

Learning Lab 6: Parallel Algorithms of Solving Differential Equations in Partial Derivatives

Learning Lab 6: Parallel Algorithms of Solving Differential Equations in Partial Derivatives	1
Lab Objective	1
Exercise 1 – Stating the Dirichlet Problem.....	2
Exercise 2 – Code the Serial Gauss-Seidel Program for Solving the Dirichlet Problem	3
Task 1 – Open the Project SerialGaussSeidel	3
Task 2 – Input the Initial Data.....	4
Task 3 – Set the Initial Data	6
Task 4 – Terminate the Program Execution	7
Task 5 – Implement the Gauss-Seidel Algorithm	7
Task 6 – Carry out the Computational Experiments.....	9
Task 3 – Develop the Parallel Gauss-Seidel Algorithm	10
Subtask Definition.....	10
Analysis of Information Dependencies	10
Scaling and Distributing the Subtasks among the Processors.....	12
Exercise 4 – Code the Parallel Gauss-Seidel Program for Solving the Dirichlet Problem	12
Task 1 – Open the Project ParallelGaussSeidel	12
Task 2 – Initialize and Terminate the Parallel Program.....	13
Task 3 – Input the Initial Data.....	14
Task 4 – Terminate the Calculations	16
Task 5 – Distribute the Data among the Processes	16
Task 6 – Exchange the Boundary Rows among the Neighboring Processes.....	18
Task 7 – Implement the Parallel Algorithm Iterations.....	19
Task 8 – Calculating the Maximum Result Deviation	21
Task 9 – Gather the Results	22
Task 10 – Test the Parallel Program Correctness	22
Task 11 – Implement the Gauss-Seidel Algorithm for Any Given Grid	24
Task 12 – Carry out the Computational Experiments.....	26
Discussions	27
Exercises.....	27
Appendix 1. The Program Code of the Serial Gauss-Seidel Algorithm	27
Appendix 2 – The Program Code of the Parallel Gauss-Seidel Algorithm	29

Partial differential equations are widely used in various scientific and technical fields. Unfortunately, analytical solution of these equations may only be possible in simple special cases. Therefore approximate numerical methods are usually used for solving partial differential equations. The amount of computations to perform here is usually significant. The use of high-performance systems is traditional for this sphere of computational mathematics. Numerical solution of differential equations in partial derivatives is a subject of intensive research.

Lab Objective

The objective of this lab is to develop a parallel program, which provides the solution to one of the problems described by differential equations in partial derivatives, i.e. the Dirichlet problem for the Poisson equation. The lab assignments include:

- Exercise 1 – Stating the Dirichlet problem.
- Exercise 2 – Code the serial Gauss-Seidel program for solving the Dirichlet problem.
- Exercise 3 – Develop the parallel Gauss-Seidel algorithm.
- Exercise 4 – Code the parallel Gauss-Seidel program for solving the Dirichlet problem.

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 12 “Parallel methods of solving differential equations in partial derivatives”. Besides, the preliminary lab “Parallel programming with MPI” is assumed to have been done.

Exercise 1 – Stating the Dirichlet Problem

In this lab we will consider the numerical solution of the Dirichlet problem for the Poisson equation. It is defined as a problem of finding the function $u = u(x, y)$ that satisfies the following equation in the domain D :

$$\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = f(x, y), \quad (x, y) \in D,$$

and takes the value $g(x, y)$ at the boundary D^0 of the domain D (the functions f and g are given in the problem statement). Such model can be used for describing steady liquid flow, stationary thermal fields, heat transportation processes with the internal heat sources, elastic plate deformation etc. This example is often used as a training problem for demonstrating the ways to provide efficient parallel computations (see Section 12 “Parallel methods of solving differential equations in partial derivatives”).

For simplicity we will further use the unit square as a function statement domain:

$$D = \{(x, y) \in \mathbb{R}^2 : 0 \leq x, y \leq 1\}.$$

The method of finite differences (the grid method) considered in this lab is most widely used for numerical solving of differential equations. Following this approach, the domain D is represented as a discrete (uniform, as a rule) set (grid) of points (nodes). Thus, for instance, the rectangular grid in the domain D can be given as follows (Figure 12.1):

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih, y_j = jh, 0 \leq i, j \leq N+1, \\ h = 1/(N+1), \end{cases}$$

where the value N specifies the number of inner nodes for each coordinate in D domain.

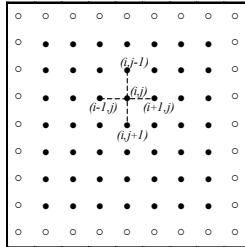


Figure 6.1. Rectangular grid in domain D (the dark points represent the inner nodes, the nodes are numbered from left to right in rows and from top to bottom in columns)

Let us denote by u_{ij} values of the function $u(x, y)$ at the points (x_i, y_j) . Then using the five-point template (see Figure 12.1) we can present the Poisson equation in the *finite difference form*:

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}.$$

This equation can be solved with regard to u_{ij} :

$$u_{ij} = 0.25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

It makes possible to determine u_{ij} value from the given values of the function $u(x, y)$ in the neighboring nodes of the used grid. This result serves as a basis for developing various iterative schemes for solving the Dirichlet problem. At the beginning of calculations in these schemes an initial approximation for values u_{ij} is formed, and then these values are sequentially recalculated in accordance with the given formula. Thus, for instance, the Gauss-Seidel method uses the following rule for updating values of approximations:

$$u_{ij}^k = 0.25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij}),$$

according to which the next k -th approximation of the value u_{ij} is calculated from the last k -th approximation of the values $u_{i-1,j}$ and $u_{i,j-1}$ and the next to last $(k-1)$ -th approximation of the values $u_{i+1,j}$ and $u_{i,j+1}$. Iterating continues till the variations of values u_{ij} , which are obtained as a result of iteration, become less than a certain given value (*the required accuracy*). The sequence of approximations obtained by this method converges to the Dirichlet problem solution, while the solution error is of h^2 order.

The pseudo-code for the described Gauss-Seidel algorithm for solving the Dirichlet problem may be presented as follows:

```
// Sequential Gauss-Seidel algorithm
do {
    dmax = 0; // maximal variation of the values u
    for ( i=1; i<N+1; i++ )
        for ( j=1; j<N+1; j++ ) {
            temp = u[i][j];
            u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                           u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
            dm = fabs(temp-u[i][j]);
            if ( dmax < dm ) dmax = dm;
        }
} while ( dmax > eps );
```

Further on we will assume the function f to be identically equal to zero, i.e. $f(x,y) \equiv 0$, in order to decrease the complexity of the lab to be executed.

Exercise 2 – Code the Serial Gauss-Seidel Program for Solving the Dirichlet Problem

The Exercise implies the necessity to implement the serial Gauss-Seidel algorithm for solving the Dirichlet problem. The initial version of the program to be developed is given in the project *SerialGaussSeidel*, which contains a part of the initial code, where the necessary project parameters are set. In the course of doing the Exercise, it is necessary to add the initial data input, the implementation of the Gauss-Seidel algorithm and the result output to the given program version.

Task 1 – Open the Project SerialGaussSeidel

Open the project *SerialGaussSeidel* 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\SerialGaussSeidel** in the dialogue window **Open Project**,
- Make double click on the file **SerialGaussSeidel.sln** or execute the command **Open** after selecting the file.

After the project has been opened in the window Solution Explorer (Ctrl+Alt+L), make double click on the file of the initial code *SerialGS.cpp*, as it is shown in Figure 6.2. After that, the code, which has to be enhanced, will be opened in the workspace of the Visual Studio.

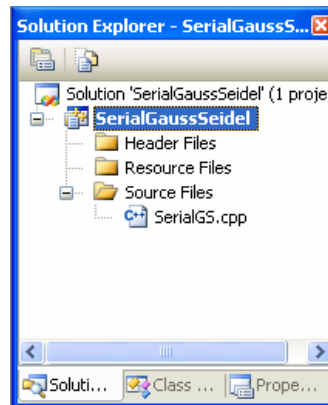


Figure 6.2. Opening the File SerialGS.cpp

The file *SerialGS.cpp* provides access to the necessary libraries and also contains the initial version of the main program function – the function *main*. The available program variant contains the declaration of variables and printout of the initial program message.

