Learning Lab 3: Parallel Methods of Solving the Linear Equation Systems

Linear equation systems appear in the course of solving a number of applied problems, which are described by differential or integral equations or by systems of non-linear (transcendent) equations. They may appear also in the problems of mathematical programming, statistic data processing, function approximation, or in discretization of boundary differential problems by methods of finite differences or of finite elements, etc.

This lab discusses one of the direct methods of solving linear equation systems, i.e. the Gauss method and its parallel generalization.

Lab Objective

The objective of this lab is to develop a parallel program for solving the linear equation systems by means of the Gauss method. The lab assignments include:

- Exercise 1 – Stating the problem of solving the linear equation systems.
- Exercise 2 – Studying the Gauss algorithm for solving the linear equation systems.
- Exercise 3 – The Realization of the Sequential Gauss Algorithm.

- Exercise 4 – Developing the parallel Gauss algorithm.
- Exercise 5 – Coding the parallel program for solving the linear equation systems.

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 9 "Parallel methods of solving the linear equation systems" of the training material. Besides, the following labs are assumed to have been done: the preliminary lab "Parallel programming with MPI", Lab 1 "Parallel algorithms of matrix-vector multiplication" and Lab 2 "Parallel algorithms of matrix multiplication".

Exercise 1 – State the Problem of Solving the Linear Equation Systems

Linear equation with n independent unknowns $x_0, x_1, \ldots, x_{n-1}$ may be described by means of the expression

$$a_0 x_0 + a_1 x_1 + \ldots + a_{n-1} x_{n-1} = b \qquad (3.1)$$

where values $a_0, a_1, \ldots, a_{n-1}$ and b are constant.

Set of n linear equations

$$\begin{aligned} a_{0,0}x_0 \quad &+a_{0,1}x_1 \quad &+\ldots+a_{0,n-1}x_{n-1} \quad &= b_0 \\ a_{1,0}x_0 \quad &+a_{1,1}x_1 \quad &+\ldots+a_{1,n-1}x_{n-1} \quad &= b_1 \\ &\ldots & & \\ a_{n-1,0}x_0 \quad &+a_{n-1,1}x_1 \quad &+\ldots+a_{n-1,n-1}x_{n-1} \quad &= b_{n-1} \end{aligned} \qquad (3.2)$$

is termed *a system of linear equations* or *a linear system*. In brief (matrix) form this system may be presented as

$$Ax = b\ ,$$

where $A=(a_{i,j})$ is a real matrix of size $n \times n$, and vectors b and x are composed of n elements.

The *problem of solving a system of linear equation* for the given matrix A and the vector b is considered to be the problem of searching the value of unknown vector x whereby all the system equations hold.

Exercise 2 - Studying the Gauss Algorithm for Solving the Linear Equation Systems

The Gauss method is a well-known *direct* algorithm of solving systems of linear equations, the coefficient matrices of which are dense. If a system of linear equations is nondegenerate, then the Gauss method guarantees solving the problem with the error determined by the accuracy of computations. The main concept of the method is a modification of matrix A by means of equivalent transformations (which do not change the solution of system (3.2)) to a triangle form. After that the values of the desired unknown variables may be obtained directly in an explicit form.

The Exercise gives the general description of the Gauss method, which is sufficient for its initial understanding and which allows to consider possible methods of parallel computations in solving the linear equation systems.

The Gauss method is based on the possibility to carry out the transformation of linear equations, which do not change the solution of the system under consideration (such transformations are referred to as equivalent). They include the following transformations:

- the multiplication of any equation by a nonzero constant,
- the permutation of equations,
- the addition of any system equation to other equation.

The Gauss method includes sequential execution of two stages. At the first stage (*the Gaussian elimination stage*) the initial system of linear equations is reduced to the upper triangle form with the use of sequential elimination of unknowns.

$$Ux = c\ ,$$

where the coefficient matrix of the obtained system looks as follows:

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ & & \dots & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}.$$

At *the back substitution* (the second stage of the algorithm) the values of unknown variables are determined. The value of the variable x_{n-1} may be calculated from the last equation of the transformed system. After that it becomes possible to find the value of the variable x_{n-2} from the second to last equation etc.

Gaussian Elimination

The Gaussian elimination consists in sequential elimination of the unknowns in the equations of the linear equation system being solved. At iteration i, $0 \le i < n-1$, of the method the variable x_i is eliminated for all the equations with numbers k greater than i (i.e. $i < k \le n-1$,). In order to do so the row i multiplied by the constant (a_{ki}/a_{ii}) is subtracted from these equations so that the resulting coefficient of unknown x_i in the rows appears zero. All the necessary computations may be described by the following expressions:

$$\begin{aligned} a'_{kj} &= a_{kj} - (a_{ki} / a_{ii}) \cdot a_{ij}, \\ b'_k &= b_k - (a_{ki} / a_{ii}) \cdot b_i, \end{aligned} \quad i \le j \le n-1, i < k \le n-1, 0 \le i < n-1 \qquad (3.3)$$

(it should be noted that similar computations are performed over the vector b too).

Let us demonstrate the Gaussian elimination using the following system of linear equations as an example:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ 2x_0 + 7x_1 + 5x_2 &= 18. \\ x_0 + 4x_1 + 6x_2 &= 26 \end{aligned}$$

At the first iteration the unknown x_0 is eliminated in the second and the third rows. For this the first row multiplied correspondingly by 2 and by 1 is subtracted from these rows. After these transformations the system looks as follows:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 + x_2 &= 16. \\ x_1 + 4x_2 &= 25 \end{aligned}$$

As a result, we need to perform the last iteration and eliminate the unknown x_1 in the third equation. For this it is sufficient to subtract the second row. In the final form the system looks as follows:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 + x_2 &= 16. \\ 3x_2 &= 9 \end{aligned}$$

Figure 3.1 shows the general scheme of the state of data at i-th iteration of the Gaussian elimination. All the coefficients of the unknowns, which are located lower than the main diagonal and to the left of column i are already zero. At i-th iteration of the Gaussian elimination the coefficients of column i located lower than the main diagonal are set to zero. It is done by means of subtracting the row i multiplied by the adequate nonzero constant. After accomplishment of $(n-1)$ of such iterations the matrix, which defines the system of linear equations is transformed in the upper triangle form.

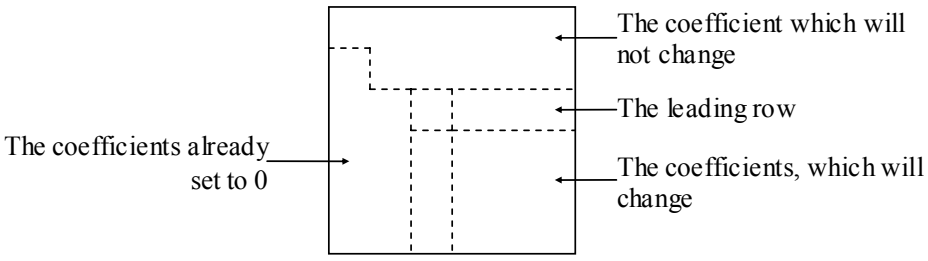

Figure 3.1. The Iteration of the Gaussian elimination

During the execution of the Gaussian elimination the row, which is used for eliminating unknowns is termed the *pivot* one, and the diagonal element of the pivot row is termed *the pivot element.* As it can be seen it is possible to perform computations only if the pivot element is a nonzero value. Moreover, if the pivot element $a_{i,i}$ has a small value, then the division and the multiplication of rows by this element may lead to accumulation of the computational errors and the computational instability of the algorithm.

A possible way to avoid this problem may consist in the following: at each next iteration of the Gaussian elimination it is necessary to determine the coefficient with the maximum absolute magnitude in the column, which corresponds to the eliminated variable, i.e.

$$y = \max_{i \le k \le n-1} |a_{ki}|,$$

and choose the row, which contains this coefficient, as the pivot one (this scheme of choosing the pivot value is termed *the method of partial pivoting*).

The computational complexity of the Gaussian elimination with the method of partial pivoting is of order $O(n^3)$.

Back Substitution

After the matrix of the coefficients has been reduced to the upper triangle form, it becomes possible to find the values of the unknowns. The value of the variable x_{n-1} may be calculated from the last equation of the transformed system. After that it is possible to determine the variable x_{n-2} from the second to last equation and so on. In the general form the computations performed during the back substitution may be presented by means of the following relations:

$$x_{n-1} = b_{n-1} / a_{n-1,n-1},$$
$$x_i = (b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j) / a_{ii}, \; i = n-2, n-3, ..., 0. \qquad (3.4)$$

It may be explained as previously using the example of the linear equation systems discussed in the previous subsection:

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 &= 1 \\ x_1 \quad + x_2 &= 16. \\ 3x_2 &= 9 \end{aligned}$$

From the last equation of the system, the value of the variable x_2 is 3. As a result, it becomes possible to solve the second equation and to find the value of the unknown x_1=13, i.e.

