirely located in RAM, but not in cache, the execution time for a single operation will be somewhat bigger, as the access time to the RAM is bigger as the access time to cache memory. If the objects are large enough to be located in the RAM, the swap file mechanism is involved. In this case the objects are stored on the external storage, and read and write memory time for this case increases significantly the recording time to the RAM. Thus, choosing an experiment as a pivot one (the experiment, for which the single operation execution time is calculated), we should be oriented at some average operation. That is why we have chosen the experiment on matrix-vector multiplication of size 5000.

Exercise 3 –Develop the Parallel Matrix-Vector Multiplication Algorithm

Parallelizing Principles

The development of algorithms (in particular, the methods of parallel computations) for solving complicated research and engineering problems can be a real challenge. Here we assume that the computational scheme for solving the problem of matrix-vector multiplication is already known. The activities for determining the efficient methods of parallel computations are the following:

- To analyze the available computation scheme and to decompose it into subtasks, which may be executed to a great degree independently,
- To select the information dependencies for the selected set of subtasks; these information dependencies should be carried out in the course of parallel computations,
- To determine the necessary or available computational system for solving the problem and to distribute the set of available subtasks among the system processors.

These stages of parallel algorithm development were first suggested by I. Foster. Foster's scheme is considered in detail in Section 6 of the training material.

Viewed in the large, it is obvious that the amount of computations for each processor must be approximately the same. It makes possible to provide equal computational load (balancing) of the processors. Besides, it is clear that the distribution of subtasks among the processors must be executed so that the number of the communication interactions among the subtasks is minimum.

Subtask Definition

The repetition of the same computational operations for different matrix elements is typical for different matrix calculation methods. It testifies to the existence of *data parallelism* in carrying out matrix calculations. As a result, parallelizing matrix operations can be reduced in most cases to distributing the processed matrices among the processors of the computational system. The choice of matrix distribution method determines the use of the concrete parallel computation method. The existence of various data distribution schemes generates a series of *parallel algorithms of matrix computations*.

Let us consider briefly the data distribution for matrix algorithms. This problem is considered in detail in Section 7 of the training material. The most general and the most widely used methods of matrix partitioning are block-striped matrix partitioning (vertically and horizontally) and chessboard block matrix partitioning.

1. Block-striped Matrix Partitioning. In case of *block-striped partitioning* each processor is assigned a certain subset of matrix rows (rowwise or horizontal partitioning) or matrix columns (columnwise or vertical partitioning) – see Figure 1.7a and 1.7b. Rows and columns are in most cases divided into stripes on a

continuous sequential basis. In case of such approach, in rowwise division, for instance, matrix A is represented as follows:

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = ik + j,\ 0 \le j < k,\ k = m/p,$$

where $a_i = (a_{i1}, a_{i2}, ... a_{in})$, $0 \le i < m$, is *i-th* row of matrix A (it is assumed, that the number of rows m is divisible by the number of processors p without a remainder, i.e. $m = k \cdot p$). Data partitioning on the continuous basis is used in all matrix multiplication algorithms and the algorithms of matrix-vector multiplication, which are studied in this and the following sections.

Another possible approach to forming rows is the use of a certain row or column alternation (circularity) scheme. As a rule, the number of processors p is used for the alternation. In this case the horizontal partitioning of matrix A looks as follows:

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = i + jp,\ 0 \le j < k,\ k = m/p.$$

The cyclic scheme of forming rows may appear to be useful for better balancing of computational load (for instance, it may be useful in case of solving a system of linear equations with the use of the Gauss method, see Section 9 of the training material).

2. Chessboard Block Matrix Partitioning. In this case the matrix is divided into rectangular sets of elements. As a rule, it is being done on a continuous basis. Let the number of processors be $p = s \cdot q$, the number of matrix rows is divisible by s, the number of columns is divisible by q, i.e. $m = k \cdot s$ and $n = l \cdot q$. Let us represent the original matrix A as a set of rectangular blocks in the following way (Figure 1.7c):

$$A = \begin{pmatrix} A_{00} & A_{02} & ...A_{0q-1} \\ & ... & \\ A_{s-11} & A_{s-12} & ...A_{s-1q-1} \end{pmatrix},$$

where A_{ij} is a matrix block, which consists of the elements:

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & ...a_{i_0 j_{l-1}} \\ & ... & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix},\ i_v = ik + v,\ 0 \le v < k,\ k = m/s,\ j_u = jl + u,\ 0 \le u \le l,\ l = n/q.$$

In case of this approach it is expedient that a computational system have a physical or at least a logical processor grid topology of s rows and q columns. Then, for data distribution on a continuous basis the processors neighboring in grid structure process adjoining blocks of the original matrix. It should be noted however that cyclic alteration of rows and columns can be also used for the chessboard block scheme.

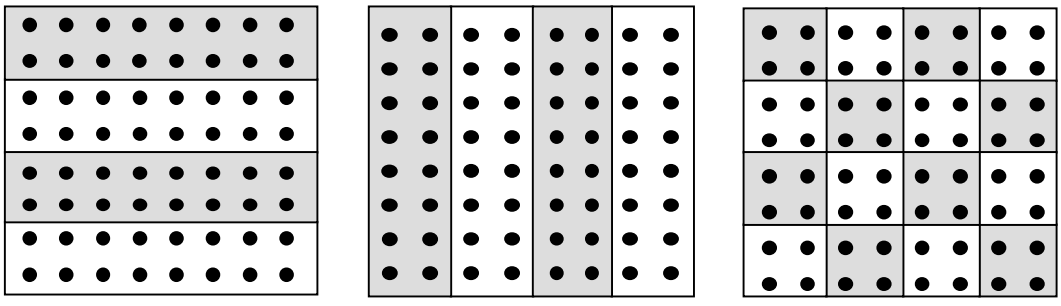

Figure. 1.7. Ways to Distribute the Matrix Elements among Processors

We will further discuss the algorithm of matrix-vector multiplication based on presenting a matrix as continuous sets (horizontal stripes) of rows. In this case of data partitioning we may choose the operation of scalar multiplication of a matrix row by the vector as the basic computational subtask.

Analysis of Information Dependencies

In order to execute the basic subtasks of scalar multiplication the processor should contain the corresponding row of the matrix *pMatrix* and a copy of the vector *pVector*. After the termination of the computations, each basic subtask determines one of the elements of the result vector *pResult*.

The general scheme of informational interaction among subtasks in the course of executed computations is shown in Figure 1.8.

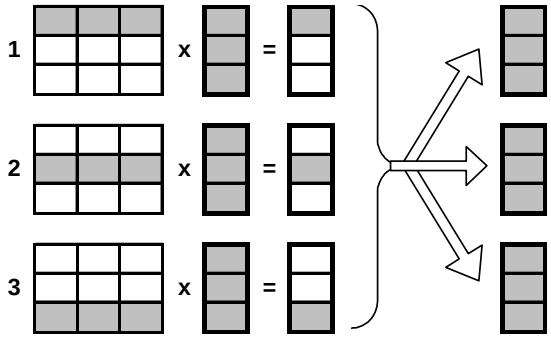

Figure. 1.8. Computation Organization in Case of Parallel Algorithm of Matrix-Vector Multiplication, Based on Rowwise Matrix Partition

In order to combine the calculation results and obtaining the whole vector *pResult* on each processor of the computational system, it is necessary to execute the all gather operation, when each processor transmits its computed element of the vector *c* to the rest of the processors. This step may be performed, for instance, with the help of the function *MPI_Allgather* from the library MPI (Figure 1.9).