Let us consider the variables, which are used in the main function of the application. The first variable *pMatrix* is the matrix, which stores the values of the given domain nodes. The variable *Size* defines the matrix size (the matrix *pMatrix* is assumed to be square and its dimension is assumed to be $Size \times Size$). The variable *Eps* is used to store the required solution accuracy. To store the number of the executed iterations of the Gauss-Seidel algorithm we will use the variable *Iterations*.

```
double* pMatrix;    // Matrix of the grid nodes
int     Size;       // Matrix size
double  Eps;        // Required accuracy
int     Iterations; // Iteration number
```

It should be noted that in order to store the matrix *pMatrix* we should use a one-dimensional array, where the matrix is stored rowwise. 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 declaration of variables, is the initial message output and waiting for pressing any button before the termination of the application execution:

```
printf ("Serial Gauss - Seidel algorithm\n");
getch();
```

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

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

"Serial Gauss-Seidel algorithm".

In order to exit the program, press any key.

Task 2 – Input the Initial Data

In order to set the initial data of the serial Gauss-Seidel algorithm for solving the Dirichlet problem, we will develop the function *ProcessInitialization*. This function is intended for inputting the grid size in the solution domain (the size of the matrix *pMatrix*) and the required accuracy *Eps* of solving the problem and for allocating the required memory. Thus, the function should have the following heading:

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps);
```

At the first step it is necessary to input the grid size in the problem domain (to set the value of the variable *Size*). Add the bold marked code to the function *ProcessInitialization*:

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps) {
    // Setting the matrix size
    printf("\n Enter the grid size: ");
```

```
scanf("%d", &Size);
printf("\n Chosen grid size = %d", Size);
}
```

The user can enter the grid size, which is read from the standard input stream *stdin* and then the value is stored in the integer variable *Size*. The value of the variable *Size* is further printed out (Figure 12.3).

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

```
void main() {
    double* pMatrix;    // Matrix of the grid nodes
    int     Size;        // Matrix size
    double  Eps;         // Required accuracy
    int     Iterations;  // Iteration number

    printf ("Serial Gauss - Seidel algorithm\n");

    // Process initialization
    ProcessInitialization(pMatrix, Size, Eps);
    getch();
}
```

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

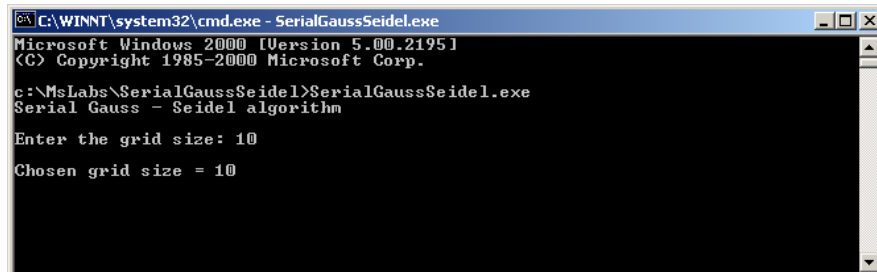


Figure 6.3. Setting the Grid Size

As in case of previous labs, we will control the input correctness. Let us implement the check of the grid size and, if there is an error there (the set size of less than 3), we will continue to ask for the grid size until some accepted value is entered. In order to provide this behavior we will place the code, which inputs the grid size, to the loop:

```
// Setting the grid size
do {
    printf("\n Enter the grid size of the initial objects: ");
    scanf("%d", &Size);
    printf("\n Chosen the grid size = %d", Size);
    if (Size <= 2)
        printf("\n Size of the grid must be greater than 2!\n");
}
while (Size <= 2);
```

Compile and run the application again. Try to enter an unaccepted number as the grid size. Make sure that invalid situations are processed correctly.

Let us implement now the input of the required accuracy value *Eps*. Add the following code to the the function *ProcessInitialization*:

```
// Setting the required accuracy
do {
    printf("\n Enter the required accuracy: ");
    scanf("%lf", &Eps);
    printf("\n Chosen accuracy = %lf", Eps);
    if (Eps <= 0)
        printf("\n Accuracy must be greater than 0!\n");
}
while (Eps <= 0);
```

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

Task 3 – Set the Initial Data

The initialization function must provide also memory allocation for storing data (add the bold marked code to the function *ProcessInitialization*):

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size) {
...
    // Setting the required accuracy
    do {
        <...>
    }
    while (Eps <= 0);

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

It is necessary to set the values of all elements of the matrix *pMatrix*. To carry out these operations, we will develop one more function *DummyDataInitialization*. The heading and the implementation of the function are given below:

```
// Function for simple setting the grid node values
void DummyDataInitialization (double* pMatrix, int Size) {
    int i, j; // Loop variables
    // Setting the grid node values
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
                pMatrix[i*Size+j] = 100;
            else
                pMatrix[i*Size+j] = 0;
    }
}
```

As you can see from the given code, this function provides setting the values of the function *u* in the grid nodes. While the value 0 is set for all the inner grid nodes, the value 100 is set for all the boundary nodes (i.e. the function *g* for setting the values at the domain border is equal to $g=100$).

The function *DummyDataInitialization* must be called immediately after the allocation of the memory in the function *ProcessInitialization*:

```
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps) {

    // Memory allocation
    <...>

    // Setting the grid node values
    DummyDataInitialization(pMatrix, Size);
}
```

Let us develop one more function, which will further help to control the initial data setting. This is the function of the formatted output of the matrix *PrintMatrix*. The pointer to the one-dimensional array, where the matrix is stored rowwise, and the matrix sizes both vertically (the number of rows *RowCount*) and horizontally (the number of columns *ColCount*) are given as arguments to the function *PrintMatrix*:

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

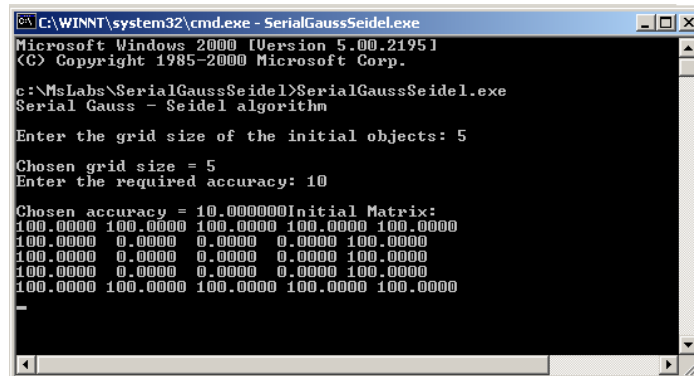
```
}
```

Let us add the call of the function to the main function:

```
// Process initialization
ProcessInitialization(pMatrix, Size, Eps);

// Matrix output
printf ("Initial Matrix: \n");
PrintMatrix (pMatrix, Size, Size);
```

Compile and run the application. Make sure that the data setting is performed according to the above-described rules (See Figure 6.4). Run the application several times, set various initial values.



```
C:\WINNT\system32\cmd.exe - SerialGaussSeidel.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

c:\MsLabs\SerialGaussSeidel\SerialGaussSeidel.exe
Serial Gauss - Seidel algorithm

Enter the grid size of the initial objects: 5
Chosen grid size = 5
Enter the required accuracy: 10
Chosen accuracy = 10.000000Initial Matrix:
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000
```

Figure 6.4. The Result of the Program Execution after the Termination of Task 3

Task 4 –Terminate the Program Execution

In this Task we should develop a function for correct program termination before the implementation of the Gauss-Seidel algorithm. For this purpose it is necessary to deallocate the memory, which has been allocated dynamically in the course of the program execution. Let us develop the corresponding function *ProcessTermination*. The memory was allocated for storing the initial matrix *pMatrix*. Consequently, this array must be given to the function *ProcessTermination* as argument:

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

The call of the function *ProcessTermination* must be executed before the termination of the function *main*:

```
// Process initialization
ProcessInitialization(pMatrix, Size, Eps);

// Matrix output
printf ("Initial Matrix: \n");
PrintMatrix (pMatrix, Size, Size);

// Computational process termination
ProcessTermination(pMatrix);
```

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

Task 5 – Implement the Gauss-Seidel Algorithm

Let us develop the main part of the computational program. In order to implement the Gauss-Seidel algorithm for solving the Dirichlet problem, we will develop the function *ResultCalculation*, which accepts the initial matrix *pMatrix*, the matrix size *Size*, the required accuracy of solving the problem *Eps* as input parameters. We will add the variable *Iterations* as an output parameter. In this variable the function will return the number of the executed iterations of the Gauss-Seidel algorithm until the required accuracy is achieved.