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 &= 1 \\ x_1 &= 13. \\ x_2 &= 3 \end{aligned}$$

The value of the unknown x_0, which is equal to -44, is determined at the last iteration of the back substitution.

With regard to the following parallel execution, it is possible to note that the accounting of the obtained unknown values may be performed in all the system equations at once (these operations may be performed in the equations simultaneously and independently). Thus, in the example under consideration the system after the determination of the value of the unknown x_2 may be reduced to the following form:

$$\begin{aligned} x_0 \quad +3x_1 &= -5 \\ x_1 &= 13 \\ x_2 &= 3 \end{aligned}$$

The computational complexity of the back substitution in the Gauss algorithm is $O(n^2)$.

Exercise 3 – The Realization of the Sequential Gauss Algorithm

In order to do the Exercise you should implement the Gauss algorithm of solving the linear equation systems. The initial variant of the program to be developed is given in the project *Serial Gauss*, which contains a certain part of the initial code and the necessary project parameters. While doing this Exercise it is necessary to add the input operations of initial matrix size, initial data setting, and the operations of the Gauss algorithm implementation and result output to the given variant of the program.

Task 1 – Open the Project SerialGauss

Open the project *SerialGauss* using the following steps:

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

After the project has been open in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *SerialGauss.cpp*, as it is shown in Figure 3.2. After that, the code, which is to be enhanced, will be opened in the Visual Studio workspace.

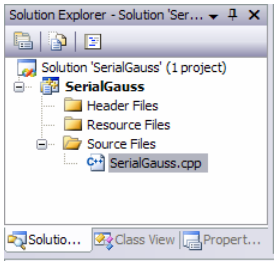

Figure 3.2. Opening the File SerialGauss.cpp

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

Let us consider the variables, which are used in the main function of the application. The first two of them (*pMatrix* and *pVector*) are correspondingly the matrix of the linear equation system and the vector of the right system parts. The third variable *pResult* is the vector, which should be obtained as a result of solving the linear equation system. The variable *Size* defines the size of the matrix and the vectors.

```
double* pMatrix;  // Matrix of the linear system
double* pVector;  // Right parts of the linear system
double* pResult;  // Result vector
int Size;         // Size of the matrix and the vector
```

As in previous labs, in order to store the matrix we should use a one-dimensional array, where the matrix is stored row by row. Thus, the element, located at the intersection of the *i*-th row and the *j*-th matrix column in a one-dimensional array, has the index *i*Size+j*.

The program code, which follows the variable declaration, is the initial message input and waiting for pressing any key before the application is terminated:

```
printf("Serial Gauss algorithm for solving linear systems\n");
getch();
```

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

Right after the program starts the following message will appear in the command console:

"`Serial Gauss algorithm for solving linear systems `".

Press any key to terminate the program execution.

Task 2 –Input the Matrix and Vector Sizes

In order to input the initial data of the Gauss algorithm of solving the linear equation systems, we will develop the function *ProcessInitialization*. This function is aimed at determining the matrix and vector sizes, allocating the memory for the initial matrix *pMatrix* and the vector *pVector*, and the result vector *pResult*. It is also used for setting the element values of the initial objects. 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 input the matrix size (to set the value of the variable *Size*). Add the bold marked 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 matrix and the vector
  printf("\nEnter the size of the matrix and the vector: ");
  scanf("%d", &Size);
  printf("\nChosen size = %d\n", Size);
}
```

The user can input the matrix size, which is read from the standard input stream *stdin* and stored in the integer variable *Size*. After that the value of the variable *Size* is printed (see Figure 3.3).

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

```
void main() {
  double* pMatrix;  // Matrix of the linear system
  double* pVector;  // Right parts of the linear system
  double* pResult;  // Result vector
  int Size;         // Size of the matrix and the vectors
  time_t start, finish;
  double duration;

  printf ("Serial Gauss algorithm for solving linear systems \n");
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

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

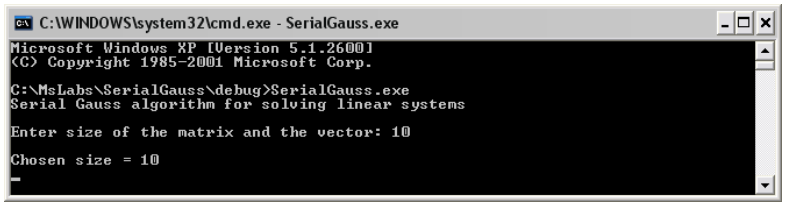

Figure 3.3. Setting the Matrix and the Vector Sizes

As in previous labs, we will check the input correctness. Let us check the size. If there is an error there (the size is either equal to zero or negative), we will continue to ask for the matrix size until a positive number is entered. To implement this behavior let us add the code, which inputs the matrix size in the loop:

```
  // Setting the size of the matrix and the vector
  do {
    printf("\nEnter the size of the matrix and the vector: ");
    scanf("%d", &Size);
    printf("\nChosen size = %d", Size);

    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);
```

Compile and run the application. Try to enter a nonpositive number as the matrix size. Make sure that the invalid situations are processed correctly.

Task 3 – Input the Initial Data

The function of the computation process initialization must also provide memory allocation for object storage (add the bold marked code to the function *ProcessInitialization*):

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
```

```
  double* &pCMatrix, int &Size) {
  // Setting the size of the matrix and the 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 matrix elements of the linear equation system *pMatrix* and the right part vector *pVector*. It should be noted that the matrix of the linear equation systems cannot be set arbitrarily. The solution for the linear equation system exists only in the case when the matrix of the linear equation system is non-degenerate (i.e. there is a reverse matrix for this matrix). It is not reasonable to check whether the matrix generated arbitrarily is non-degenerate (this operation is very "expensive"). That is why we will develop the function of generating the initial data set so that the matrix is originally non-degenerate. We will generate the lower triangle matrix, i.e. the matrix where all non-zero elements are located either on the main diagonal or lower than the diagonal. To set the element values of the matrix *pMatrix* and the vector *pVector* we will develop the function *DummyDataInitialization*. The heading and the implementation of the function are given below:

```
// Function for simple initialization of the matrix and the vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
    pVector[i] = i+1;
    for (j=0; j<Size; j++) {
      if (j <= i)
        pMatrix[i*Size+j] = 1;
      else
        pMatrix[i*Size+j] = 0;
     }
  }
}
```

As it can be seen from the given code, this function provides setting the matrix and the vector elements in rather a simple way: the values of all the elements of the matrix матрицы *pMatrix*, which are located above the main diagonal, are equal to 0, the rest of the elements are equal to 1. The vector *pVector* consists of sequential integer positive numbers from 1 to *Size*. So in case when the user chooses the object size equal to 4, the following matrix and the vector will be determined:

$$pMatrix = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \; pVector = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}.$$

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

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, int &Size) {
    <…>
  // Memory allocation
  <…>

  // Initialization of the matrix and the 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*. These functions were developed in Lab 1 and the code of the functions is available in the project (more detailed information on the functions *PrintMatrix* and *PrintVector* is given in Task 3, Exercise 2, Lab 1). Let us add these functions to print out the matrix *pMatrix* and the vector *pVector* to the main function:

```
  // Memory allocation and data initialization
  ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, 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 3.4). Run the application several times setting various matrix sizes.

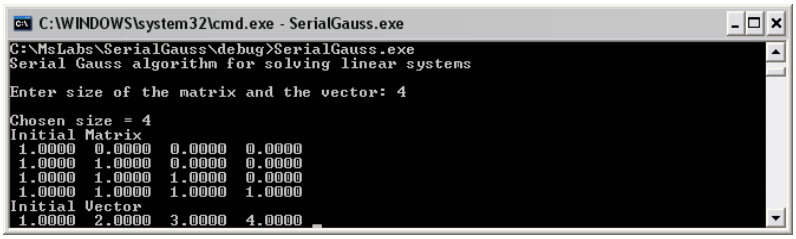

Figure 3.4. The Result of the Program Execution after the Completion of Task 3

It should be noted that if the matrix of the linear equation system and the vector of the right parts are set according to the above described rules, then the system has a simple solution, and all the result vector elements of the vector *pResult* must be equal to 1.

Let us develop one more function of generating the initial data. In this function we will also set the lower triangle matrix but the matrix elements and the elements of the right part vector will be determined by means of a random number generator (the random number generator is initiated by the current clock):

```
// Function for random initialization of the matrix and the 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++) {
      if (j <= i)
        pMatrix[i*Size+j] = rand()/double(1000);
      else
        pMatrix[i*Size+j] = 0;
    }
  }
}
```

Replace the call of the function for simple generation of the initial data *DummyDataInitialization* by the call of the function for random generation *RandomDataInitialization*. Compile and run the application. Make sure that the data is set according to the above described rules.

Task 4 –Terminate the Program Execution

Let us first develop the function for correct computation process termination, before working out the Gauss algorithm. For this purpose it is necessary to deallocate the memory, which has been dynamically allocated in the course of the program execution. Let us develop the corresponding function *ProcessTermination*. The memory has been allocated for storing the initial matrix *pMatrix* and the vector *pVector*, and also for storing the result vector of solving the linear equation system *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 immediately before the program termination:

```
  // 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);

  // Process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

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

Task 5 – Implement the Gaussian Elimination

Let us develop the main computational part of the program. In order to solve the linear equation system by means of the Gauss algorithm, we will implement the function *SerialResultCalculation*, which gets the initial matrix *pMatrix* and the vector *pVector*, the sizes of the objects *Size*, and the result vector *pResult* as input parameters.