Scaling and Distributing the Subtask among the Processors

In the process of multiplying the dense matrix by the vector the number of computational operations for obtaining the scalar product is the same for all the basic subtasks. Therefore, in case when the number of processors p is less than the number of basic subtasks m ($p<m$), we can combine the basic subtasks in such a way that each processor would execute several of these tasks. The tasks in their turn must correspond to the continuous sequence of rows of matrix *pMatrix* . In this case upon the completion of computations, each extended basic subtask determines several elements of the result vector *pResult*.

Subtasks distribution among the processors of the computational system may be executed in an arbitrary way.

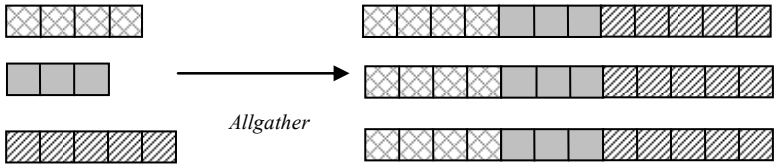

Figure. 1.9. Gather Communication Operation and Data Exchange among the Processors

Exercise 4 – Code the Parallel Matrix-Vector Multiplication Program

In order to perform the tasks, you will have to develop a parallel program of matrix-vector multiplication. For this purpose you should do the following:

- Study the fundamentals of MPI, the structure of MPI programs and several basic MPI functions,
- Get some basic experience in developing parallel programs.

The following structural parts may be selected in the parallel programs, which use the message passing interface (MPI):

- The initialization of the MPI program environment,
- The main part of the program, where the necessary algorithm of solving the stated problem is implemented and the message exchange among the program parts executed in parallel is performed,
- The MPI program termination.

Below you will find the general description of the fundamental concepts of MPI. This theme is considered in more detail in Section 4 of the training material.

Conception of Parallel Program with MPI

Within the frame of MPI a parallel program means a number of simultaneously carried out processes. The processes may be carried out on different processors. At the same time several processes may be located on a

single processor (in this case processes are carried out in the time-shared mode). In the extreme case a single processor may be used to carry out all the processes of parallel program. As a rule, this method is applied for the initial verification of the parallel program correctness.

The number of processes and the number of the processors used are determined at the moment of the parallel program start by the means of MPI program execution environment. These numbers must not be changed in the course of computations (standard MPI-2 provides the opportunity of the dynamic change of the processors number). All the program processes are sequentially enumerated from 0 to $p-1$, where p is the total number of processors. The process number is termed the process rank.

Introduction to Communicators and Groups of Processes

Parallel program processes are united into *groups*. *The communicator* in MPI is a specially designed control object, which unites within itself a group of processes and a number of complementary parameters (*context*), which are used in carrying out data communication operations.

As a rule, point-to-point data transmission operations are carried out for the processes, which belong to the same communicator. Collective operations are applied simultaneously to all the processes of the communicator. As a result, it is obligatory to point to the communicator being used for data communication operations in MPI.

In the course of computations new groups may be created and the already existing groups of processes and communicators may be deleted. The same process may belong to different groups and communicators. All the processes available in a parallel program belong to the communicator with the identifier MPI_COMM_WORLD, which is created on default.

Task 1 – Open the Project ParallelMatrixVectorMult

Open the project **ParallelMatrixVectorMult** 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,**
- Make the double click on the file **ParallelMatrixVectorMult.sln** or select it and execute the command **Open.**

After the project has been opened in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code **ParallelMV.cpp**, as it is shown in Figure 1.10. After that, the code, which has to be modified, will be opened in the workspace of Visual Studio

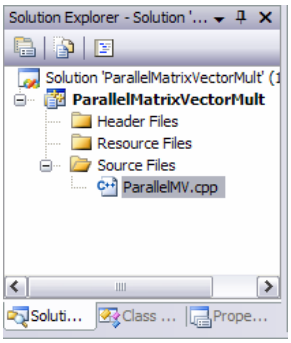

Figure. 1.10. Opening the File ParallelMV.cpp with the Use of Solution Explorer

The main function of the parallel algorithm to be developed, which contains the declarations of the necessary variables, is located in the file *ParallelMV.cpp***.** The following functions have been copied from the project, which contains the serial matrix-vector multiplication algorithm, are also located in the file *ParallelMV.cpp***:** *DummyDataInitialization*, *RandomDataInitialization*, *ResultCalculation*, *PrintMatrix* and *PrintVector* (the purposes of the functions are considered in detail in Exercise 2 of the lab). These functions may be also used in the parallel program. Besides, the frameworks for the functions of the computation initialization (*ProcessInitialization*) and process termination (*ProcessTermination*) are also located there.

Compile and run the application. Make sure that the initial message "`Parallel matrix-vector multiplication program`" is output into the command console.

Task 2 – Initialize and Terminate the Parallel Program

Before using MPI functions in the application, you should add the header file MPI to the program. For the application written in C/C++ the header file is called *mpi.h*. This file contains all the definitions and headings of

the MPI library functions. Add the bold marked line to the list of the libraries in the file of the parallel program initial code:

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

It is necessary to initialize the environment of the MPI program execution in the main function of the parallel program and to terminate its use of the environment on the program exit. Add the following bold marked code immediately after the block of the variable declarations:

```
void main(int argc, char* argv[]) {
  double* pMatrix;  // First argument - initial matrix
  double* pVector;  // Second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;         // Sizes of initial matrix and vector
  double Start, Finish, Duration;

  MPI_Init(&argc, &argv);
  printf ("Parallel matrix-vector multiplication program\n")
  MPI_Finalize();
}
```

The function *MPI_Init* initializes the environment of the MPI program execution. The arguments of the function *main* (the number *argc* of the command line arguments and the array *argv*, which contains these arguments*)* have to be given as the arguments of this function. The function *MPI_Init* must be called in each MPI program before calling any MPI functions. The function *MPI_Init* may be called only once in each program.

After the accomplishment of all the necessary operations before the program termination, it is necessary to close the MPI program execution environment. The function *MPI_Finalize* serves this purpose. Add the call of the function *MPI_Finalize* to the final line of the parallel program.

Let us consider the procedure of starting the parallel application. Compile the parallel application using **Visual Studio** (execute the command **Rebuild Solution** of the menu option **Build**). In order to run the parallel program you should start the program **Command prompt**, doing the following:

1. Press the button **Start**, and then execute the command **Run**,
2. Type the name of the program **cmd** in the dialog window, which appears on the screen (Figure 1.11).