In accordance with the algorithm described in Exercise 1, the code of the function must be as follows:

```
// Function for the Gauss-Seidel algorithm
void ResultCalculation(double* pMatrix, int Size, double &Eps,
    int &Iterations) {
    int i, j; // Loop variables
    double dm, dmax, temp;
    Iterations = 0;
    do {
        dmax = 0;
        for (i = 1; i < Size - 1; i++)
            for (j = 1; j < Size - 1; j++) {
                temp = pMatrix[Size * i + j];
                pMatrix[Size * i + j] = 0.25 * (pMatrix[Size * i + j + 1] +
                    pMatrix[Size * i + j - 1] +
                    pMatrix[Size * (i + 1) + j] +
                    pMatrix[Size * (i - 1) + j]);

                dm = fabs(pMatrix[Size * i + j] - temp);
                if (dmax < dm) dmax = dm;
            }
        Iterations++;
    }
    while (dmax > Eps);
}
```

Let us call the function, which implements the Gauss-Seidel algorithm, from the main program. In order to control the correctness of the program execution, we will print out the matrix of values:

```
// Process initialization
ProcessInitialization(pMatrix, Size);

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

// Gauss-Seidel method
ResultCalculation(pMatrix, Size, Eps, Iterations);

// Printing the result
printf("\n Number of iterations: %d\n", Iterations);
printf ("\n Result matrix: \n");
PrintMatrix (pMatrix, Size, Size);

// Computational process termination
ProcessTermination(pMatrix);
```

Compile and run the application. Analyze the results of the Gauss-Seidel algorithm execution. Carry out several computational experiments changing the computation grid size.

```
C:\WINNT\system32\cmd.exe - SerialGaussSeidel.exe
c:\MsLabs\SerialGaussSeidel>SerialGaussSeidel.exe
Serial Gauss - Seidel algorithm
Enter the grid size of the initial objects: 5
Chosen grid size = 5
Enter the required accuracy: 10
Chosen accuracy = 10.000000Initial Matrix:
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000

Number of iterations: 5

Result matrix:
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 94.2078 94.1574 97.0703 100.0000
100.0000 94.1574 94.1406 97.0682 100.0000
100.0000 97.0703 97.0682 98.5341 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000
```

Figure 6.5. The Result of the Gauss-Seidel Algorithm Execution

Task 6 – Carry out the Computational Experiments

In order to estimate the further speed up of the parallel program, it is necessary to carry out experiments on the computation of the execution time for the serial algorithm. It is reasonable to analyze the execution time of the algorithm for large enough computational grid sizes.

To determine the time we will add the call of the functions, which allow us to find out the execution time, to the developed program. As previously we use the following function:

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

Let us add the computation and output of the time spent for solving the Dirichlet problem to the program code. For this purpose we will clock in before and after the call of the function *ResultCalculation*:

```
// Gauss-Seidel method
start = clock();
ResultCalculation(pMatrix, Size, Eps, Iterations);
finish = clock();
duration = (finish-start)/double(CLOCKS_PER_SEC);

// Printing the time spent by the Gauss-Seidel method
printf("\n Time of execution: %f", duration);
```

Compile and run the application. In order to carry out computational experiments with large computational grids, switch off the matrix result output (transform the corresponding code lines into comment). Carry out the computational experiments, and register the results in Table 6.1.

Table 6.1. The Results of the Computational Experiments for the Gauss-Seidel Method

Test Number	Grid Size	Number of Iterations	Execution Time (sec)
1	10		
2	100		
3	1000		
4	2000		
5	3000		
6	4000		

Let us estimate the computational complexity of the Gauss-Seidel algorithm analytically (see Section 12 “Parallel methods of solving differential equations in partial derivatives” of the training materials). The execution time for problem solving may be generally estimated according to the expression

$$T_1 = kmN^2, \quad (6.1)$$

where N is the number of inner nodes for each coordinate of the domain D , m is the number of operations performed by the method for a grid node ($m=6$), k is the number of method iterations before the accuracy requirement is met, and τ is the execution time of the basic computational operation.

Complete the table of comparison of the experiment execution time to the time obtained according to formula (6.1). In order to compute the execution time of the basic computational operation, we will use the following method: let us choose one of the experiments as a pivot (for instance, the experiment where the grid size is equal to 2000) and divide the execution time of this experiment by the number of the executed operations (the number of operations may be calculated according to the formula (6.1)). Thus, we will compute the execution time of the operation. Then we will use this value and compute the theoretical execution time for all the other experiments.

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

Table 6.2. The Comparison of the Experimental and the Theoretical Gauss-Seidel Method Execution Time

The Execution Time of Basic Computational Operation τ (sec):			
Test Number	Matrix Size	Execution Time (sec)	Theoretical Time (sec)
1	10		
2	100		
3	1000		
4	2000		

5	3000		
6	4000		

Task 3 – Develop the Parallel Gauss-Seidel Algorithm

For the purpose of developing the parallel algorithm, it is necessary to choose the method for partitioning the processed data among the processors. Two different methods of data distribution are possible in case of developing the parallel methods of solving the Dirichlet problem. These are the one-dimensional (or *block-striped*) scheme and two-dimensional (or *chessboard block*) computational grid partitioning.

We will consider the block-striped scheme of the computation decomposition in detail later (see also Section 12 “Parallel methods of solving differential equations in partial derivatives” of the training materials).

Subtask Definition

In case of block-striped partitioning the computational domain is divided into horizontal or vertical stripes (Figure 6.6a and 6.6b). The number of stripes is set by the number of processors. The stripe size is usually the same for all the processors. The horizontal boundary nodes (the first and the last rows) are included into the first and the last stripes correspondingly. The stripes for processing are distributed among the processors.

The division of the rows and columns into stripes is performed in the majority of cases on the continuous (sequential) basis. This is the approach that we use in this lab. This approach involves presenting the matrix A for horizontal rowwise partitioning in the following way:

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

where $a_i = (a_{i1}, a_{i2}, \dots, a_{in})$, $0 \leq i < m$, is the i -th row of the matrix A (the row number m is assumed to be divisible by the number of processes p without remainder, i.e. $m = k \cdot p$).

The essential aspect of implementing the Gauss – Seidel method computations with data distribution of this type is that the boundary rows of the previous and the next grid stripes should be replicated on the processor, which performs processing of the stripe. The replicated stripe boundary rows are used only for calculations. The recalculation of these rows is performed in the stripes of the initial row location. Thus, boundary rows should be replicated prior to the beginning of each iteration of the grid method.

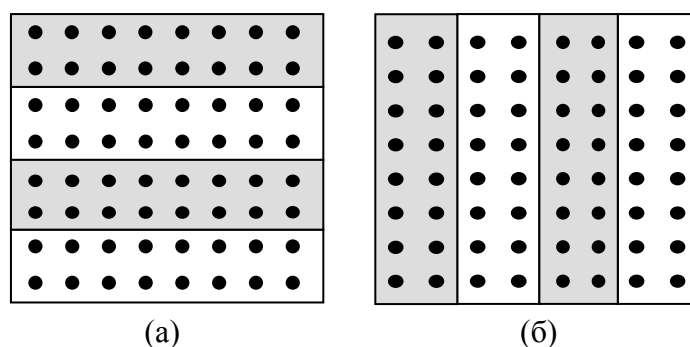


Figure 6.6. Block-Striped Decomposition of the Grid Nodes among the Processors

В качестве начального варианта рассмотрим предельный случай, когда количество процессоров совпадает с числом внутренних строк сетки, т.е. $p=N$. В такой ситуации полоса каждого процессора состоит из трех строк, из которых только одна является перевычисляемой, а две других строки дублируются с соседних процессов. Примем далее все вычисления, связанные с обработкой каждой из таких полос, в качестве *базовой вычислительной подзадачи*.

Analysis of Information Dependencies

The parallel version of the Gauss - Seidel method in case of block-striped data distribution consists in simultaneous stripes processing on all the available processors according to the following scheme:

```
// Gauss-Seidel method, block-striped data distribution
// Calculations performed on each processor
do {
  // < exchanging the boundary stripe rows among the neighbors >
```

```
// < stripe processing >
// < calculating the total computation error dmax >}
while ( dmax > eps ); // eps - the required accuracy of computations
```

To make the following algorithm presentation more precise let us introduce the following notation:

- **ProcNum** – the number of processor, which perform the described operations,
- **PrevProc**, **NextProc** – the numbers of neighboring processors containing the previous and the following stripes,
- **NP** – the number of processors,
- **M** – the number of rows in the stripe (the replicated boundary rows are not included),
- **N** – the number of inner nodes in a grid row (i.e. $N+2$ nodes in a row).

To enumerate the stripe rows we will use the enumeration where rows 0 and $M+1$ are the boundary rows replicated from the neighboring stripes and own stripe rows of the processor are enumerated from 1 to M .