According to the algorithm given in Exercise 2, the execution of the Gauss algorithm consists of the two stages: the Gaussian elimination and the back substitution. At the stage of executing the Gaussian elimination the linear equation system is reduced to the upper triangle form by means of equivalent transformations. In order to carry out this stage, we will develop the function *SerialGaussianElimination*. In the course of executing the back substitution, the result vector values are determined by means of reducing the matrix to the diagonal form. In order to carry out this stage, we will develop the function *SerialBackSubstitution*. Thus, the code of the function *SerialResultCalculation* should be the following:

```
// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);
}
```

In this task we will develop the Gaussian elimination. The back substitution will be executed in the next task of the lab.

The Gaussian elimination reduces the linear equation system matrix to the upper triangle form by means of equivalent transformation. At each iteration of the executed transformations we can use the method of partial pivoting for choosing the pivot row (see Exercise 2). According to this method the row, containing the element of the next matrix column, which is maximum in absolute magnitude, is chosen as the pivot one.

In order to store the order of choosing the pivot rows, we will use the array *pSerialPivotPos*. The *i*-th element of the array will store the number of the row, which was chosen as the pivot one in the course of executing the *i*-th iteration of the Gaussian elimination. Besides, we will use one more auxiliary array *pSerialPivotIter*. Each element of the array *pSerialPivotIter[i]* will store the number of the iteration , where the row with the number *i* was chosen as the pivot one. Originally we will fill the array *pSerialPivotIter* with the elements equal to -1 (i.e. the value -1 in the element of the array *pSerialPivotIter[i]* means that the row with the number *i* has not been chosen as the pivot one yet).

Let us declare the corresponding arrays as global variables, allocate the memory for these arrays before the beginning of executing the function *GaussianElimination*, deallocate the allocated memory after executing the back substitution of the Gauss method (the function *BackSubstitution*):

```
int* pSerialPivotPos;  // Number of pivot rows selected at the iterations
int* pSerialPivotIter; // Iterations, at which the rows were pivots
```

```
// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pSerialPivotPos  = new int [Size];
  pSerialPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pSerialPivotPos;
  delete [] pSerialPivotIter;
}
```

According to the computational scheme of the Gaussian elimination, it is necessary to determine the pivot matrix row at each iteration. The pivot row contains the column element being maximum in the absolute magnitude. This column number is equal to the number of the current iteration and the pivot row is selected among the rows, which have not been previously chosen as the pivot ones. The number of the pivot row is stored in the variable *Pivot* and is recorded in the corresponding element of the array *pSerialPivotPos*. Besides, the value of the element of the array *pSerialPivotIter*, which corresponds to the selected row, is set equal to the number of the current iteration.

Let us develop the function *FindPivotRow* in order to choose the pivot row. It is necessary to use the matrix of the linear equation system *pMatrix*, the matrix size *Size* and the number of the current iteration *Iter* as the arguments of the function. This function must look through the rows, which have not been previously selected as the pivot ones, choose among them the row, which contains the maximum element in the position *Iter*, and return the number of the selected row:

```
// Function for finding the pivot row
int FindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1;    // Index of the pivot row
  double MaxValue =  0; // Value of the pivot element
  int i;                // Loop variable

  // Choose the row, that stores the maximum element
  for (i=0; i<Size; i++) {
    if ((pSerialPivotIter[i] == -1) &&
       (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
      PivotRow = i;
      MaxValue = fabs(pMatrix[i*Size+Iter]);
    }
  }
  return PivotRow;
}
```

Let us add the call of the function *FindPivotRow* to the function, which carries out the Gaussian elimination. We will store the obtained value in corresponding element of the array *pPivotPos* and print the numbers of the selected pivot rows in order to check the computation correctness:

```
// Function for the Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;     // Number of the iteration of the Gaussian elimination
  int PivotRow; // Number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size, Iter);
    pSerialPivotPos[Iter] = PivotRow;
    pSerialPivotIter[PivotRow] = Iter;
  }
```

```
  printf ("Indices of the pivot rows: \n");
  for (int i=0; i<Size; i++)
    printf("%d ", pSerialPivotPos[i]);
}
```

Turn the call of the function, which executes the back substitution of the Gauss method, into comments in the function *SerialResultCalculation*. Add the call of the function *SerialResultCalculation* to the main function:

```
void main() {
  <…>
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // The Matrix and the vector output
  <…>
  // Execution of Gauss algorithm
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

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

Compile and run the application. Make sure that the pivot rows are chosen correctly. If you use the function *DummyDataInitialization,* the number of the pivot rows must coincide with the number of the iterations, at which the rows were selected. If the function *RandomDataInitialization* is used, the results of the print are shown in Figure 3.5 (to make it clearer the values of the pivot elements are marked by the red color).

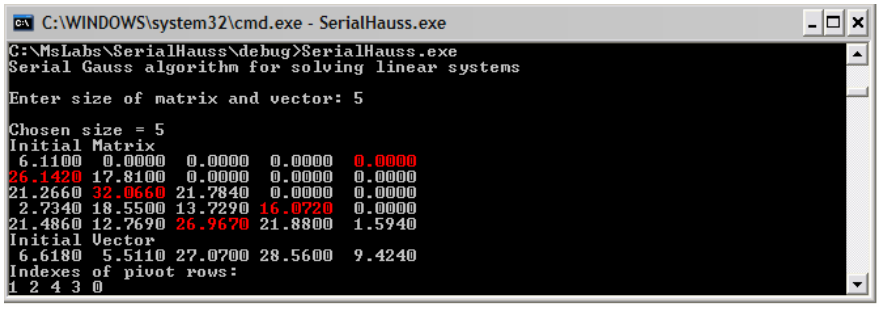

Figure 3.5. Selecting the Leading Pivot Rows

After selecting the pivot rows, these rows multiplied by the corresponding multipliers are subtracted from the rows, which have not yet been chosen as the pivot ones, and thus, the elements of the corresponding columns are zeroed. In order to carry out the subtraction we will develop the function *SerialEliminateColumns*. This function takes the matrix of the linear equation system *pMatrix*, the vector of the right part *pVector*, the number of the current pivot row *Pivot*, the number of the current iteration *Iter* and the size *Size* as the input arguments. For all the rows of the matrix *pMatrix* the function *EliminateRows* executes the following operations: it checks whether the given row has been chosen as the pivot one at one of the previous iterations using the values recorded in the array *pSerialPivotIter*; and if the result of the check is negative, the row undergoes the transformation according to formula (3.3):

```
// Function for the column elimination
void SerialColumnElimination (double* pMatrix, double* pVector, int Pivot,
  int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
```

```
}
```

Add the call of the function *SerialColumnElimination* to the code of the function, which executes the Gaussian elimination. Instead of printing the array *pPivotPos* print the matrix of the linear equation system *pMatrix*. It must be reduced to the upper triangle form (accurate to the row permutation, i.e. there must be a possibility to permute the matrix rows so as to obtain the upper triangle matrix) (see Figure 3.6).