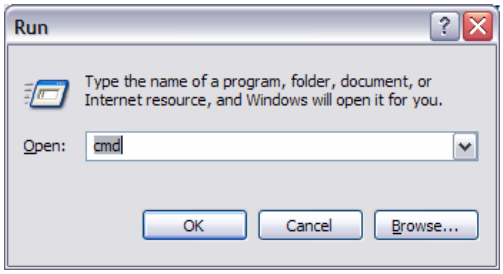

Figure. 1.11. The Start of Command Prompt

In the command line of the cmd program window go to the folder, which contains the developed program (Figure 1.12):

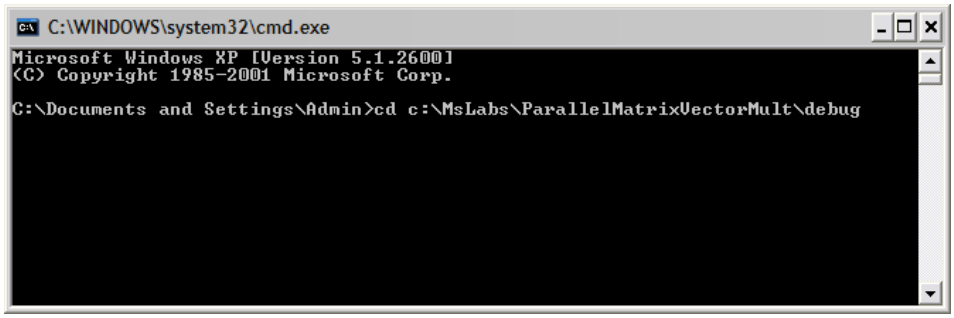

Figure. 1.12. Setting the Folder with the Parallel Program

To run the MPI program you should use the utility **mpiexec**. The call format looks generally as follows:

```
mpiexec –n <number of processes> <name of the executed module>
<arguments>
```

To run the parallel program, which contains 4 processes, type the command (Figure 1.13):

```
mpiexec –n 4 ParallelMatrixVectorMult.exe
```

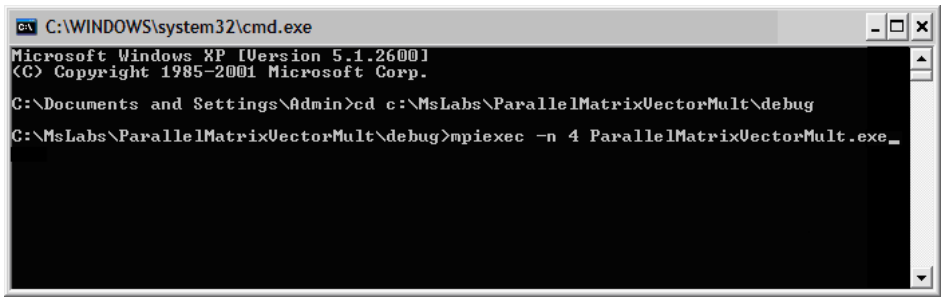

Figure. 1.13. Starting the Parallel Program

If everything has been done correctly, the following four identical initial message lines will appear on the command console: `"Parallel matrix-vector multiplication program"`, as each process of the parallel program performed printing (Figure 1.14).

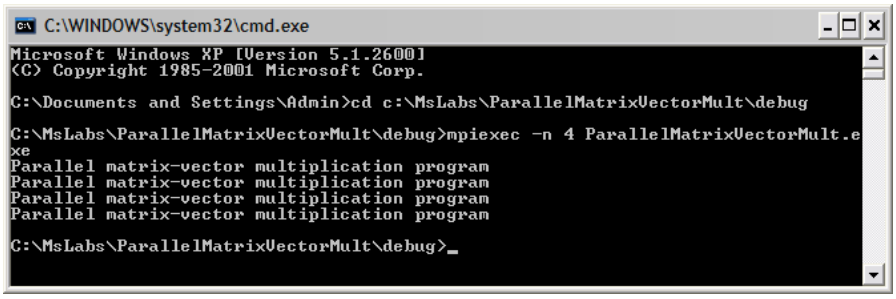

Figure. 1.14. The Result of the First Parallel Program Execution

Task 3 – Determine the Number of Processes

The number of processes in a parallel program is determined by means of the function *MPI_Comm_size*. The communicator, for which the number of processes is determined, is given in the function parameters (thus, in order to determine the total number of the processes available for MPI program, it is necessary to point out the communicator *MPI_COMM_WORLD*). To determine the rank of the process in the frames of the communicator you should use the function *MPI_Comm_rank* (it should be noted that each process in the communicator has a unique integer number – *rank*). Let us use the variables of the integer type for storing the number of the available processes *ProcNum* and the rank of the current process *ProcRank*. These values are usually used in all functions of the parallel program. So in order to make the variables available, let us declare *ProcNum* and *ProcRank* as global variables.

Add the bold marked lines to the corresponding place in the program:

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

void main(int argc, char* argv[]) {
  double* pMatrix;  // First argument - initial matrix
  double* pVector;  // Second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;         // Sizes of initial matrix and vector
  double Start, Finish, Duration;

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

  printf ("Parallel matrix-vector multiplication program\n")
```

```
  MPI_Finalize();
}
```

Let us print out the number of the available processes of the MPI program *ProcNum* and the rank of each process *ProcRank*. Add the marked lines to the main application function after the initial message line:

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

  printf (“Parallel matrix-vector multiplication program\n”)
  printf ("Number of available processes = %d \n", ProcNum);
  printf ("Rank of current process = %d \n", ProcRank);

  MPI_Finalize();
```

Compile and run the application using 4 processes. If everything has been done correctly, the result of the program operation will look as it is shown in Figure 1.15. Each process must print the following three lines: the initial message, the value of the number of processes, and its own rank. The value of the number of processes in all processes is the same, and the ranks are different. Pay attention to the fact that the ranks are not printed in order. Run the application several times. Make sure that the order of ranks printed can change from one execution of the program to another.

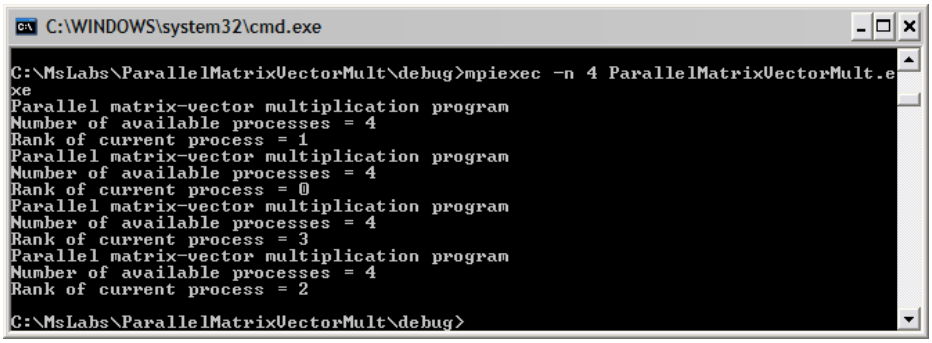

Figure. 1.15. Printing the Number and Ranks of the Processes

It is reasonable to introduce some changes into the code so that printing out the initial message and the number of the available processes is performed only by a process, for instance, by the process with the rank 0. Add the marked code to the application:

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

  if (ProcRank == 0) {
    printf (“Parallel matrix-vector multiplication program\n”);
    printf ("Number of abailable processes = %d \n", ProcNum);
  }
  printf ("Rank of current process = %d \n", ProcRank);

  MPI_Finalize();
```

Compile and run the application once again. Make sure that the initial message and the number of processes are printed only once. Start the application several times changing the number of the available processes.

Task 4 –Input the Matrix and Vector Size

Now let us pass over to the data input and output. As it is known from the Exercise 2, the development of the application, which includes matrix-vector multiplication, should begin with the setting the initial objects. It is necessary to initialize the size of the objects at the very first stage.

The function *ProcessInitialization* serves as previously for initializing the computations:

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int &Size);
```

In order to input the matrix and vector sizes it is necessary to realize the dialog with the user. This dialog should be provided by only one process. This process will be further referred to as *the root process* (usually it is the process with 0 rank). Add the marked code to the function *ProcessInitialization*:

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int &Size) {
  if (ProcRank == 0) {
    printf("\nEnter size of the matrix and vector: ");
    scanf("%d", &Size);
  }
}
```

Answering the question the user inputs the size of the matrix and vector, which is further read by the root process of the parallel program from the standard input stream *stdin* and stored in the variable *Size.* So after the execution of the parallel program the root process of the parallel system stored the entered size in the variable *Size.*

It has to be pointed again the invalid situations may occur in the course of the size input. Thus, for instance, as an matrix and vector sizes the user may point to the number, which is smaller than the number of available processes. Besides, for faster and simpler preparation of the first parallel program variant we will first assume that the size of the object is divisible without remainder by the number of processes. In this case all the processes will operate with the same number of the initial matrix rows and obtain the same number of the result vector elements (the program for the general case, when the matrix size is not divisible by the number of processes will be considered in Task 11). If the user inputs an incorrect matrix and vector size, the application must either terminate the execution or continue to ask for the size until the user inputs the "correct" number. Let us realize the second option, as it has been previously done. For this purpose we will enter the code fragment, which inputs the matrix and vector size, and put it in the post-conditioned loop:

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, int &Size) {
  if (ProcRank == 0) {
    do {
      printf("\nEnter size of the matrix and vector: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Size of the objects must be greater than "
               "number of processes! \n ");
      }
      if (Size%ProcNum != 0) {
        printf("Size of objects must be divisible by "
               "number of processes! \n");
      }
    }
    while ((Size < ProcNum) || (Size%ProcNum != 0));
  }
}
```

After the value of the variable *Size* is defined correctly, it is necessary to transfer the value to other processes. For this purpose we will use the MPI broadcast function. The function has the following heading:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type, int root,
  MPI_Comm comm),
where
 - buf, count, type – the memory buffer containing the message to be sent
          (for the process with the rank root), and for message reception
          for the other processes,
 - root – the process rank, which performs the data broadcasting,
 - comm – the communicator, within the frame of which data broadcasting is
          executed.
```

In this case it is necessary to transfer the value of the variable *Size* from the root process to the other processes:

```
  if (ProcRank == 0) {
    <…>
```

```
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Add the call of the function of the computation initialization instead of the lines, which print the number of the processes and their ranks:

```
void main(int argc, char* argv[]) {
    <…>

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

  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  MPI_Finalize();
}
```

Compile and run the application. Make sure that all the invalid situations are processed correctly. For this purpose run the application several times setting different number of the parallel processes (by means of the utility **mpiexec**) and various matrix and vector sizes.

Task 5 – Input the Initial Data

After the matrix and vector size has been determined, we may pass over to allocating memory and setting the values of the matrix and vector elements. Usually setting the initial data is accomplished by one of the processes (let the process be the process with the rank 0, as it has been previously). Further, according to the scheme of the parallel computations, given in Exercise 3, the initial matrix is distributed among all the processes so that each process processes a continuous set of rows (a horizontal stripe). It should be noted that the first version of the developed program is oriented at the case when the number of matrix rows is divisible by the number of processes without remainder, i.e. the matrix stripes on all the processes contain the same number of rows. This number of rows will be stored in the variable *RowNum*. The addresses of the memory buffers, which contain the horizontal stripes of rows on each of the processes, will be stored in the variable *pProcRows* (*pProcRows* is the matrix, which contains *RowNum* rows and *Size* columns rowwise). The initial vector *pVector* is copied from the root process to the other processes. As a result of matrix stripe – vector multiplication, each process obtains *RowNum* elements of the result vector. These elements will be stored in the array *pProcResult*.

We will declare the following variables in the main program function:

```
void main(int argc, char* argv[]) {
  double* pMatrix;     // First argument - initial matrix
  double* pVector;     // Second argument - initial vector
  double* pResult;     // Result vector for matrix-vector multiplication
  int Size;            // Sizes of initial matrix and vector
  double* pProcRows;   // Stripe of the matrix on current process
  double* pProcResult; // Block of result vector on current process
  int RowNum;          // Number of rows in matrix stripe
  double Start, Finish, Duration;
```

Let us change the list of arguments of the function *ProcessInitialization* so that this function can determine the value of the variable *RowNum* and allocate memory for storing new objects:

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcResult,
  int &Size, int &RowNum)
```

Let us determine the value of the variable *RowNum*, allocate memory for storing objects and initialize the initial matrix and vector on the leading process. Add the marked code to the function *ProcessInitialization*:

```
  if (ProcRank == 0) {
    <…>
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Determine the number of matrix rows stored on each process
  RowNum = Size/ProcNum;
```

```
  // Memory allocation
  pVector = new double [Size];
  pResult = new double [Size];
  pProcRows = new double [RowNum*Size];
  pProcResult = new double [RowNum];

  // Obtain the values of initial data
  if (ProcRank == 0) {
    // Initial matrix exists only on the root process
    pMatrix = new double [Size*Size];
    // Values of elements are defined only on the root process
    DummyDataInitialization(pMatrix, pVector, Size);
  }
```

To set the matrix and vector elements on the root process, we used the function of data generation *DummyDataInitialization*, which was developed in the course of implementation of the sequential application for matrix-vector multiplication. It should be noted that the function fills the vector *pVector* with the value 1, and the value of the element of the matrix *pMatrix* is equal to the number of the row, where the element is located.

In order to control the correctness of the initial data input, it is possible to use the functions *PrintMatrix* and *PrintVector*, which were developed in the course of implementation of the sequential application. After the call of the function *ProcessInitialization* in the main function, add the calls of the functions *PrintMatrix* and *PrintVector* for the matrix *pMatrix* and the vector *pVector* on the root process. Compile and run the application. Make sure that the data is input correctly.

Task 6 – Terminate the Calculations

In order to make the application complete at each stage of the development, we will develop the function for correct terminating computations. For this purpose it is necessary deallocate the memory, which was allocated dynamically in the process of program execution. Let us develop the corresponding function *ProcessTermination*. The memory for storing the initial matrix *pMatrix* was allocated on the root process, and the memory for storing the initial vector *pVector* and the result vector *pResult*, and also for storing matrix stripe *pProcRows* and the result vector block *pProcResult,* was allocated on all the processes. All these objects must be given to the function *ProcessTermination* as arguments:

```
// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult,
  double* pProcRows, double* pProcResult) {
  if (ProcRank == 0)
    delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
  delete [] pProcRows;
  delete [] pProcResult;
}
```

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

```
  // Process termination
  ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);
  MPI_Finalize();
}
```

Compile and run the application. Make sure that the application is functioning correctly.

Task 7 – Distribute the Data among the Processes

In accordance with the parallel computation scheme, given in the previous Exercise, the matrix must be distributed among the processes in equal horizontal stripes, and the initial vector must be copied onto all the processes.

The function *DataDistribution* is responsible for this. It should be provided with the initial matrix *pMatrix*, the vector *pVector*, the horizontal matrix stripes *pProcRows*, the size of the matrix and vector *Size* and the number of rows in the horizontal stripe *RowNum* as arguments:

```
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  int Size, int RowNum);
```

In order to copy the vector onto the parallel processes, we will use the broadcast function, as previously:

```
// Function for distribution of the initial data among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  int Size, int RowNum) {
  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

In case of this approach the matrix is stored in a one-dimensional array *pMatrix* rowwise. Consequently, in order to divide the matrix into horizontal stripes, it is necessary to divide the array into blocks of the same size and to broadcast the blocks to the processes. This operation is called Scatter communication (from one process to all the process of MPI program) (*data distribution*). This operation differs from broadcast transfer as a process transmits different data to all the other program processes. The execution of this operation may be provided by means of the following function:

```
int MPI_Scatter(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 transmitted message (scount
   defines the number of the elements transmitted onto each process),
 - rbuf, rcount, rtype – the parameters of the message received in the
         processes,
 - root – the rank of the process, which performs data transferring,
 - comm – the communicator within the frames of which data transmission is
         performed.
```

Add the call of the function *MPI_Scatter* to the function *DataDistribution*:

```
// Function for distribution of the initial data among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  int Size, int RowNum) {
  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(pMatrix, RowNum*Size, MPI_DOUBLE, pProcRows, RowNum*Size,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Correspondingly, it is necessary to call this function from the main program immediately after the call of the initialization function *ProcessInitialization*, before starting matrix-vector multiplication:

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult,
    Size, RowNum);

  // Distributing the initial data between the processes
  DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
```

Now let us test the correctness of the data distribution among the processes. For this purpose after the execution of the function *DataDistribution* we will print out the initial matrix and vector, and then the matrix stripes, which are distributed on each of the processes. Let us add one more function to the application. This function serves for testing the correctness of the data distribution. We will call the function *TestDistribution*.

In order to provide the formatted output of matrix and vector we will use the functions *PrintMatrix* and *PrintVector*:

```
void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
  int Size, int RowNum) {
  if (ProcRank == 0) {
    printf("Initial Matrix: \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector: \n");
    PrintVector(pVector, Size);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("\nProcRank = %d \n", ProcRank);
      printf(" Matrix Stripe:\n");
      PrintMatrix(pProcRows, RowNum, Size);
```

```
      printf(" Vector: \n");
      PrintVector(pVector, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

This method of checking the program correctness is called debugging print and is often used in the process of software development in the cases when the amount of the data to be checked is not big.

Let us clarify the implementation of the function *TestDistribution*. In some situations it is necessary to synchronize the computations independently executed in the processes. Synchronization of processes, i.e. simultaneous reaching some required computational points by the processes, is provided by means of the following MPI function:

```
int MPI_Barrier(MPI_Comm comm);
```

The function *MPI_Barrier* is an collective coomunication operation, and, thus, must be called by all the processes of the communicator being used. When the function *MPI_Barrier* is called, the execution of the process is blocked, and the computations will be continued only after the call of the function *MPI_Barrier* by all the communicator processes.

The function *MPI_Barrier* is used in the function *TestDistribution* in so that to provide the sequential order of the data output. Thus, it is first necessary to print the initial data on the root process. In order to prevent the other processes of the parallel program from being printed at the same time, the function *MPI_Barrier* is called. The execution of the operations on the other processes will start only after the root process calls MPI_Barrier on terminating the print of the initial data. This scheme is further used so that the processes print their matrix stripes in sequential order (first the process with the rank 0 prints its stripe, then the process with the rank 1 etc.).

Add the call of the function for data distribution testing immediately after the function *DataDistribution*:

```
// Distributing the initial data among the processes
DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);

// Distribution test
TestDistribution(pMatrix, pVector, pProcRows, Size, RowNum);
```

It should be noted that the function of initial data generation *DummyDataInitialization* is developed in such a way that it assigns the value of the matrix element equal to the number of the row, where it is located. Thus, after data distribution the matrix rows, which contain the values from *i*RowNum* to *(i+1)*RowNum-1* must appear on the process with the rank i.

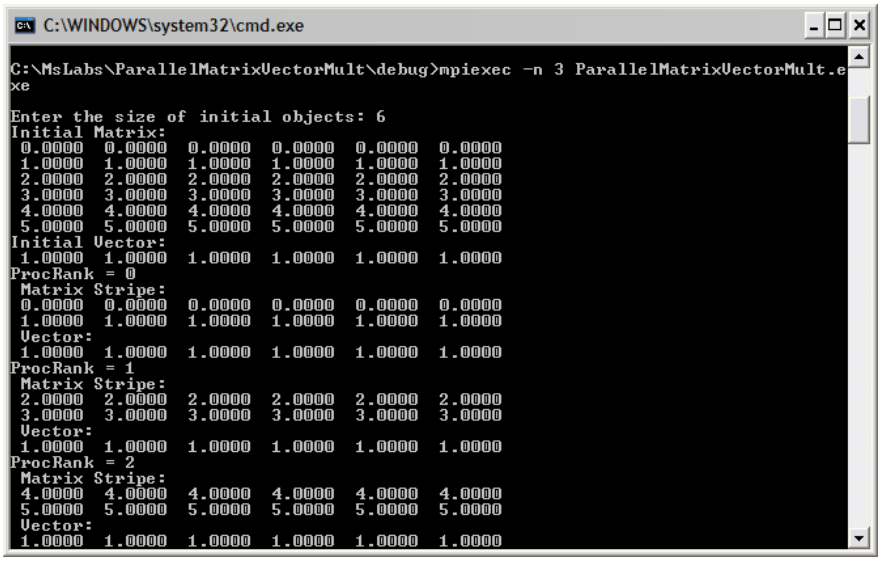

Figure. 1.16. Data Distribution in Case when the Application is Run Using Three Processes, and the Matrix Size is Equal to Six

Compile the application. If you find errors, correct them, comparing your code to the code given in the manual. Run the application using three processes and set the data size equal to 6. Make sure that data distribution is performed correctly (Figure 1.16).

Task 8 – Implement the Parallel Matrix-Vector Multiplication

The multiplication is executed in the function *ParallelResultCalculation*. In order to calculate a block of the result vector, it is necessary to have access to the matrix stripe *pProcRows*, the vector *pVector* and the result vector block *pProcResult*. It is necessary to know, besides, the sizes of these objects. Thus, it is necessary to provide the function *ParallelResultCalculation* the following arguments:

```
void ParallelResultCalculation(double* pProcRows, double* pVector,
  double* pProcResult, int Size, int RowNum);
```

To compute the value of the result vector element, it is necessary, as in the serial algorithm, to perform scalar multiplication of the matrix row by the vector. The difference from the serial code is in the fact that the process operates not with the matrix itself, but with its part *pProcRows* and processes not *Size*, but only *RowNum* of rows.