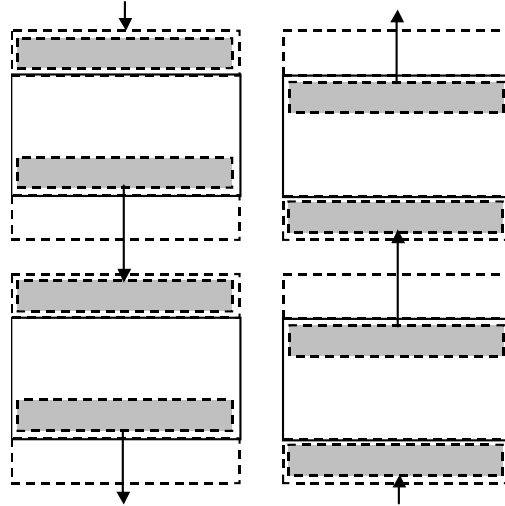


Figure 6.7. The Scheme of the Boundary Rows Transmission among the Neighboring Processors

The procedure of boundary row exchange among the neighboring processors may be divided into two sequent operations. During the first operation each processor transmits its lowest boundary row to the following processor and receives an identical row from the previous processor (see Figure 6.7). The second transmission part is performed in the reverse order: the processors transmit their upper boundary rows to the previous neighbors and receive identical rows from the following neighboring processors.

Carrying out such data transmissions may be represented in a general way as follows (for data transmissions we use a pseudo code close to MPI functions):

```
// Transmission of the lower boundary row to the following processor and
// receiving the transmitted row from the previous processor
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
```

The implementation of such combined function *Sendrecv* is usually performed so as to provide both the correct execution on the boundary processors, when it is not necessary to perform one of the transmission operations, and the alternation of the transmission procedures on the processors in order to avoid deadlock situations. It should be also noted that the function *Sendrecv* executes usually all the necessary data transmission operations in parallel.

In order to compute the total computational error for all the processors, you can use the cascade scheme. MPI provides the special function *MPI_Allreduce* for this purpose.

The general computational scheme for each processor may be presented in the pseudo-code in the following way:

```
// Gauss-Seidel method, block-striped data distribution
// Calculations performed on each processor
do {
    // Exchanging the boundary stripe rows among the neighbors
    Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
    Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
```

```
// < stripe processing with the error estimation dm >
// Computing the total computational error dmax
Allreduce(dm,dmax,MAX,0);
} while ( dmax > eps ); // eps - the required accuracy
```

Scaling and Distributing the Subtasks among the Processors

The number of the available processors p is, as a rule, considerably less than the number of the basic subtasks N ($p \ll N$). A possible way to aggregate computations is to use the block-stripped decomposition scheme of the matrix A . This approach corresponds to the aggregation of computations connected with updating the nodes of one or several grid rows (horizontal partitioning) or columns (vertical partitioning) of the matrix A within a basic computational subtask. These two types of partitioning are practically equal. Taking into account that the arrays are located rowwise for the algorithmic language C we will further consider only partitioning the matrix A into horizontal stripes.

Exercise 4 – Code the Parallel Gauss-Seidel Program for Solving the Dirichlet Problem

In order to perform the tasks, you will have to develop the parallel Gauss-Seidel program for solving the Dirichlet problem. This Exercise is aimed at:

- Getting additional skills at developing parallel programs, becoming familiar with variants of data decomposition, collective communication operations,
- Developing the first version of the parallel program, which implements the Gauss-Seidel algorithm for solving the Dirichlet problem.

As previously, the parallel program to be developed, will be composed of the following basic parts:

- 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 data exchange among the processes executed in parallel is carried out,
- Termination of MPI program.

Task 1 – Open the Project ParallelGaussSeidel

Open the project *ParallelGaussSeidel* 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\ParallelGaussSeidel**, in the dialog window **Open Project**,
- Make double click on the file **ParallelGaussSeidel.sln** or select it and execute the command **Open**.

After the project has been opened in the window Solution Explorer (Ctrl+Alt+L), make double click on the file *ParallelGS.cpp*, as it is shown in Figure 6.9. After that, the code, which has to be enhanced, will be opened in the workspace of the Visual Studio.

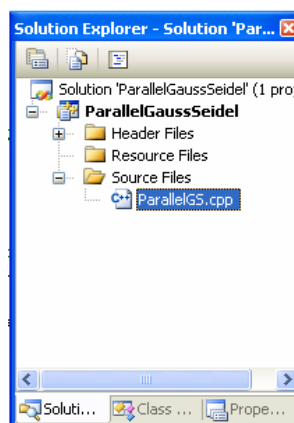


Figure 6.8. Opening the File ParallelGS.cpp with the Use of the Solution Explorer

The main function of the parallel program to be developed, which contains the declaration of the necessary variables, is located in the file *ParallelGS.cpp*. The following functions copied from the serial project are also located in the file *ParallelGS.cpp*: *DummyDataInitialization*, *ResultCalculation*, *PrintMatrix* (detailed information on the purpose of these functions is given in Exercise 2 of this lab). These functions may be also used in the parallel program. Besides, the drafts for the functions of the computation initialization (*ProcessInitialization*) and termination (*ProcessTermination*) are also located there.

Compile and run the application using the Visual Studio. Make sure that the initial message:

"Parallel Gauss - Seidel program"

is output into the command console.

Task 2 –Initialize and Terminate the Parallel Program

You should add the MPI header file to the program in order to use the MPI functions in your application. Add the bold marked line to the list of the libraries in the initial code of the parallel program:

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

Next you should initialize the MPI program environment, determine the number of available processes, the rank of the process in the communicator MPI_COMM_WORLD, and declare global variables for storing these values (*ProcNum* and *ProcRank* correspondingly). Add the following bold marked code:

```
static int ProcNum = 0;    // Number of the available processes
static int ProcRank = -1;  // Rank of the current process

void main(int argc, char* argv[]) {
    double* pMatrix;  // Matrix of the grid
    int     Size;      // Matrix size
    double  Eps;       // Required accuracy
    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 Gauss - Seidel algorithm \n");

    MPI_Finalize();
}
```

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 key **Start**, and the key **Run**,
2. Type the name of the program **cmd** in the dialog window, which appears on the screen (Figure 6.9)

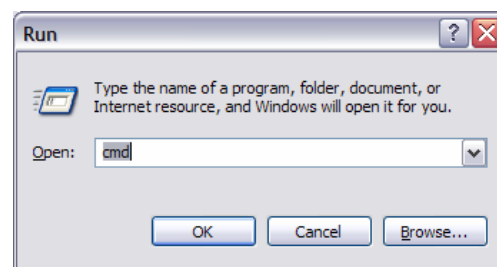


Figure 6.9. Command Prompt Start

In the command line go to the folder, which contains the developed program, which is being executed (Figure 6.10):

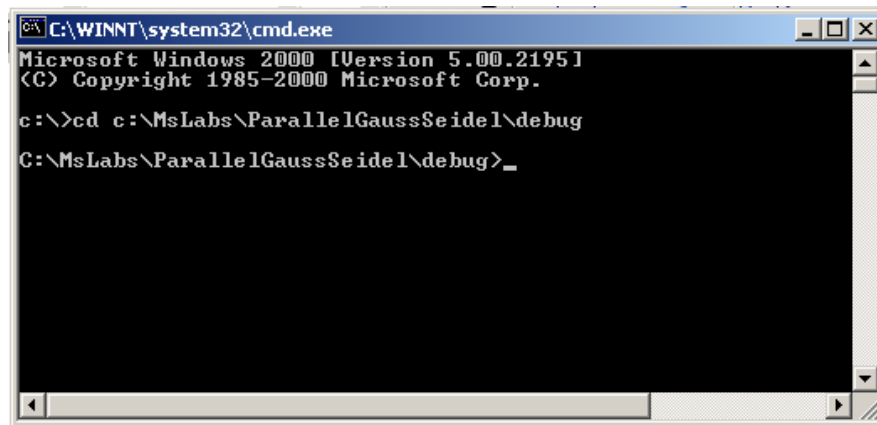


Figure 6.10. Setting the Folder, which Contains the Parallel Program

Type the command (Figure 6.11) in order to run the program using 4 processes:

```
mpiexec -n 4 ParallelGaussSeidel.exe
```

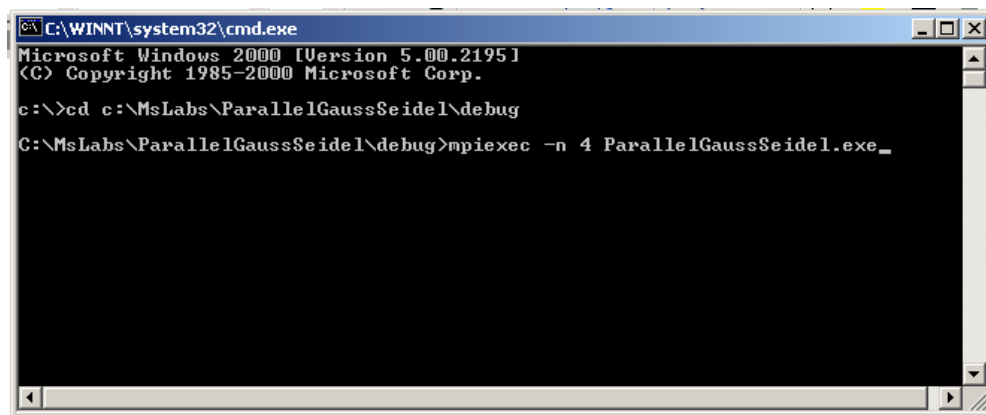


Figure 6.11. Starting the Parallel Program

Make sure that initial message
 "Parallel Gauss-Seidel algorithm"
 is output to the command console.