```
// Function for the Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // Number of the iteration of the Gaussian elimination
  int PivotRow;   // Number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size,Iter);
    pSerialPivotPos[Iter] = PivotRow;
    pSerialPivotIter[PivotRow] = Iter;
    SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
  printf ("The matrix of the linear system after the elimination: \n");
  PrintMatrix(pMatrix, Size, Size);
}
```

Compile and run the application. Make sure that the Gaussian elimination is executed correctly.

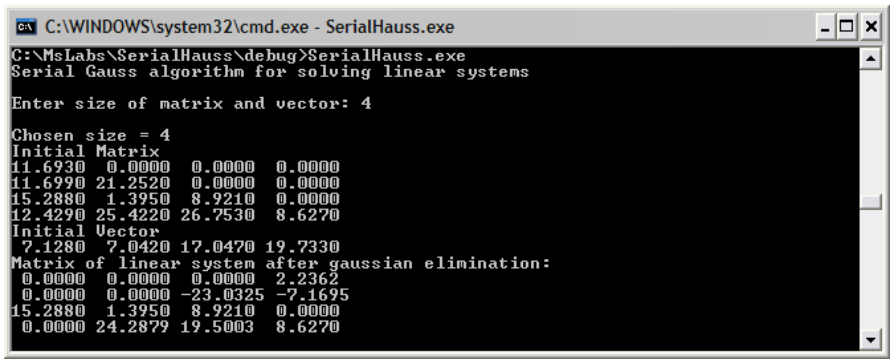

Figure 3.6. The Result of the Execution of the Gaussian Elimination

Task 6 – Implement the Back Substitution

To execute the back substitution of the Gauss algorithm we will develop the function *SerialBackSubstitution*. Let us use the matrix of the linear equation system *pMatrix*, the vector of the right parts *pVector*, the result vector *pResult* and the size *Size* as the function input arguments: :

```
// Function for the back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size);
```

The back substitution is described in detail in Exercise 2. The execution of the back substitution starts with the matrix row, which was chosen as the pivot one at the last iteration of the Gaussian elimination. You can find out the row number from the last element of the array *pSerialPivotPos* (on the analogy with this, the number of the row chosen as the pivot at the iteration next to the last, is stored in the next to the last element of the array *pSerialPivotPos* etc.). Using this row you may compute an element of the result vector. Then using the element it is possible to simplify the remaining matrix rows:

```
// Function for the back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[j] -= pMatrix[Row*Size+i]*pResult[i];
```

```
      pMatrix[Row*Size+i] = 0;
    }
  }
}
```

Eliminate the matrix print after the execution of the Gaussian elimination. In order to generate the initial data use the method *DummyDataInitialization* again. Restore the call of the function executing the back substitution (delete the comment signs in the call line). Call the print of the result vector after the execution of the Gauss algorithm in the main function:

```
void main() {
 <…>

  // The execution of Gauss algorithm
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

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

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

Compile and run the application. If the algorithm is implemented correctly, all the result vector elements must be equal to 1 (Figure 3.7).

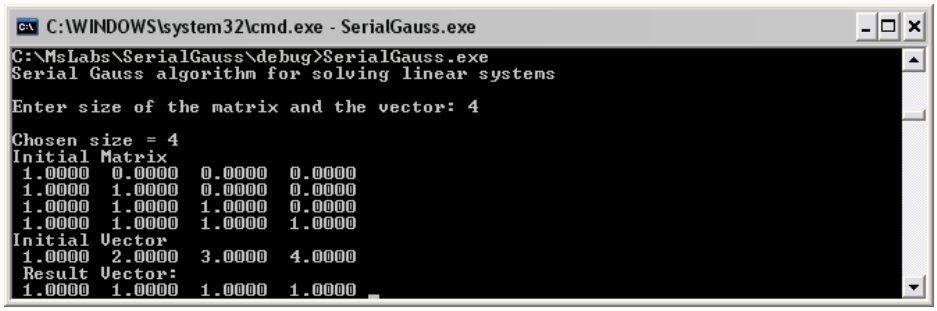

Figure 3.7. The Result of Executing the Gauss Algorithm

Task 7 – Carry out Computational Experiments

In order to test the speed up of the parallel calculations, 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 linear equation systems. We will set the elements of large matrices and vectors by means of the random data generator (the function *RandomDataInitialization*):

In order to determine the time, we will add the calls of the functions, which allow us to find out the computation execution time, to the obtained program. As previously, we will use the following function:

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

Let us add the computation and the output of the Gauss method execution time to the program code. For this purpose we will clock in before and after the call of the function *SerialResultCalculation*:

```
  // The execution of Gauss algorithm
  start = clock();
  SerialResultCalculation(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 execution time of Gauss method
printf("\n Time of execution: %f\n", duration);
```

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

Table 3.1. The Execution Time of the Serial Gauss Algorithm

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

The analysis of the computations executed in the Gauss method can demonstrate that the theoretical execution time for the serial Gauss algorithm may be calculated in accordance with the following expression (see Section 9 "Parallel methods of solving the linear equation systems")

$$T_1 = \left(2Size^3 / 3 + Size^2\right) \cdot \tau \qquad (3.5)$$

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

Let us fill out the table of comparison of the experiment execution time to the time, which may be obtained according to the formula (3.5). In order to compute the execution time of a single operation τ, we will apply the following technique: choose one of the experiments as a pivot. Then let us divide the execution time of a pivot experiment by the number of the executed operations (the number of the operations may be calculated using formula (3.5)). Thus, we will calculate the execution time of a basic computational operation. Then using this value we will calculate the theoretical execution time for the remaining experiments. It should be noted that the execution time of this basic operation depends generally speaking on the linear equation system size. That is why we should be oriented at a certain average case while choosing the pivot.

Compute the theoretical execution time for the Gauss algorithm. Give the results in the form of the table:

Table 3.2. The Comparison of the Experiment Execution Time of the Serial Gauss Algorithm to the Theoretically Calculated Time

Basic Computational Operation Execution Time τ (sec):			
Test Number	Matrix Size	Execution Time (sec)	Theoretical Time (sec)
1	10		
2	100		
3	500		
4	1,000		
5	1,500		
6	2,000		
7	2,500		
8	3,000		

Exercise 4 –Developing the Parallel Gauss Algorithm

Subtask Definition

In close consideration of the Gauss method it is possible to note that all the computations are reduced to the same computational operations on the rows of the coefficient matrix of the linear equation system. As a result, the data parallelism principle may be applied as the basis of the Gauss algorithm parallel implementation. All the

computations connected with processing a row of the matrix A and the corresponding element of the vector b may be taken as *the basic computational subtask* in this case.

Analysis of Information Dependencies

Let us consider the general scheme of parallel computations and the information dependencies among the basic subtasks, which appear in the course of computations.

For the execution of **the Gaussian elimination** stage it is necessary to perform *(n-1)* iterations of eliminating the unknown variables in order to transform the matrix A to the upper triangle form.

The execution of iteration i, $0 \le i < n-1$, of the Gaussian elimination includes a number of sequential operations. First of all, at the very beginning of the iteration it is necessary to select the pivot row, which (if the partial pivoting scheme is used) is determined by the search of the row with the maximum absolute value among the elements of the column i, which corresponds to the eliminated variable x_i. As the rows of matrix A are distributed among subtasks, the subtasks with the numbers k, $k > i$, should exchange their coefficients of the eliminated variable x_i for the maximum value search. After all the necessary data has been accumulated in each subtask, it is possible to determine, which of them holds the pivot row, and which value is the pivot element.

To carry out the computations further the pivot subtask has to broadcast its pivot row of the matrix A and the corresponding element of the vector b to all the other subtasks with the numbers k, $k > i$. After receiving the pivot row the subtasks perform the subtraction of rows, thus, providing the elimination of the corresponding variable x_i.

During the execution of **the back substitution** the subtasks perform the necessary computations for calculating the value of the unknowns. As soon as some subtask i, $0 \le i < n-1$, determines the value of its variable x_i, this value must be broadcasting to all the subtasks with the numbers k, $k < i$. After communications the subtasks substitute the variables x_i for the obtained value and modify the elements of the vector b.

Scaling and Subtask Distribution among the Processors

The basic subtasks are characterized by the same computational complexity and balanced amount of the transmitted data. In case when the size of the matrix, which describes the linear equation system appears to be greater than the number of the available processors (i.e. $p < n$), the basic subtasks may be enlarged by uniting several matrix rows in a subtask. Let us make use of the familiar scheme of block striped data partitioning: each process is allocated a continuous sequence of linear equation matrix rows.

The distribution of the subtasks among the processors must take into account the nature of the communication operations performed in the Gauss method. One-to-all broadcast is the main form of the information communication of the subtasks. As a result, the data transmission network topology must be a hypercube or a complete graph in order to implement the desired information communications among the basic subtasks efficiently.

Exercise 5 – Coding the Parallel Gauss Program for Solving the Linear Equation Systems

To do this Exercise you should develop the parallel Gauss program for solving the linear equation systems. This Exercise will help you:

- To get experience in developing the nontrivial parallel programs,
- To be familiar with collective data transmission operations in MPI.

Task 1 – Open the Project ParallelGauss

Open the project *ParallelGauss* using the following steps:

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

After the project has been open in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *ParallelGauss.cpp*, as it is shown in Figure 3.8. After that, the code, which is to be enhanced, will be opened in the Visual Studio workspace.