```
// Process rows and vector mulriplication
void ParallelResultCalculation(double* pProcRows, double* pVector,
  double* pProcResult, int Size, int RowNum) {
  int i, j;
  for (i=0; i<RowNum; i++) {
    pProcResult[i] = 0;
    for (j=0; j<Size; j++) {
      pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
    }
  }
}
```

Call the function *ParallelResultCalculation* in the main program in the following way:

```
// Distributing the initial objects among the processes
DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
TestDistribution(pMatrix, pVector, pProcRows, Size, RowNum);

// Parallel matrix vector multiplication
ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);
```

This stage should be tested as well as the previous ones. Let us develop for this purpose the function of testing the partial results, which were obtained by each of the processes, *TestPartialResults*. Let us make use of the debugging print again:

```
// Function for testing the multiplication of matrix stripe and vector
void TestPartialResults(double* pProcResult, int RowNum) {
  int i;    // Loop variable
  for (i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d \n", ProcRank);
      printf("Part of result vector: \n");
      PrintVector(pProcResult, RowNum);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

To decrease the amount of the debugging output, comment the call of the function *TestDistribution*. The call of the function *TestPartialResults* should be placed immediately after the execution of parallel multiplication:

```
DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
// TestDistribution(pMatrix, pVector, pProcRows, Size, RowNum);

// Parallel matrix vector mulriplication
ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);
TestPartialResults(pProcResult, RowNum);
```

For the matrices, the elements of which are set by means of the function *DummyDataInitialization*, the results of matrix-vector multiplication, is known beforehand. The result vector block, which contains the values from *Size*(i*RowNum)* to *Size*((i+1)*RowNum-1)*, is obtained on the process with the rank *i*. Thus, for instance, if a parallel application is run using two processes, and the matrix and vector size is equal to 6, the

block (0, 6, 12) must appear on the first process, and the second process will obtain the block (18, 24, 30) (see Figure 1.17).

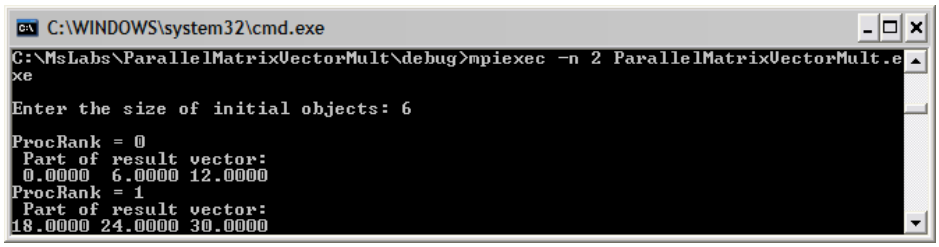

Figure. 1.17. The Result of Testing Partial Result Blocks of Matrix-Vector Multiplication in Case when Two Processes Are Used and the Matrix and Vector Size Is Equal to Six

Compile and run the application. Check the correctness of the obtained partial results according to the given formulas, changing the number of processes and the matrix and vector size.

Task 9 – Gather the Results

At the next stage it is necessary to gather the parts of the result vector located on different processes. As it has already been mentioned in Exercise 3, the MPI library provides the corresponding function *MPI_Allgather*, which combines the blocks located on different processes into a single array and copies this array onto all the processes. The function has the following interface:

```
int MPI_Allgather(void *sbuf,int scount,MPI_Datatype stype,
  void *rbuf,int rcount,MPI_Datatype rtype, MPI_Comm comm),
where
 - sbuf, scount, stype – the parameters of the transmitted message,
 - rbuf, rcount, rtype - the parameters of the received message,
 - comm – the communicator, within the frames of which data is transmitted.
```

The function *ResultReplication* is responsible for gathering results. It consists only of calling the function *MPI_Allgather*:

```
// Function for result vector replication
void ResultReplication(double* pProcResult, double* pResult, int Size,
  int RowNum) {
  MPI_Allgather(pProcResult, RowNum, MPI_DOUBLE, pResult, RowNum,
    MPI_DOUBLE, MPI_COMM_WORLD);
}
```

The call of the function from the main program:

```
  ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);

  // Result replication
  ResultReplication(pProcResult, pResult, Size, RowNum);
```

After the results are gathered, add the print of the result vector to the main function by means of the function *PrintVector* on all the parallel processes. Compile and run the application. Estimate the correctness of its operation.

Task 10 – Test the Parallel Program Correctness

After the function of the result collection is developed, it is necessary to test the correctness of the program execution. Let us develop the function *TestResult* for this purpose. It will compare the results of the serial and parallel programs. To execute the serial algorithm, it is possible to use the function *SerialResultCalculation*, developed in Exercise 2. The result of this function will be stored in the vector *pSerialResult*, and then we will compare this vector to the vector *pResult*, obtained by means of the parallel program element by element. The function *TestResult* must have access to the initial matrix *pMatrix* and the vector *pVector*. Consequently, it may be executed only on the root process:

```
// Testing the result of parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  double* pSerialResult; // Result of serial matrix-vector multiplication
  int equal = 0;  // =0, if the serial and parallel results are identical
```

```
  int i;          // Loop variable

  if (ProcRank == 0) {
    pSerialResult = new double [Size];
    SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);
    for (i=0; i<Size; i++) {
      if (pResult[i] != pSerialResult[i])
        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 result of the function is the print of the diagnostic message. It is possible to check the results of the parallel program using this message regardless of the initial matrix and vector size in case of any values of the initial data.

Comment on the calls of the functions, using the debugging print, which have been previously used for testing the correctness of parallel application (the functions *TestDistribution*, *TestPartialResult*). Instead of the function *DummyDataInitialization*, which generates matrices of simple type, call the function *RandomDataInitialization*, which generates the matrix and the vector by means of random data generator. Compile and run the application. Set various amounts of the initial data. Make sure that the application is functioning properly.

Task 11 – Implement the Computations for Any Matrix Sizes

The parallel application, which was developed in the course of executing the previous tasks, was aimed at the initial matrix and vector size *Size* divisible by the number of processors *ProcNum* without remaider. The matrix in this case is divided among the processes into equal stripes, the number of rows *RowNum*, which are operated by a process, is equal for all the processes.

Now let us consider the case, when the matrix and vector size *Size* is not divisible by the number of processes *ProcNum*. In this case the value *RowNum* of the number of the operated rows will be different on each process: some processes will obtain $\lfloor Size/ProcNum \rfloor$, and the rest of them - $\lceil Size/ProcNum \rceil$ of the matrix rows (the operation $\lfloor\ \rfloor$ means the rounding the value down to the nearest integer number, and operation $\lceil\ \rceil$ – means rounding up to the nearest integer number).

Now let us permit situations, occurring in case when the object size is not divisible by the number of processes without remainder, allowable the function *ProcessInitialization.* Now it is necessary to determine, how many rows should each process operate. One of the simplest methods may consist in the following: $\lfloor Size/ProcNum \rfloor$ matrix rows are assigned to all the processes except the last one (the process with the rank *ProcNum*-1). The last process is allocated all the other rows ($Size - \lfloor Size/ProcNum \rfloor \cdot (ProcNum - 1)$ rows). However, in this case the load may be distributed among the process unequally. Thus, for instance, if the matrix size is equal to five, and the parallel application is run using three processes, the first two processes will be allocated a single matrix row each, and the last processes will get three rows.

In order to avoid such inequality, we will use the distribution algorithm described below. Let us allocate the rows to the processes sequentially: first, let us determine how many rows will be processed by the process with rank 0, then by the process with the rank 1 etc. The process with the rank 0 will be allocated $\lfloor Size/ProcNum \rfloor$ rows (the operation result $\lfloor\ \rfloor$ coincides with the result of the integer division of the variable *Size* by the variable *ProcNum*). After the execution of the operation, we should distribute $Size - \lfloor Size/ProcNum \rfloor$ rows among the processes *ProcNum-1* etc. As a result, each next process *i* will be assigned the number of rows equal to the result of the integer division of the remaining row *RestRows* by the number of remaining processes, i.e. $\lfloor RestRows/(ProcNum - i) \rfloor$ rows.

Let us change the assignment of the value of the variable *RowNum* in the function *ProcessInitialization*:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcResult,
  int &Size, int &RowNum) {
  int RestRows; // Number of rows, that haven't been distributed yet
  int i;        // Loop variable

  if (ProcRank == 0) {
    do {
      printf("\nEnter size of the initial objects: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Size of the objects must be greater than
               number of processes! \n ");
      }
//    if (Size%ProcNum != 0) {
//      printf("Size of objects must be divisible by "
//             "number of processes! \n");
//    }
    }
    while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);

  <…>
}
```