Task 3 – Input the Initial Data

At the next stage of developing the parallel application it is necessary to input the grid size, allocate memory for data storage and set the initial values.

The dialog with the user for inputting the grid size must be executed by only one process (let it be the process with the rank 0).

As previously, the function *ProcessInitialization* is used to initialize the computations:

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

First it is necessary to input the grid size, i.e. to set the value of the variable *Size*. Let us implement the dialog with the user in order to input the grid size. As in case of the previous labs, we will check the correctness of the input value. Add the following bold marked code to the function *ProcessInitialization*:

```
// Function for memory allocation and and data initialization
void ProcessInitialization (double* &pMatrix, double* &pProcRows,
    int &Size, int &RowNum, double &Eps) {
    if (ProcRank == 0) {
        do {
            printf("\n Enter the grid size: ");
            scanf("%d", &Size);
            printf("\n Chosen grid size = %d", Size);
```

```

    if (Size <= 2) {
        printf("\n Grid size must be greater than 2! \n");
    }
    if (Size < ProcNum) {
        printf("\n The grid size must be greater than"
            "the number of processes! \n ");
    }
    if ((Size-2)%ProcNum != 0) {
        printf("\n Number of inner rows of the grid must be divisible by"
            "the number of processes! \n");
    }
} while ( (Size <= 2) || (Size < ProcNum) || ((Size-2)%ProcNum != 0));
// Setting the required accuracy
do {
    printf("\n Enter the required accuracy: ");
    scanf("%lf", &Eps);
    printf("\n Chosen accuracy = %lf", Eps);
    if (Eps <= 0)
        printf("\n Accuracy must be greater than 0!\n");
} while (Eps <= 0);
}
}

```

After the values of the variables *Size* and *Eps* have been defined correctly, it is necessary to broadcast these values to the other processes. For this purpose we should use the function of the broadcast *MPI_Bcast*. Add the following code to the program. Pay attention to the fact that the call of the function *MPI_Bcast* must be executed by all the processes:

```

if (ProcRank == 0) {
    <...>
}
// Broadcasting the grid size
MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// Calculating the number of grid rows stored on each process
RowNum = (Size-2)/ProcNum + 2;

```

As it can be noted the number of the grid rows located on each process is also calculated in the added code.

Add the call of the initialization function to the program *main*. Compile and run the application. Make sure that all the invalid situations are processed correctly. For this purpose, run the application several times setting various number of parallel processes (by means of the utility *mpiexec*) and various values of input data.

As the next step we should allocate the memory and set the values of the grid nodes. Setting the initial data is performed by the process with the rank 0 only. Then, according to the scheme of parallel computations, described in Exercise 3, the grid nodes should be distributed among the processes in such a way that each process operates with a continuous sequence of grid rows (a horizontal stripe). It should be noted that the first version of the program being developed is oriented at the case when the number of the inner grid nodes is divisible by the number of processes, i.e. the grid stripes on all the processes contain the same number of grid rows. In order to store the stripe size we will use the variable *RowNum*. The horizontal stripe of grid rows on each process, will be stored in the variable *pProcRows* (*pProcRows* is the matrix, which contains *RowNum* rows and *Size* columns and which is stored rowwise).

Let us declare the following variables in the main program functions:

```

void main(int argc, char* argv[]) {
    double* pMatrix; // Matrix of the grid
    int Size; // Grid size
    double Eps; // Required accuracy
    double* pProcRows; // Stripe of the matrix on current process
    int RowNum; // Number of rows in matrix stripe
    double Start, Finish, Duration;

```

Let us allocate the memory for storing the data and initialize the grid node values on the root process (the process with the rank 0). We will use the function *DummyDataInitialization* in order to set the grid node values.

Add the bold marked code to the function *ProcessInitialization*:

```

<...>
// Calculating the number of grid rows stored on each process
RowNum = (Size-2)/ProcNum + 2;

// Memory allocation
pProcRows = new double [RowNum*Size];

// Setting the initial values of the grid nodes
if (ProcRank == 0) {
    // Initial matrix exists only on the root process
    pMatrix = new double [Size*Size];
    // Values of grid nodes are set only on the root process
    DummyDataInitialization(pMatrix, Size);
}

```

In order to control the correctness of the initial data input, it is possible to use the function *PrintMatrix*. After the call of the function *ProcessInitialization* in the main program function, add the call of the function *PrintMatrix* to print the matrix *pMatrix* on the root process. Compile and run the application. Make sure that the data is set correctly.

Task 4 –Terminate the Calculations

In order to make the application complete at each stage of the development, we will develop the function for correct program termination. For this purpose it is necessary develop the function *ProcessTermination* in order to deallocate the memory for storing the matrix *pMatrix* and the matrix stripe *pProcRows*. All these arrays must be given to the function *ProcessTermination* as arguments:

```

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

```

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

```

// Process termination
ProcessTermination(pMatrix, pProcRows);
MPI_Finalize();
}

```

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

Task 5 – Distribute the Data among the Processes

At this stage it is necessary to execute the data distribution in accordance with the parallel computation scheme, given in the previous Exercise; the grid matrix must be distributed among the processes in equal horizontal stripes. For this purpose let us develop the function *DataDistribution*. It should be provided that the following parameters must be given to the function as arguments: the grid matrix *pMatrix*, the grid matrix stripe *pProcRows*, and the matrix and stripe sizes (the size of the matrix *Size* and the number of rows in the horizontal stripe *RowNum*):

```

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

```

Broadcasting the values of the grid matrix *pMatrix* may be provided with the help of the function *MPI_Scatter*. In addition the upper boundary row of the grid should be copied to the process 0, and the lower boundary row of the grid should be sent to the process with the rank *ProcNum-1*.

```

// Function for distribution of the grid rows among the processes
void DataDistribution(double* pMatrix, double* pProcRows, int Size,
    int RowNum) {
    MPI_Status status;
    MPI_Scatter(pMatrix+Size, (RowNum-2)*Size, MPI_DOUBLE, pProcRows+Size,
        (RowNum-2)*Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    // Copying the upper boundary row to the process 0
}

```



```

if (ProcRank == 0 ){
    for(int i=0;i<Size;i++)
        pProcRows[i]=pMatrix[i];
}
// Sending the lower boundary row to the process ProcNum-1
if (ProcRank == 0)
    MPI_Send(pMatrix + Size * (Size-1), Size, MPI_DOUBLE,
        ProcNum - 1, 5, MPI_COMM_WORLD);
if (ProcRank == ProcNum - 1)
    MPI_Recv(pProcRows + (RowNum - 1 ) * Size,
        Size, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD, &status);
}

```

The call of the function *DataDistribution* must be executed immediately after the call of the initialization function *ProcessInitialization*:

```

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

// Data distribution among the processes
DataDistribution(pMatrix, pProcRows, Size, RowNum);

```

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

In order to provide the formatted output of the matrix we will use the method *PrintMatrix*:

```

// Function for testing the data distribution
void TestDistribution(double* pMatrix, double* pProcRows, int Size,
    int RowNum) {
    if (ProcRank == 0) {
        printf("Initial Matrix: \n");
        PrintMatrix(pMatrix, Size, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    for (int i=0; i<ProcNum; i++) {
        if (ProcRank == i) {
            printf("\nProcRank = %d \n", ProcRank);
            fprintf(" Matrix Stripe:\n");
            PrintMatrix(pProcRows, RowNum, Size);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
}

```

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

```

// Data distribution among the processes
DataDistribution(pMatrix, pProcRows, Size, RowNum);

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

```

Compile the application. If you find errors, correct them, comparing your code to the code given in this exercise. Run the application using three processes and set the grid size equal to 5. Make sure that the data distribution is performed correctly (Figure 6.12).


```
// do {
    Iterations++;
    // Exchanging the boundary rows of the process stripe
    ExchangeData(pProcRows, Size, RowNum);
// } while(Iteration < 2);
}
```

(it should be noted that the maximum possible number of the algorithm iterations is set to 2 in an effort to simplify the developed program testing.