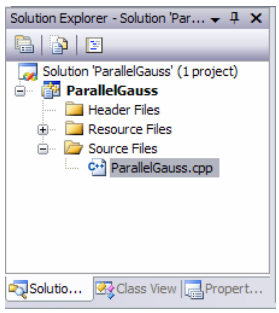

Figure 3.8. Opening the File ParallelGauss.cpp with the use of the Solution Explorer

The main function of the parallel application, which is located in the file *ParallelGauss.cpp*, contains the declaration of the necessary variables, the calls of the initialization functions and the MPI program environment execution termination, the functions for determining the number of the available processes and the process rank:

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

void main(int argc, char* argv[]) {
  double* pMatrix;        // Matrix of the linear system
  double* pVector;        // Right parts of the linear system
  double* pResult;        // Result vector
  int     Size;           // Size of the matrix and the vectors
  double Start, Finish, Duration;

  setvbuf(stdout, 0, _IONBF, 0);

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

  if (ProcRank == 0)
    printf("Parallel Gauss algorithm for solving linear systems\n");

  MPI_Finalize();
}
```

It should be noted that the variables *ProcNum* and *ProcRank* were declared global as in case of the previous labs.

The following functions and variables copied from the serial project, are also located in the file *ParallelGauss.cpp*: the global variable *pSerialPivotPos*, the functions *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *SerialGaussianElimination*, *SerialBackSubstitution*, *SerialColumnElimination*, *FindPivotRow* (the use of the functions and variables is described in detail in Exercise 3 of this lab). The first two functions will be used in the parallel application for data initializing. The other functions will provide the opportunity to execute the serial algorithm and to compare the results of executing the serial and the parallel Gauss algorithms.

In this parallel application we will use the functions *PrintMatrix* and *PrintVector* to print matrices and vectors. The implementation of the functions has also been copied to the parallel application. Besides, there are also preliminary versions for the initialization functions (*ProcessInitialization*) and the function of the process termination (*ProcessTermination*).

Compile and run the application using the Visual Studio. Make sure that the initial message is output on the command console:

"Parallel Gauss algorithm for solving linear equation systems".

Task 2 – Input the Initial Data

At the next stage of the development of the parallel application it is necessary to set the sizes of the linear equation system matrix, the right part vector, the result vector and to allocate memory for storing them. According to the parallel computation scheme the initial objects can exist only on the root process (the process with the rank 0). On each process at any given moment of time there is a stripe of the linear equation system

matrix, a block of the right part vector and a block of the result vector. Let us determine the variables for storing the blocks and the block sizes:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // Matrix of the linear system
  double* pVector;        // Right parts of the linear system
  double* pResult;        // Result vector
  double *pProcRows;      // Rows of the matrix A
  double *pProcVector;    // Block of the vector b
  double *pProcResult;    // Block of the vector x
  int     Size;           // Size of the matrix and vectors
  int     RowNum;         // Number of the matrix rows
  double Start, Finish, Duration;
```

In order to determine the matrix size and the vector size, to calculate the number of matrix rows, which will be processed by a given process, in order to allocate the memory for storing the matrix, the vectors and their blocks and also to generate the initial matrix and vector elements, we will develop the function *ProcessInitialization*.

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

In order to set the size *Size*, we will implement a dialog with the user as it has been done in the previous labs. The application to be developed in this Exercise is oriented at the most general case: it is not required that the size should be divisible by the number of the available processes. The only restriction is that the size *Size* should not be smaller than the number of the processes *ProcNum* so that each process has at least a row of the linear equation system matrix. If the user inputs an incorrect number, he is asked to repeat the input. The dialog is carried out only on *the root process.* When the sizes of the matrix and the vector are defined correctly, the value of the variable *Size* is broadcast to all the processes:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum) {
  if (ProcRank == 0) {
    do {
      printf("\nEnter the size of the matrix and the vector: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf ("Size must be greater than number of processes! \n");
      }
    } while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

After the size is set, it is possible to determine the number of the matrix rows, which will be processed by each process, and to allocate the memory for storing the matrix, the vector, the result matrix, the matrix stripe and the vector blocks. In order to determine the number of rows *RowNum*, which will be processed by a given process, let us use the method described in Lab 1 for the development of the parallel application of matrix-vector multiplication, if the matrix size is not divisible by the number of processes:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum) {
  <…>
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

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

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

  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
  }
}
```

In order to determine the elements of the linear equation system matrix *pMatrix* and the right part vector *pVector* we will use the function *DummyDataInitialization*, which was developed in the course of the implementation of the serial Gauss algorithm:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcVector,
  double* &pProcResult, int &Size, int &RowNum) {
  <…>
  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    pVector = new double [Size];
    pResult = new double [Size];
    // Initialization of the matrix and the vector elements
    DummyDataInitialization (pMatrix, pVector, Size);
  }
}
```

Let us call the function *ProcessInitialization* from the main function of the parallel application. In order to control the correctness of the initial data input, we will use the function of the formatted matrix output *PrintMatrix* and the vector *PrintVector*, let us print out the linear equation system matrix and the right part vector on the root process.

```
void main(int argc, char* argv[]) {
  <…>
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcVector,
    pProcResult, Size, RowNum);
  if (ProcRank == 0) {
    printf("Initial matrix \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial vector \n");
    PrintVector(pVector, Size);
  }
  MPI_Finalize();
}
```

Compile and run the application. Make sure that the dialog for the input of the size makes possible to enter only the correct size value. Analyze the values of the matrix and the vector. If the data is set correctly, the linear system matrix must be lower and triangle, all the elements located lower than the main diagonal must be equal to 1, the right part vector elements must be integer positive numbers from 1 to *Size*. (Figure 3.9).

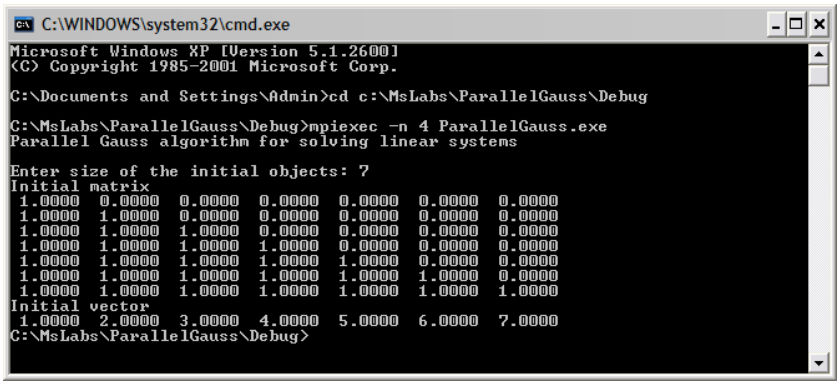

Figure 3.9. The Test Values of The Initial Data

Task 3 – Terminate the Parallel Program

In order to terminate the application at each stage of development, we should develop the function of correct termination. For this purpose we should deallocate the memory, which has been allocated dynamically in the course of the program execution. Let us develop the corresponding function *ProcessTermination*. The memory for storing the matrix *pMatrix*, the vector *pVector* and the result vector *pResult*, was allocated on the root process; besides, memory was allocated on all the processes for storing the stripe of the matrix *pProcRows,* the blocks of the right part vector *pProcVector* and the result vector *pProcResult*. 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* pProcVector, double* pProcResult) {
  if (ProcRank == 0) {
    delete [] pMatrix;
    delete [] pVector;
    delete [] pResult;
  }
  delete [] pProcRows;
  delete [] pProcVector;
  delete [] pProcResult;
}
```

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

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

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

Task 4 – Distribute the Data among the Processes

In accordance with the parallel computation scheme, described in the previous Exercise, the system of linear equations must be distributed among the processes in horizontal stripes (divided into continuous sequences of rows).

The function *DataDistribution* is responsible for data distribution. The matrix *pMatrix* and the vector *pVector*, the horizontal stripe of the matrix *pProcRows*, and the corresponding block of the right part vector *pProcVector*, and also the object sizes (the matrix size and the vector size *Size* and the number of rows in the horizontal stripe *RowNum*) must be given to the function as arguments:

```
// Function for the data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  double* pProcVector, int Size, int RowNum);
```

In order to distribute the matrix into horizontal stripes and broadcast these stripes, we will use the procedure described in Lab 1 for the development of the parallel application of matrix vector multiplication in case when the matrix size is not divisible by the number of processes.