In case when the matrix is distributed among process unequally, we cannot use the function *MPI_Scatter* for data distribution. Instead we should use the more general function *MPI_Scatterv*, which gives the opportunity to one of the processes to distribute the variable sized data among the communicator processes, including itself. The function has the following hearding:

```
MPI_Scatterv (void *send_buffer, int* send_cnt, int* send_disp,
  MPI_Datatype send_type, void *receive_buffer, int recv_cnt,
  MPI_Datatype recv_type, int root,MPI_COMM communicator ),
where
 - send_buffer –the data to be distributed,
 - send_cnt    – the i-th element – the number of sequential elements in
     send_buffer, assigned to the process i,
 - send_disp   – the i-th element – the offset of the first element,
     assigned to the process i with regard to the beginning of send_buffer,
 - send_type   – the element type in send_buffer,
 - recv_buffer - the buffer for receiving data by the process,
 - recv_cnt    – the number of elements to be received by the process,
 - recv_type   – the element type in recv_buffer,
 - root – the identifier of the process contaning the data to be scattered,
 - communicator – the communicator where scattering orccurs.
```

In order to call the function *MPI_Scatterv* it is necessary to define the two auxiliary arrays, the size of which coincides with the number of the available processes. Let us introduce the necessary changes in the code of the function *DataDistribution*:

```
// Function for distribution of the initial data among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  int Size, int RowNum) {
  int *pSendNum; // Number of elements sent to the process
  int *pSendInd; // Index of the first data element sent to the process
  int RestRows=Size; // Number of rows, that haven't been distributed yet

  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];

  // Determine the disposition of the matrix rows for current process
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (int i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}
```

Very much in the same way we will use the more general function *MPI_Allgatherv* for data gathering instead of the function *MPI_Allgather* oriented at gathering the data of the same size from all the communicator processes. The function *MPI_Allgatherv* has the following heading:

```
MPI_Allgatherv(void* send_buffer, int send_cnt, MPI_Datatype send_type,
  void* recv_buffer, int* recv_cnt, int* recv_disp, MPI_Datatype recv_type,
  MPI_Comm communicator),
where
 - send_buffer – the buffer, from which the process sends the data,
 - send_cnt    – the number of elements in send_buffer,
 - send_type   – element type in send_buffer,
 - recv_buffer – the buffer, which contains the result of gathering,
 - recv_cnt    – the i-th element is equal to the amount of data
     transmitted by the process with the rank i,
 - recv_disp   - the i-th element – the offset of the first element
     received from the process i with regard to the beginnig of recv_buffer,
 - recv_type   – the element type in recv_buffer,
 - communicator – the communicator where gathering takes place.
```

As in case of *MPI_Scatterv*, the use of *MPI_Allgatherv* requires using two additional arrays:

```
// Function for gathering the result vector
void ResultReplication(double* pProcResult, double* pResult, int Size,
  int RowNum) {
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  // Index of the first element from current process
                     // in result vector
  int RestRows=Size; // Number of rows, that haven't been distributed yet
  int i;             // Loop variable

  // Alloc memory for temporary objects
  pReceiveNum = new int [ProcNum];
  pReceiveInd = new int [ProcNum];

  // Determine the disposition of the result vector block
  pReceiveInd[0] = 0;
  pReceiveNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
    RestRows -= pReceiveNum[i-1];
    pReceiveNum[i] = RestRows/(ProcNum-i);
```

```
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
  }
  // Gather the whole result vector on every processor
  MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank], MPI_DOUBLE, pResult,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

  // Free the memory
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}
```

Compile and run the application. Check the correctness of the multiplication by means of the function *CheckResult*.

Task 12 – Carry out the Computational Experiments

The main challenge in the implementation of the parallel algorithms for solving complicated problems is to provide the increase of speed up (in comparison with the serial computations) by using several processors. Parallel program execution time is supposed to be less than that of the serial program.

Let us obtain the execution time of the parallel program. For this purpose we will add clocking start and finish time to the program code. It should be noted that there is a special function for clocking the time in MPI:

```
    MPI_Wtime();
```

As the parallel program includes the stage of data distribution, the computation of partial 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 *ResultReplication*:

```
  <…>
  Start = MPI_Wtime();
  // Distributing the initial data between the processes
  DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
// Parallel matrix vector mulriplication
  ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);
  // Result replication
  ResultReplication(pProcResult, pResult, Size, RowNum);
  Finish = MPI_Wtime();
  Duration = Finish-Start;

  TestResult(pMatrix, pVector, pResult, Size);
  if (ProcRank == 0) {
    printf("Time of execution = %f\n", Duration);
  }
  <…>
```

It is obvious that this way we will calculate the time to be spent on the execution of the calculations done by the root process. 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 main function of the parallel program. Compile and run the application. Fill out the table:

Matrix size	Serial algorithm	Parallel algorithm					
		2 processors		4 processors		8 processors	
		Time	Speed up	Time	Speed up	Time	Speed up
10							
100							
1,000							
2,000							
3,000							
4,000							
5,000							

6,000							
7,000							
8,000							
9,000							
10,000							

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

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

$$T_p = \lceil n/p \rceil \cdot (2n-1) \cdot \tau + \alpha \lceil \log_2 p \rceil + w \lceil n/p \rceil (2^{\lceil \log_2 p \rceil} - 1)/\beta \qquad (1.4)$$

(the detailed derivation of the formula is given in Section 7 of the training material). Here n is the matrix and vector size, p is the number of processors, 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 communication 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 (1.4). Tabulate the results in the following way:

Matrix size	2 processors		4 processors		8 processors	
	Model	Experiment	Model	Experiment	Model	Experiment
10						
100						
1,000						
2,000						
3,000						
4,000						
5,000						
6,000						
7,000						
8,000						
9,000						
10,000						

Discussions

- The time spent by the first process was chosen as the parallel execution time. What should be modified in the code in order to select the maximum execution time among all the time values obtained on all the processes?
- 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. Study the parallel algorithm of matrix-vector multiplication based on columnwise block-striped matrix partitioning. Develop a program implemention of this algorithm.