The matrix stripe *pProcRows*, the row size *Size*, and the number of stripe rows *RowNum* are the arguments of the function *ExchangeData*. The procedure of exchanging the boundary rows among the neighboring processes may be implemented with the use of the function *MPI_Sendrecv*:

```
// Function for exchanging the boundary rows of the process stripe
void ExchangeData(double* pProcRows, int Size, int RowNum) {
    MPI_Status status;
    int NextProcNum = (ProcRank == ProcNum-1)? MPI_PROC_NULL : ProcRank+1;
    int PrevProcNum = (ProcRank == 0)          ? MPI_PROC_NULL : ProcRank-1;
    // Send to NextProcNum and receive from PrevProcNum
    MPI_Sendrecv(pProcRows+Size*(RowNum-2), Size, MPI_DOUBLE, NextProcNum, 4,
        pProcRows, Size, MPI_DOUBLE, PrevProcNum, 4, MPI_COMM_WORLD, &status);
    // Send to PrevProcNum and receive from NextProcNum
    MPI_Sendrecv(pProcRows + Size, Size, MPI_DOUBLE, PrevProcNum, 5,
        pProcRows + (RowNum-1)*Size, Size, MPI_DOUBLE, NextProcNum, 5,
        MPI_COMM_WORLD, &status);
}
```

Compile the application. If you find errors, correct them, comparing your code to the code given in the exercise. Run the application using three processes and set the grid size equal to 5. Make sure that the boundary row exchange is performed correctly (Figure 6.13).

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\MsLabs\ParallelGaussSeidel\debug>mpiexec -n 3 ParallelGaussSeidel.exe
Parallel Gauss - Seidel algorithm

Enter the grid size: 5

Enter the required accuracy: 0.5

Chosen accuracy = 0.500000Initial Matrix:
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000

ProcRank = 0
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
ProcRank = 1
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
ProcRank = 2
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000

c:\MsLabs\ParallelGaussSeidel\debug>
```

Figure 6.13. The Grid Matrix Distribution After the Boundary Row Exchange (the Parallel Program uses three Processes and the Grid Size Is Equal to 5)

Let us note that after the boundary row exchange the values of these rows are set and are printed correctly.

Task 7 – Implement the Parallel Algorithm Iterations

Let us enhance the implementation of the function *ParallelResultCalculation* using the function *IterationCalculation* in order to execute the iterations of the parallel Gauss-Seidel algorithm for solving the

Dirichlet problem. For processing you should provide to the functions the grid matrix stripe *pProcRows*, the row size *Size* and the number of the stripe rows *RowNum* as input parameters:

```
// Function for the execution of the Gauss-Seidel method iteration
double IterationCalculation(double* pProcRows, int Size, int RowNum);
```

In accordance with the algorithm described in Exercise 1, the program code of the function will look as follows:

```
// Function for the execution of the Gauss-Seidel method iteration
double IterationCalculation(double* pProcRows, int Size, int RowNum) {
    int i, j; // Loop variables
    double dm, dmax, temp;
    dmax = 0;
    for (i = 1; i < RowNum-1; i++)
        for(j = 1; j < Size-1; j++) {
            temp = pProcRows[Size * i + j];
            pProcRows[Size * i + j] = 0.25 * (pProcRows[Size * i + j + 1] +
                pProcRows[Size * i + j - 1] +
                pProcRows[Size * (i + 1) + j] +
                pProcRows[Size * (i - 1) + j]);
            dm = fabs(pProcRows[Size * i + j] - temp);
            if (dmax < dm) dmax = dm;
        }
    return dmax;
}
```

Add the call of the function *IterationCalculation* to the function *ParallelResultCalculation*:

```
// Function for the parallel Gauss-Seidel method
void ParallelResultCalculation(double* pProcRows, int Size, int RowNum,
    double Eps, int &Iterations) {
    <...>
    // do {
        Iterations++;
        // Exchanging the boundary rows of the process stripe
        ExchangeData(pProcRows, Size, RowNum);

        // The Gauss-Seidel method iteration
        ProcDelta = IterationCalculation(pProcRows, Size, RowNum);
        TestDistribution(pMatrix, pProcRows, Size, RowNum);
    // } while(Iteration < 2);
}
```

As it can be seen from the given program code, the call of the test function *TestDistribution* must be moved to the line after the call of the newly developed function *IterationCalculation*.

Compile the application again. Carry out experiments and make sure that the computations are performed correctly. It should be noted that the obtained results may differ from the results of the serial algorithm and the only method of control is the manual check of the computation results (to simplify debugging it may be reasonable to transform the loop into the comment).

If the application is run using three processes and the grid size is equal to 5, the computation results must coincide with the given ones (Figure 6.14). To continue the check up, turn the comments into the loop operator in the function *ParallelResultCalculation* (you may also change the constant in the loop condition and set the necessary number of the executed parallel Gauss-Seidel method iterations).

```

C:\WINDOWS\system32\cmd.exe

c:\MsLabs\ParallelGaussSeidel\debug>mpiexec -n 3 ParallelGaussSeidel.exe
Parallel Gauss - Seidel algorithm

Enter the grid size: 5

Enter the required accuracy: 0.5

Chosen accuracy = 0.500000Initial Matrix:
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000

ProcRank = 0
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 65.6250 57.8125 71.0938 100.0000
100.0000 25.0000 6.2500 26.5625 100.0000

ProcRank = 1
100.0000 50.0000 37.5000 59.3750 100.0000
100.0000 51.5625 38.2813 64.2578 100.0000
100.0000 50.0000 37.5000 59.3750 100.0000

ProcRank = 2
100.0000 25.0000 6.2500 26.5625 100.0000
100.0000 65.6250 57.8125 71.0938 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000

```

Figure 6.14. The Calculation Results in Case when the Application Is Run Using three Processes and the Grid Size Is Equal to 5

Task 8 – Calculating the Maximum Result Deviation

To implement the Gauss-Seidel method completely we have only to add the calculation of the maximum computation result deviation obtained at the algorithm iteration. The necessary changes of the function *ParallelResultCalculation* consist in the following (add the bold marked code):

```

// Function for the parallel Gauss-Seidel method
void ParallelResultCalculation(double* pProcRows, int Size, int RowNum,
double Eps, int &Iterations) {
    <...>
    do {
        <...>
        // Calculating the maximum value of the deviation
        MPI_Allreduce(&ProcDelta, &Delta, 1, MPI_DOUBLE, MPI_MAX,
            MPI_COMM_WORLD);
    } while( Delta > Eps );
}

```

Pay attention to the change of the loop condition of the Gauss-Seidel method iteration repetition cycle. To decrease the amount of the debugging output, delete the call of the function *TestDistribution*.

Compile the application and check the correctness of the executed computations. Thus, for instance, if the parallel application is run using three processes and the grid size is equal to 5, the required accuracy is equal to 0.1, the processes must obtain the results given in Figure 6.15.