```
// Function for the data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  double* pProcVector, int Size, int RowNum) {

  int *pSendNum;     // Number of the elements sent to the process
  int *pSendInd;     // Index of the first data element sent to the process
  int RestRows=Size; // Number of rows, that have not been distributed yet
  int i;             // Loop variable

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

```
  pSendNum = new int [ProcNum];

  // Define the disposition of the matrix rows for the current process
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (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;
}
```

To partition the vector we will use the same sequence of actions. We will make only slight changes: in case of the parallel Gauss algorithm execution we should be able to use the row number and to determine, on which of the process the row is located and what number it has in the process stripe. In order to solve the problem efficiently we will arrange two global arrays: *pProcInd* and *pProcNum*. There must be *ProcNum* elements in each of the arrays. The element of the first array *pProcInd*[*i*] determines the number of the first row located on the process with rank *i*. The element of the second array *pProcNum*[*i*] determines the number of the linear system rows, which are processed by the process with rank *i*. Let us declare the corresponding global variables, allocate the memory for the arrays in the function *DataDistribution*, fill the arrays with values. It should be noted that the arrays may be used for scattering the right part vector by means of the function *MPI_Scatter*.

```
int* pProcInd; // Number of the first row located on the processes
int* pProcNum; // Number of the linear system rows located on the processes

// Function for the data distribution among the processes
void DataDistribution(double* pMatrix, double* pProcRows, double* pVector,
  double* pProcVector, int Size, int RowNum) {
  <…>
  // Alloc memory for temporary objects
  pProcInd = new int [ProcNum];
  pProcNum = new int [ProcNum];
  <…>
  // Free the memory
  delete [] pSendNum;
  delete [] pSendInd;
}
```

Correspondingly, it is necessary to call the data distribution function from the main program immediately after the call of the computational process initialization, before starting the execution of the Gauss algorithm:

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

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

Let us delete the initial print after the execution of the function *ProcessInitialization*. We will check the correctness of the data distribution among the processes up. For this purpose we should print the matrix and vector, and then the matrix stripes and the vector blocks located on each of the processes after the execution of the function *DataDistribution.* Let us add to the application code one more function, which serves for checking the correctness of the data distribution stage. This function will be referred to as *TestDistribution*.

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

```
// Function for testing the data distribution
void TestDistribution(double* pMatrix, double* pVector, double* pProcRows,
  double* pProcVector, int Size, int RowNum) {
  if (ProcRank == 0) {
    printf("Initial Matrix: \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector: \n");
    PrintVector(pVector, Size);
  }
  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(pProcVector, RowNum);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

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

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

Compile the application. If you find errors in the process of compiling, correct them, comparing your code to the code given in the lab. Run the application. Make sure that the data is distributed correctly (Figure 3.10):

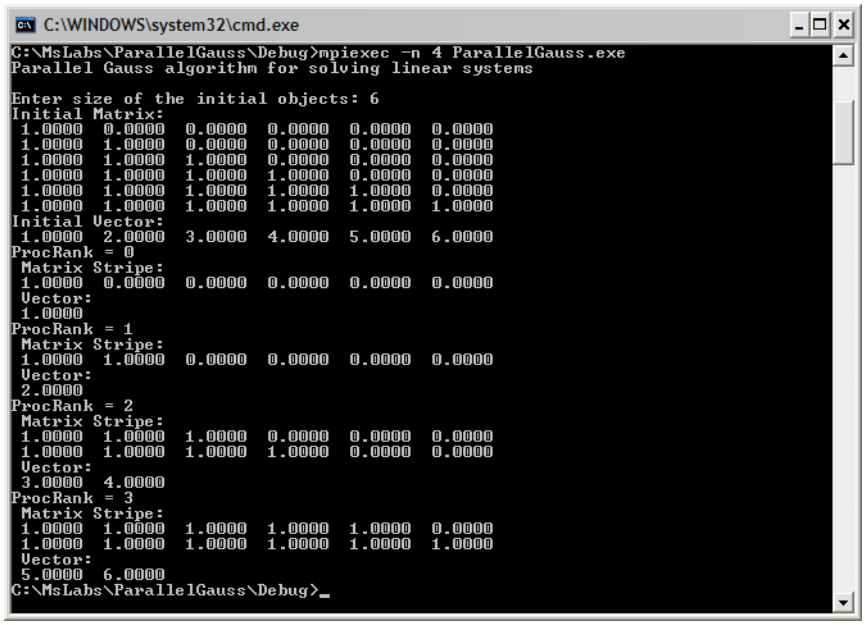

Figure 3.10. Data Distribution for the Cases when the Application Is Run Using Four Processes and the Size of the Equation System is Equal to Six

Task 5 – Implement the Gaussian Elimination

According to the computational scheme of the Gauss algorithm for solving the linear equation systems, the method consists of the two stages: the Gaussian elimination and the back substitution. In order to execute the parallel Gauss algorithm we will develop the function *ParallelResultCalculation*, which contains the calls of the functions for executing the Gauss algorithm stages:

```
// Function for execution of the parallel Gauss algorithm
void ParallelResultCalculation(double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  // Gaussian elimination
```

```
  ParallelGaussianElimination (pProcRows, pProcVector, Size, RowNum);
  // Back substitution
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult, Size,
    RowNum);
}
```

In order to develop the parallel version of Gauss algorithm we will need two auxiliary arrays *pParallelPivotPos* and *pProcPivotIter*.

The elements of the array *pParallelPivotPos* define the numbers of the matrix rows selected as the pivot ones at the iterations of the Gauss elimination. The back substitution iterations for finding the values of the unknown linear equation systems must be executed in exactly this order. The array *pParallelPivotPos* is global and any change in any of its processes requires executing the operation of broadcasting the data to the other program processes.

The elements of the array *pProcPivotIter* determine the number of iterations for the Gaussian elimination. At these iterations the process rows were used as the pivot ones (i.e. the row *i* of the process was chosen the pivot one at the iteration *pProcPivotIter[i]*). The original value of the array elements is set equal to -1 and , thus, this element value of the array *pProcPivotIter[i]* signifies that the row *i* of the process is still to be processed. Besides, it is important to note that the iteration numbers stored in the elements of the array *pProcPivotIter* mean the numbers of the unknowns, which must be determined with the help of the corresponding equation rows. The array *pProcPivotIter* is local for each process.

Let us declare the corresponding global variables:

```
int *pParallelPivotPos; // Number of rows selected as the pivot ones
int *pProcPivotIter;    // Number of iterations, at which the process
                        // rows were used as the pivot ones
```

Let us allocate the memory for storing these objects before the execution of the parallel Gauss method stages. After the termination of the back substitution we will deallocate the memory:

```
// Function for the execution of the parallel Gauss algorithm
 void ParallelResultCalculation(double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {

  // Memory allocation
  pParallelPivotPos = new int [Size];
  pProcPivotIter = new int [RowNum];
  for (int i=0; i<RowNum; i++)
    pProcPivotIter[i] = -1;

  // Gaussian elimination
  ParallelGaussianElimination (pProcRows, pProcVector, Size, RowNum);
  // Back substitution
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult, Size,
    RowNum);

  // Memory deallocation
  delete [] pParallelPivotPos;
  delete [] pProcPivotIter;
}
```

Later in this Exercise we will develop the Gaussian elimination. The back substitution of the Gauss method will be implemented in the next task of the lab.

So the function *ParallelGaussianElimination* is intended for the execution of the Gaussian elimination in parallel. The matrix stripe of the linear equation system, which processes a given process (*pProcRows*), and a block of the right part vector *pProcVector*, the size *Size* and the number of rows in the stripe *RowNum*, have to be given to the function as arguments:

```
// Function for the Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum);
```

The purpose of the function is to reduce the matrix of the linear equation system to the upper triangle form using equivalent transformations.

The number of the Gaussian elimination is equal to the order of the linear equation system. At each iteration the pivot row is selected with the help of the method of partial pivoting. As the matrix rows of the linear

equation systems are distributed among the subtasks, in order to find the maximum value the subtasks must exchange their elements of the column with the eliminated variable (at the iteration *i* of the Gaussian elimination the *i*-th unknown is eliminated). After gathering all the necessary data in each subtask it is possible to determine, which of the subtasks contains the pivot row and which value is the pivot element.

Let us develop the procedure of choosing the pivot row in two stages. At the first stage the local pivot rows are selected on each process. For this purpose it is necessary to look through the rows to be processed (the row with the number *i* should be processed if the value of the element *pProcPivotIter[i]* is equal to -1), and select the row, which contains the maximum in absolute magnitude coefficient of eliminated unknown variable:

```
// Function for the Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;   // Value of the pivot element of the process
  int    PivotPos;   // Position of the pivot row in the process stripe