2. Study the parallel algorithm of matrix-vector 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-Vector Multiplication

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

// Function for simple setting the matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

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

// Function for random setting the matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++)
      pMatrix[i*Size+j] = rand()/double(1000);
  }
}

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {

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

  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];
  // Setting the values of the  matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}

// 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 formatted vector output
```

```
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
    for (j=0; j<Size; j++)
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}

// Function for computational process termination
void ProcessTermination(double* pMatrix,double* pVector,double* pResult) {
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}

void main() {
  double* pMatrix;  // First argument - initial matrix
  double* pVector;  // Second argument - initial vector
  double* pResult;  // Result vector for matrix-vector multiplication
  int Size;           // Sizes of initial matrix and vector
  time_t start, finish;
  double duration;

  printf("Serial matrix-vector multiplication program\n");

  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Matrix-vector multiplication
  start = clock();
  ResultCalculation(pMatrix, pVector, pResult, Size);
  finish = clock();
  duration = (finish-start)/double(CLOCKS_PER_SEC);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

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

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Appendix 2 – The Program Code of the Parallel Application for Matrix-Vector Multiplication

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

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

// Function for simple setting the matrix and vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

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

// Function for random setting the matrix and vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++)
      pMatrix[i*Size+j] = rand()/double(1000);
  }
}

// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcResult,
  int &Size, int &RowNum) {
  int RestRows; // Number of rows, that haven't been distributed yet
  int i;        // Loop variable

  setvbuf(stdout, 0, _IONBF, 0);
  if (ProcRank == 0) {
    do {
      printf("\nEnter the size of the matrix and vector: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Size of the objects must be greater than
                number of processes! \n ");
      }
    }
    while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Determine the number of matrix rows stored on each process
  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);

  // Memory allocation
  pVector = new double [Size];
```

```
  pResult = new double [Size];
  pProcRows = new double [RowNum*Size];
  pProcResult = new double [RowNum];

  // Setting the values of the matrix and vector elements
  if (ProcRank == 0) {
    // Initial matrix exists only on the root process
    pMatrix = new double [Size*Size];
    // Values of elements are defined only on the root process
    RandomDataInitialization(pMatrix, pVector, Size);
  }
}

// Function for distribution of the initial data between the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  int Size, int RowNum) {
  int *pSendNum; // Number of elements sent to the process
  int *pSendInd; // Index of the first data element sent to the process
  int RestRows=Size; // Number of rows, that haven't been distributed yet

  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Alloc memory for temporary objects
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];

  // Define the disposition of the matrix rows for current process
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (int i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }

  // Scatter the rows
  MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}

// Function for result vector replication
void ResultReplication(double* pProcResult, double* pResult, int Size,
  int RowNum) {
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  // Index of the first element from current process
                     // in result vector
  int RestRows=Size; // Number of rows, that haven't been distributed yet
  int i;             // Loop variable

  //Alloc memory for temporary objects
  pReceiveNum = new int [ProcNum];
  pReceiveInd = new int [ProcNum];

  //Define the disposition of the result vector block of current processor
  pReceiveInd[0] = 0;
  pReceiveNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
```

```
    RestRows -= pReceiveNum[i-1];
    pReceiveNum[i] = RestRows/(ProcNum-i);
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
  }
  //Gather the whole result vector on every processor
  MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank], MPI_DOUBLE, pResult,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

  //Free the memory
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}

// Function for sequential matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pResult[i] = 0;
    for (j=0; j<Size; j++)
      pResult[i] += pMatrix[i*Size+j]*pVector[j];
  }
}

// Process rows and vector multiplication
void ParallelResultCalculation(double* pProcRows, double* pVector,
  double* pProcResult, int Size, int RowNum) {
  int i, j;  // Loop variables
  for (i=0; i<RowNum; i++) {
    pProcResult[i] = 0;
    for (j=0; j<Size; j++)
      pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
  }
}

// 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 formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
  int Size, int RowNum) {
  if (ProcRank == 0) {
    printf("Initial Matrix: \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector: \n");
    PrintVector(pVector, Size);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  for (int i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
```

```
      printf("\nProcRank = %d \n", ProcRank);
      printf(" Matrix Stripe:\n");
      PrintMatrix(pProcRows, RowNum, Size);
      printf(" Vector: \n");
      PrintVector(pVector, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Fuction for testing the multiplication of matrix stripe and vector
void TestPartialResults(double* pProcResult, int RowNum) {
  int i;    // Loop variables
  for (i=0; i<ProcNum; i++) {
    if (ProcRank == i) {
      printf("\nProcRank = %d \n Part of result vector: \n", ProcRank);
      PrintVector(pProcResult, RowNum);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Testing the result of parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  double* pSerialResult; // Result of serial matrix-vector multiplication
  int equal = 0;  // =0, if the serial and parallel results are identical
  int i;          // Loop variable

  if (ProcRank == 0) {
    pSerialResult = new double [Size];
    SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);
    for (i=0; i<Size; i++) {
      if (pResult[i] != pSerialResult[i])
        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* pMatrix, double* pVector, double* pResult,
  double* pProcRows, double* pProcResult) {
  if (ProcRank == 0)
    delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
  delete [] pProcRows;
  delete [] pProcResult;
}

void main(int argc, char* argv[]) {
  double* pMatrix;     // First argument - initial matrix
  double* pVector;     // Second argument - initial vector
  double* pResult;     // Result vector for matrix-vector multiplication
  int     Size;        // Sizes of initial matrix and vector
  double* pProcRows;   // Stripe of the matrix on the current process
  double* pProcResult; // Block of the result vector on the current process
```

```
  int RowNum;          // Number of rows in the matrix stripe
  double Start, Finish, Duration;

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

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

  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult,
    Size, RowNum);

  Start = MPI_Wtime();

  // Distributing the initial data between the processes
  DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);

  // Parallel matrix-vector multiplication
  ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);

  // Result replication
  ResultReplication(pProcResult, pResult, Size, RowNum);

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

  TestResult(pMatrix, pVector, pResult, Size);
  if (ProcRank == 0) {
    printf("Time of execution = %f\n", Duration);
  }

  // Process termination
  ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);

  MPI_Finalize();
}
```