  // The iterations of the Gaussian elimination
  for (int i=0; i<Size; i++) {
    // Calculating the local pivot row
    for (int j=0; j<RowNum; j++) {
      if ((pProcPivotIter[j] == -1) &&
          (MaxValue < fabs(pProcRows[j*Size+i]))) {
        MaxValue = fabs(pProcRows[j*Size+i]);
        PivotPos = j;
      }
    }
  }
}
```

After calculating the pivot row on each process, we should choose the maximum element among the obtained pivot elements and determine, at which process it is located. The library MPI provides the function *MPI_Allreduce* for carrying out these operations. The function has the following heading:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,int count,MPI_Datatype type,
  MPI_Op op,MPI_Comm comm),
where
 - sendbuf – the memory buffer with the sent message,
 - recvbuf – the memory buffer for the result message,
 - count   - the number of elements in the messages,
 - type    - the type of the message elements,
 - op      - the operation to be executed with the data,
 - comm    - the communicator, where the operation is executed.
```

Let us reduce the data in order to determine the value of the pivot element and the process, where the pivot row is located:

```
// Function for the Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;  // Value of the pivot element of the process
  int PivotPos;     // Position of the pivot row in the process stripe

  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
  // The Iterations of the Gaussian elimination
  for (int i=0; i<Size; i++) {
    <…>
    // Finding the global pivot row
    ProcPivot.MaxValue = MaxValue;
    ProcPivot.ProcRank = ProcRank;
    // Finding the pivot process
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
      MPI_COMM_WORLD);
  }
}
```

After the execution of the data reduction operation the value of the pivot element and the number of the process where the corresponding pivot row is located will be stored in the variable *Pivot*.

Let us fill the corresponding element of the array *pProcPivotIter* on the process where the pivot row is located. Besides, let us place the number of the pivot row to the global array *pPrallelPivotPos* (we know the number of the process where the pivot row is located and the row number in the stripe, which is located on the process; this data allows us to determine the number of the row in the equation system using the values in the arrays *pProcInd* and *pProcNum*).

```
  // Function for the Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;  // Value of the pivot element of the process
  int PivotPos;     // Position of the pivot row in the process stripe

  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
  // The Iterations of the Gaussian elimination stage
  for (int i=0; i<Size; i++) {
    <…>
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
      MPI_COMM_WORLD);
    // Storing the number of the pivot row
    if ( ProcRank == Pivot.ProcRank ){
      pProcPivotIter[PivotPos]= i;
      pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos;
    }
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
      MPI_COMM_WORLD);
  }
}
```

In order to carry out the transformation of the remaining matrix rows it is necessary to broadcast the pivot row and the corresponding element of the right part vector to all the processes. Let us have a buffer for storing the pivot row on the process, the rank of which was determined in the course of reduction (*Pivot.ProcRank*). Let us copy the row into the buffer and execute the broadcast:

```
// Function for the Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  double MaxValue;  // Value of the pivot element of the process
  int PivotPos;     // Position of the pivot row in the process stripe

  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
  double *pPivotRow;   // Pivot row of the current iteration
  pPivotRow = new double [Size+1];
  // The iterations of the Gaussian elimination stage
  for (int i=0; i<Size; i++) {
    <…>
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
      MPI_COMM_WORLD);
    // Broadcasting the pivot row
    if ( ProcRank == Pivot.ProcRank ){
      // Fill the pivot row
      for (int j=0; j<Size; j++) {
        pPivotRow[j] = pProcRows[PivotPos*Size + j];
      }
      pPivotRow[Size] = pProcVector[PivotPos];
    }
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
      MPI_COMM_WORLD);
  }
  delete [] pPivotRow;
}
```

After obtaining the pivot row the subtasks carry out the subtraction of rows, thus, providing the elimination of the corresponding unknown. Let us implement the subtraction with the help of the function *ParallelEliminateRows*:

```
// Fuction for the column elimination
void ParallelEliminateColumns(double* pProcRows, double* pProcVector,
  double* pPivotRow, int Size, int RowNum, int Iter) {
  double PivotFactor;
  for (int i=0; i<RowNum; i++) {
    if (pProcPivotIter[i] == -1) {
      PivotFactor = pProcRows[i*Size+Iter] / pPivotRow[Iter];
      for (int j=Iter; j<Size; j++) {
        pProcRows[i*Size + j] -= PivotFactor* pPivotRow[j];
      }
      pProcVector[i] -= PivotFactor * pPivotRow[Size];
    }
  }
}
```

Let us call the function of subtraction from the function, which executes the parallel algorithm of the Gaussian elimination:

```
// Function for the Gaussian elimination
void ParallelGaussianElimination (double* pProcRows, double* pProcVector,
  int Size, int RowNum) {
  <…>
  for (int i=0; i<Size; i++) {
    <…>
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
      MPI_COMM_WORLD);
    // Column elimination
    ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow, Size,
      RowNum, i);
  }
  delete [] pPivotRow;
}
```

Delete the call of the function testing the data distribution stage. Transform the call of the function, which executes the back substitution of the Gauss method *ParallelBackSubstitution*, into the comment. To check the correctness of executing the Gaussian elimination, call the function *TestDistribution* immediately after *ParallelResultCalculation*:

```
  // The execution of the parallel Gauss algorithm
  ParallelResultCalculation (pProcRows, pProcVector, pProcResult Size,
    RowNum);
  TestDistribution(pMatrix, pVector, pProcRows, pProcVector, Size, RowNum);
```

Compile and run the application. It should be noted that after the execution of the Gaussian elimination the matrix must be reduced to the upper triangle form. Run the application.

Make sure that the developed functions are operated correctly (Figure 3.11).

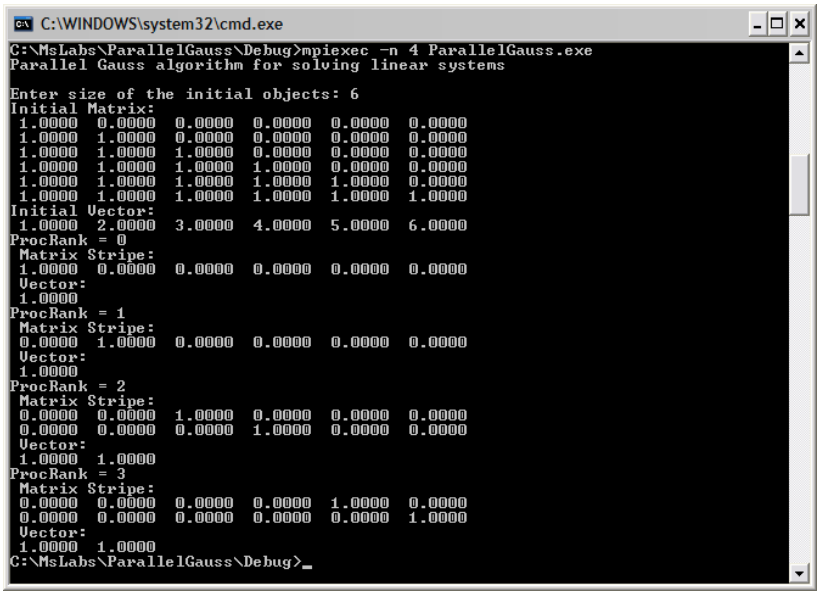

Figure 3.11. The Result of the Execution of the Gaussian elimination

Task 7 – Implement the Back Substitution

In the course of the execution of the back substitution the processes carry out the calculations necessary for obtaining the values of the unknown variables. As soon as any process determines the value of its variable, this variable must be broadcast to all the processes so that they can substitute the obtained value of the new unknown variable and correct the values for the elements of the right part vector.

The back substitution execution consists of *Size* iterations. At each iteration it is necessary to determine the row, which makes possible to calculate the value of the next result vector element. The row number is stored in the array *pParallelPivotIter*. Using the row number you should determine the number of process, where the row is stored, and the number of the row in the stripe *pProcRows* of the process. Let us develop the function *FindBackPivotRow* in order to carry out these operations:

```
// Function for finding the pivot row of the back substitution
void FindBackPivotRow(int RowIndex, int &IterProcRank,
  int &IterPivotPos) {
  for (int i=0; i<ProcNum-1; i++) {
    if ((pProcInd[i]<=RowIndex) && (RowIndex<pProcInd[i+1]))
      IterProcRank = i;
  }
  if (RowIndex >= pProcInd[ProcNum-1])
    IterProcRank = ProcNum-1;
  IterPivotPos = RowIndex - pProcInd[IterProcRank];
}
```

The number *RowIndex* of the row, for which we determine the location is given to the function as the argument. The function writes the rank of the process, where the row index is located, to the variable *IterProcRank,* and the number of the row in the buffer *pProcRows* - to the variable *IterPivotPos*.

After the location of the row has been determined, the process, which contains the row, calculates the value of the corresponding result vector element and broadcasts it to all the processes. Then the processes carry out the transformation of their matrix rows:

```
// Function for the back substitution
void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  int IterProcRank;    // Rank of the process with the current pivot row
  int IterPivotPos;    // Position of the pivot row of the process
  double IterResult;   // Calculated value of the current unknown
  double val;

  // The iterations of the back substitution
  for (int i=Size-1; i>=0; i--) {
```

```
    // Calculating the rank of the process, which holds the pivot row
    FindBackPivotRow(pParallelPivotPos[i],Size,IterProcRank,IterPivotPos);

    // Calculating the unknown
    if (ProcRank == IterProcRank) {
      IterResult = pProcVector[IterPivotPos] /
                   pProcRows[IterPivotPos*Size+i];
      pProcResult[IterPivotPos] = IterResult;
    }
    // Broadcasting the value of the current unknown
    MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);

    // Updating the values of the vector
    for (int j=0; j<RowNum; j++)
      if ( pProcPivotIter[j] < i ) {
        val = pProcRows[j*Size + i] * IterResult;
        pProcVector[j]=pProcVector[j] - val;
    }
  }
}
```

Restore the call of the function, which executes the back substitution of the Gauss algorithm (remove the comment signs in the call line). After the execution of the parallel Gauss algorithm, print the result vector blocks on each parallel process. Compile and run the application. Test the correctness of the program execution: if the function *DummyDataInitialization* is used to generate the initial data, all the result vector elements must be equal to 1.

Task 8 – Gather the Result

After the execution of the back substitution of the Gauss algorithm the result vector blocks are located on each process. It is necessary to collect the result vector on the root process. Let us execute the result gather by means of the function *MPI_Gatherv*. The arrays necessary for calling the function were already determined in the course of executing the function *DataDistribution*. Thus, the function performing the gather has a very simple implementation:

```
// Function for gathering the result vector
void ResultCollection(double* pProcResult, double* pResult) {
  //Gathering the result vector on the pivot processor
  MPI_Gatherv(pProcResult, pProcNum[ProcRank], MPI_DOUBLE, pResult,
    pProcNum, pProcInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Add the call of the function for gathering the result vector into the main function of the parallel application.

```
// The execution of the parallel Gauss algorithm
  ParallelResultCalculation(pProcRows, pProcVector, pProcResult,
    Size, RowNum);

  // Gathering the result vector
  ResultCollection(pProcResult, pResult);
```

It should be noticed, that the order of the unknowns in *pResult* vector is the same with the order of pivot rows selection, that was carried out during the Gaussian elimination stage. So, this order is stored in the *pParallelPivotPos* array. This order should be taken into account while printing the result vector. Let's develop the function *PrintResultVector* for formatted result vector output.

```
// Function for formatted result vector output
void PrintResultVector (double* pResult, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pResult[pParallelPivotPos[i]]);
}
```

Print the result vector on the root process with the help of *PrintResultVector* function:

```
  // Gathering the result vector
  ResultCollection(pProcResult, pResult);
```

```
  if (ProcRank == 0) {
    printf ("Result vector \n");
    PrintResultVector(pResult, Size);
  }
```

Compile and run the application. Check the correctness of the algorithm execution: if the Gauss method is implemented correctly, all the result vector elements must be equal to 1 (if the function *DummyDataInitialization* is used to generate the initial data).

Task 9 – Test the Parallel Program Correctness

After the function of the result collection is developed, it is necessary to check the correctness of the program execution. Let us develop the function *TestResult* for this purpose. It will perfom the multiplication of the linear system matrix by the vector of unknowns, that has been obtained by the means of Gauss method. Remember, that the order of the unknowns is stored in the *pParallelPivotPos* array. The result of the multiplication will be stored in the variable *pRightPartVector*. Then, the function will compare the vector of right parts *pVector* and the result of multiplication *pRightPartVector* element by element. In order to obtain each element of the result vector, it is necessary to execute serial multiplication and summation of real numbers. The order of executing these operations can influence the machine inaccuracy of computations and its value. That is why it is impossible to check of the vector elements are identical or not. Let us introduce the allowed divergence value of the serial and parallel algorithm results – the value *Accuracy*. The vectors are assumed to be the same if the corresponding elements differ by no more than the value of the allowed error *Accuracy*.

The function *TestResult* must have access to the linear equation system matrix *pMatrix* and the right part vector *pVector*. Consequently, it may be executed only on the root process:

```
// Function for testing the result
void TestResult(double* pMatrix, double* pVector, double* pResult,
  int Size) {
  /* Buffer for storing the vector, that is a result of multiplication
     of the linear system matrix by the vector of unknowns */
  double* pRightPartVector;
  // Flag, that shows wheather the right parts vectors are identical or not
  int equal = 0;
  double Accuracy = 1.e-6; // Comparison accuracy

  if (ProcRank == 0) {
    pRightPartVector = new double [Size];
      for (int i=0; i<Size; i++) {
        pRightPartVector[i] = 0;
        for (int j=0; j<Size; j++) {
          pRightPartVector[i] +=
             pMatrix[i*Size+j]*pResult[pParallelPivotPos[j]];
        }
      }

    for (int i=0; i<Size; i++) {
        if (fabs(pRightPartVector[i]-pVector[i]) > Accuracy))
        equal = 1;
    }
    if (equal == 1)
      printf("The result of the parallel Gauss algorithm is NOT correct."
             "Check your code.");
    else
      printf("The result of the parallel Gauss algorithm is correct.");
      delete [] pRightPartVector;
  }
}
```

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

Transform the calls of the functions into comment, using the debugging print, which have been previously used for checking the correctness of parallel application execution. Instead of the function *DummyDataInitialization*, which generates the linear equation system of a simple type, call the function *RandomDataInitialization*, which generates the system of equations with the lower triangle matrix, where nonzero elements are set by means of the random data generator. Compile and run the application. Set various amounts of the initial data. Make sure that the application is functioning properly.

Task 10 – Carry out the Computational Experiments

Let us determine the parallel algorithm execution time. For this purpose add clocking to the program code. As the parallel algorithm 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 *ResultCollection*:

```
<…>
Start = MPI_Wtime();

// Distributing the initial data between the processes
DataDistribution(pMatrix, pProcRows, pVector, pProcVector, Size, RowNum);
// The execution of the parallel Gauss algorithm
ParallelResultCalculation(pProcRows, pProcVector, pProcResult,
  Size, RowNum);
// Gathering the result vector
ResultCollection(pProcResult, pResult, Size, RowNum);

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

// Testing the result
TestResult(pMatrix, pVector, pResult, Size);

// Printing the time spent by parallel Gauss algorithm
if (ProcRank == 0)
  printf("\n Time of execution: %f\n", Duration);
```

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

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

Table 3.3. The Execution Time of the Parallel Gauss Algorithm for Solving the Linear Equation Systems and the Speed Up

Test Number	System Size	Serial Algorithm	Parallel Algorithm					
			2 processors		4 processors		8 processors	
			Time	Speed Up	Time	Speed Up	Time	Speed Up
1	10							
2	100							
3	500							
4	1,000							
5	1,500							
6	2,000							
7	2,500							
8	3,000							

Give the serial algorithm execution time in the column “Serial algorithm”. The time must be measured in the course of testing the serial application in Exercise 3. In order to calculate the speed up, divide the serial program execution time by the parallel program execution time. Place the results in the corresponding column of the table.

In order to estimate the theoretical execution time of the parallel algorithm implemented according to the computational scheme, which was given in Exercise 4, you might use the following expression:

$$T_p = \frac{1}{p}\sum_{i=2}^{n}(3i + 2i^2)\,\tau + (n-1)\cdot \log_2 p \cdot (3\alpha + w(n+2)/\beta) \qquad (3.6)$$

(the detailed derivation of the formula is given in Section 9 “Parallel methods of solving the linear equation systems” of the training material). Here n is the linear equation system size, p is the number of processes, 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 performing 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 (3.6). Tabulate the results in the following way (Table 3.4):

Table 3.4. The Comparison of the Experiment Parallel Execution Time to the Theoretically Calculated Execution Time

Test number	System Size	2 processors		4 processors		8 processors	
		Model	Experiment	Model	Experiment	Model	Experiment
1	10						
2	100						
3	500						
4	1,000						
5	1,500						
6	2,000						
7	2,500						
8	3,000						

Discussions

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

Exercises

1. Study the conjugate gradient method of solving the linear equation systems. Develop the serial and the parallel variants of the method.

Appendix 1. The Program Code of the Serial Gauss Algorithm for Solving the Linear Equation Systems

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

int* pSerialPivotPos;  // Number of pivot rows selected at the iterations
int* pSerialPivotIter; // Iterations, at which the rows were pivots

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

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