```

C:\WINDOWS\system32\cmd.exe

c:\MsLabs\ParallelGaussSeidel\debug>mpiexec -n 3 ParallelGaussSeidel.exe
Parallel Gauss - Seidel algorithm

Enter the grid size: 5

Enter the required accuracy: 0.5

Chosen accuracy = 0.500000Initial Matrix:
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000

ProcRank = 0
100.0000 100.0000 100.0000 100.0000 100.0000
100.0000 50.0000 37.5000 59.3750 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000

ProcRank = 1
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 25.0000 6.2500 26.5625 100.0000
100.0000 0.0000 0.0000 0.0000 100.0000

ProcRank = 2
100.0000 0.0000 0.0000 0.0000 100.0000
100.0000 50.0000 37.5000 59.3750 100.0000
100.0000 100.0000 100.0000 100.0000 100.0000

```

Figure 6.15. The Results of the Parallel Gauss-Seidel Algorithm, in Case when the Application Is Run Using Three Processes and the Grid Size is Equal to 5

Task 9 – Gather the Results

After the termination of the Gauss-Seidel method, it is necessary to gather the grid stripes located on different processes, on the root process (the process with the rank 0). Let us use the function *MPI_Allgather*, which composes a single array of the blocks located on different communicator processes.

The function of gathering results *ResultCollection* will consist only of the call of the function *MPI_Allgather*:

```

// Function for gathering the calculation results
void ResultCollection(double* pMatrix, double* pProcRows, int Size,
    int RowNum) {
    MPI_Gather(pProcRows+Size, (RowNum-2)*Size, MPI_DOUBLE, pMatrix+Size,
        (RowNum-2)*Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

```

Add the call of the function *ResultCollection* to the function *main*:

```

// Parallel Gauss-Seidel method
ParallelResultCalculation(pProcRows, Size, RowNum, Eps, Iterations);
// TestDistribution(pMatrix, pProcRows, Size, RowNum);

// Gathering the calculation results
ResultCollection(pMatrix, pProcRows, Size, RowNum);
TestDistribution(pMatrix, pProcRows, Size, RowNum);

```

Compile and run the application, Estimate the correctness of its execution. Use, as previously, the print function *TestDistribution* to check the correctness.

Task 10 – Test the Parallel Program Correctness

After the function of the result collection is performed, it is necessary to check 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.

To make the serial algorithm *SerialResultCalculation* operate the same data as the developed function *ParallelResultCalculation*, it is necessary to produce a copy of the data using the function *CopyData*:

```

// Function to copy the initial data
void CopyData(double *pMatrix, int Size, double *pSerialMatrix) {
    copy(pMatrix, pMatrix + Size, pSerialMatrix);
}

```

Let us add the call of this function to the program code. It is necessary to declare the variable for storing the copy of the data and to make ready this copy:

```
...
double* pMatrix;          // Matrix of the grid
double* pProcRows;        // Stripe of the matrix on the current process
double* pSerialMatrix; // Result of the serial method
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
<...>
// Data distribution among the processes
DataDistribution(pMatrix, pProcRows, Size, RowNum);

// Creating the copy of the initial data
if (ProcRank == 0) {
    pMatrixCopy = new double[Size*Size];
    CopyData(pMatrix, Size, pSerialMatrix);
}
```

Besides, it is necessary to deallocate the allocated memory for the serial algorithm:

```
// Process termination
if (ProcRank == 0) delete []pSerialMatrix;
ProcessTermination(pMatrix, pProcRows);
MPI_Finalize();
```

The function *TestResult* must have access to the matrices *pMatrix* and *pCMatrix* and should be executed only on the root process:

```
// Function for testing the computation result
void TestResult(double* pMatrix, double* pSerialMatrix, int Size,
double Eps) {
    int equal = 0; // =1, if the matrices are not equal
    int Iter;

    if (ProcRank == 0) {
        SerialResultCalculation(pSerialMatrix, Size, Eps, Iter);
        for (int i=0; i<Size*Size; i++) {
            if (fabs(pSerialMatrix[i]-pMatrix[i]) >= Eps) {
                equal = 1; break;
            }
        }
        if (equal == 1)
            printf("The results of the sequential and parallel programs"
                "are NOT identical. Check your code.");
        else
            printf("The results of the sequential and parallel programs"
                "are identical.");
    }
}
```

The result of the function execution is printing a diagnostic message. You can test the result of the parallel program regardless of the initial data values with the help of this function.

It should be noted that in the general case the results of the serial and the parallel variants of the Gauss-Seidel methods may differ, as the sequences of processing the grid nodes may be different in these two variants. However, the deviation of the obtained calculation results must be within the limits of the required accuracy *Eps* (see Section 12 “Parallel methods of solving the differential equation in partial derivatives” of the training materials).

Transform the call of the debugging function *TestDistribution*, which has been previously used for testing the correctness of the parallel program, into comments. Instead of the function *DummyDataInitialization*, which generates the initial data of the simple type, call the function *RandomDataInitialization*, which generates the initial data in the innerr grid nodes by means of the random data generator.

```
// Function for setting the grid node values by a random generator
void RandomDataInitialization (double* pMatrix, int Size) {
```

```

int i, j; // Loop variables
srand(unsigned(clock()));
// Setting the grid node values
for (i=0; i<Size; i++) {
    for (j=0; j<Size; j++)
        if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
            pMatrix[i*Size+j] = 100;
        else
            pMatrix[i*Size+j] = rand()/double(1000);
    }
}

```

Compile and run the application. Set different grid sizes and the values of the required computation accuracy. Make sure that the application operates correctly.

Task 11 – Implement the Gauss-Seidel Algorithm for Any Given Grid

The parallel program, which was developed in the course of executing the previous tasks, was implemented for the case when the number of the inner grid nodes ($Size-2$) is divisible by the number of processors $ProcNum$. In this case the grid matrix is divided among the processes in equal stripes, and the number of rows $RowNum$ processed by the process is the same for all the processes.

Let us consider the general case when the number of the inner grid nodes ($Size-2$) is not divisible by the number of processes $ProcNum$. In this case the number of rows in the stripe on each process can be different: some processes will get $\lfloor (Size-2)/ProcNum \rfloor + 2$, and the rest of them - $\lceil (Size-2)/ProcNum \rceil + 2$ matrix rows (the operation $\lfloor \rfloor$ means rounding the value down to the nearest smaller integer number, the operation $\lceil \rceil$ means rounding the value up to the nearest greater integer number).

Let us eliminate the processing of an invalid situation in the function *ProcessInitialization*. This situation occurs in the case when the number of inner grid nodes is not divisible by the number of processes. Now we will use the following distribution algorithm: we will allocate the rows to processes sequentially. It is necessary to determine how many rows the process with the rank 0 should operate, then the process with the rank 1 etc. The process with the rank 0 is allocated $\lfloor (Size-2)/ProcNum + 2 \rfloor$ rows (the result of the operation $\lfloor \rfloor$ coincides with the result of the integer division). After the execution of this operation we have to distribute $Size - \lfloor (Size-2)/ProcNum \rfloor - 2$ rows among $ProcNum-1$ processes etc. As a result, each next process i is assigned the number of rows equal to the result of the integer division of the remaining row number $RestRows$ by the remaining process numbers, i.e. $\lfloor (RestRows-2)/(ProcNum-i)+2 \rfloor$ rows.

Let us change the program code for calculating the value of the variable $RowNum$ in the function *ProcessInitialization*:

```

// Function for allocating the memory and initialization of the grid nodes
void ProcessInitialization (double* &pMatrix, double* &pProcRows, int &Size,
    int &RowNum, double &Eps) {
    int RestRows; // Number of the rows, that have't been distributed yet
    <...>
    // Define the number of the matrix rows stored on each process
    RestRows = Size - 2;
    for (i=0; i<ProcRank; i++)
        RestRows = RestRows - RestRows / ( ProcNum - i );
    RowNum = RestRows / ( ProcNum - ProcRank ) + 2;
    <...>
}

```

In case when the grid matrix is distributed among process unequally, we cannot use the function *MPI_Scatter* for data distribution. Instead we should use a more general function *MPI_Scatterv*, which gives the opportunity to one of the processes to distribute the data among the communicator processes in continuous element blocks of unequal size.

In order to call the function *MPI_Scatterv* it is necessary to define two auxiliary arrays for setting the offset and the size of the transmitted blocks. Let us add the necessary changes in the code of the function *DataDistribution*:

```

// Function for distribution of the grid rows among the processes
void DataDistribution(double* pMatrix, double* pProcRows, int RowNum,
    int Size) {

```



```

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;
// 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 - 2 ) / ProcNum + 2;
pSendNum[0] = RowNum * Size;
pSendInd[0] = 0;
for (int i=1; i < ProcNum; i++) {
    RestRows = RestRows - RowNum + 2;
    RowNum = ( RestRows - 2 ) / ( ProcNum - i ) + 2;
    pSendNum[i] = RowNum * Size;
    pSendInd[i] = pSendInd[i-1] + pSendNum[i-1] - Size;
}
// Scatter the rows
MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
delete []pSendInd;
delete []pSendNum;
}

```

Very much in the same way we will use the more general function *MPI_Allgatherv* for data gathering instead of the function *MPI_Allgather*. As in case of using *MPI_Scatterv*, the use of the function *MPI_Allgatherv* requires two additional arrays:

```

// Function for gathering the result vector
void ResultCollection(double *pMatrix, double* pProcResult,
    int Size, int RowNum) {
    int *pReceiveNum; // Number of elements, that the current process sends
    int *pReceiveInd; // Index of the first element of the received block
    int RestRows = Size;
    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;
    RowNum = ( Size - 2 ) / ProcNum + 2;
    pReceiveNum[0] = RowNum * Size;
    for ( i=1; i < ProcNum; i++){
        RestRows = RestRows - RowNum + 2;
        RowNum = ( RestRows - 2 ) / ( ProcNum - i ) + 2;
        pReceiveNum[i] = RowNum * Size;
        pReceiveInd[i] = pReceiveInd[i-1] + pReceiveNum[i-1] - Size;
    }
    //Gather the whole result vector on every processor
    MPI_Allgatherv(pProcRows, pReceiveNum[ProcRank], MPI_DOUBLE, pMatrix,
        pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

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

```

Compile and run the application. Check the correctness of the Gauss-Seidel algorithm execution by means of the function *TestDistribution*.

Task 12 – Carry out the Computational Experiments

The main challenge in the development of the parallel algorithms for solving complicated computational problems is to provide the increase of speed up (in comparison with the serial algorithm) at the expense of using several processors. The execution time of the parallel algorithm should be less than the execution time of the serial algorithm.

Let us determine the parallel algorithm execution time. For this purpose we will add clocking to the program code. As the parallel algorithm includes the stage of data distribution, the stage of the computation of the Gauss-Seidel iterations on each process and the stage of result gather, the clocking should start immediately before the call of the function *DataDistribution*, and stop right after the execution of the function *ResultCollection*:

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

Start = MPI_Wtime();
// Data distribution among the processes
DataDistribution(pMatrix, pProcRows, Size,RowNum);

// the Paralle Gauss-Seidel method
ParallelResultCalculation(pProcRows, Size,RowNum,Eps, Iterations);

//Gathering the calculation results
ResultCollection(pProcRows, pMatrix, Size, RowNum);
Finish = MPI_Wtime();
Duration = Finish-Start;

<...>
```

It is obvious that this way we will obtain the execution time of the root process. The execution time of the other processes may appear to be slightly different. But this difference must not be significant, as we paid special attention to the equal loading (balancing) of processes at the stage of the development of the parallel algorithm.

Add the selected code to the main function. Compile and run the application. Carry out the computational experiments and register the results in Table 6.3.

Table 6.3. The Results of the Computational Experiments for the Parallel Gauss-Seidel Algorithm

Grid size	Serial Algorithm	Parallel Algorithm					
		2 processors		4 processors		8 processors	
		Time	Speed up	Time	Speed up	Time	Speed up
10							
100							
1000							
2000							
3000							
4000							
5000							
6000							
7000							
8000							
9000							
10000							

The column “Serial Algorithm” is assigned for writing the execution times of the serial Gauss-Seidel algorithm measured in the course of testing the serial application in Exercise 2. In order to compute the speed up divide the execution time of the serial program by the parallel programm execution time. Give the results in the corresponding column of the table 6.3.

В отличие от ранее выполненных лабораторных работ проведите самостоятельно теоретическую оценку времени выполнения параллельного алгоритма Гаусса-Зейделя. Полученные оценки внесите в таблицу 6.4 и сравните с реальным временем выполнения экспериментов.

Table 6.4. The Computation Speed Up Obtained for the Parallel Gauss-Seidel Algorithm

Grid Size	2 processors		4 processors		8 processors	
	Model	Experiment	Model	Experiment	Model	Experiment
10						
100						
1000						
2000						
3000						
4000						
5000						
6000						
7000						
8000						
9000						
10000						

Discussions

- Were the theoretical and the experiment execution time congruent? What might be the reason for incongruity?

Exercises

1. Study the parallel Gauss-Seidel algorithm for solving the Dirichlet problem, based on block-stripped vertical matrix partitioning. Develop a program, which implements the algorithm.
2. Study the parallel Gauss-Seidel algorithm for solving the Dirichlet problem, based on chessboard block matrix partitioning. Develop a program, which implements the algorithm.

Appendix 1. The Program Code of the Serial Gauss-Seidel Algorithm

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

// Function for the Gauss-Seidel algorithm
void ResultCalculation(double* pMatrix, int Size, double &Eps,
    int &Iterations) {
    double dm, dmax, temp;
    int i, j; // Loop variables
    Iterations = 0;
    do {
        dmax = 0;
        for (i = 1; i < Size - 1; i++)
            for(j = 1; j < Size - 1; j++) {
                temp = pMatrix[Size * i + j];
                pMatrix[Size * i + j] = 0.25 * (pMatrix[Size * i + j + 1] +
                    pMatrix[Size * i + j - 1] +
                    pMatrix[Size * (i + 1) + j] +
                    pMatrix[Size * (i - 1) + j]);

                dm = fabs(pMatrix[Size * i + j] - temp);
                if (dmax < dm) dmax = dm;
            }
        Iterations++;
    }
    while (dmax > Eps);
}
```

```

// Function for computational process termination
void ProcessTermination(double* pMatrix) {
    delete [] pMatrix;
}
// 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 simple setting the grid node values
void DummyDataInitialization (double* pMatrix, int Size) {
    int i, j; // Loop variables
    double h = 1.0 / (Size - 1);
    // Setting the grid node values
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
                pMatrix[i*Size+j] = 100;
            else
                pMatrix[i*Size+j] = 0;
    }
}
// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, int &Size, double &Eps) {
    // Setting the grid size
    do {
        printf("\nEnter the grid size: ");
        scanf("%d", &Size);
        printf("\nChosen grid size = %d", Size);
        if (Size <= 2)
            printf("\nSize of grid must be greater than 2!\n");
    } while (Size <= 2);
    // Setting the required accuracy
    do {
        printf("\nEnter the required accuracy: ");
        scanf("%lf", &Eps);
        printf("\nChosen accuracy = %lf", Eps);
        if (Eps <= 2)
            printf("\nAccuracy must be greater than 0!\n");
    } while (Eps <= 0);

    // Memory allocation
    pMatrix = new double [Size*Size];

    // Setting the grid node values
    DummyDataInitialization(pMatrix, Size);
}

void main() {
    double* pMatrix; // Matrix of the grid nodes
    int Size; // Matrix size
    double Eps; // Required accuracy
    int Iterations; // Iteration number

    printf ("Serial Gauss - Seidel algorithm\n");

    // Process initialization
    ProcessInitialization(pMatrix, Size, Eps);
}

```

```

// Matrix output
printf ("Initial Matrix: \n");
PrintMatrix (pMatrix, Size, Size);

// The Gauss-Seidel method
ResultCalculation(pMatrix, Size, Eps, Iterations);

// Printing the result
printf("\n Number of iterations: %d\n",Iterations);
printf ("\n Result matrix: \n");
PrintMatrix (pMatrix, Size, Size);
getch();

// Computational process termination
ProcessTermination(pMatrix);
}

```

Appendix 2 – The Program Code of the Parallel Gauss-Seidel Algorithm

```

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

static int ProcNum = 0; // Number of available processes
static int ProcRank = -1; // Rank of current process

// Function for distribution of the grid rows among the processes
void DataDistribution(double* pMatrix, double* pProcRows, int RowNum,
int Size) {
    int *pSendNum; // Number of elements sent to the process
    int *pSendInd; // Index of the first data element sent to the process
    int RestRows=Size;
    // 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-2)/ProcNum+2;
    pSendNum[0] = RowNum*Size;
    pSendInd[0] = 0;
    for (int i=1; i<ProcNum; i++) {
        RestRows = RestRows - RowNum + 2;
        RowNum = (RestRows-2)/(ProcNum-i)+2;
        pSendNum[i] = (RowNum)*Size;
        pSendInd[i] = pSendInd[i-1]+pSendNum[i-1]-Size;
    }
    // Scatter the rows
    MPI_Scatterv(pMatrix , pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
    delete []pSendInd;
    delete []pSendNum;
}

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

```

```

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

// Function for the execution of the Gauss-Seidel method iteration
double IterationCalculation(double* pProcRows, int Size, int RowNum) {
    int i, j; // Loop variables
    double dm, dmax,temp;
    dmax = 0;
    for (i = 1; i < RowNum-1; i++)
        for(j = 1; j < Size-1; j++) {
            temp = pProcRows[Size * i + j];
            pProcRows[Size * i + j] = 0.25 * (pProcRows[Size * i + j + 1] +
                pProcRows[Size * i + j - 1] +
                pProcRows[Size * (i + 1) + j] +
                pProcRows[Size * (i - 1) + j]);

            dm = fabs(pProcRows[Size * i + j] - temp);
            if (dmax < dm) dmax = dm;
        }
    return dmax;
}

// Function for testing the data distribution
void TestDistribution(double* pMatrix, double* pProcRows, int Size,
    int RowNum) {
    if (ProcRank == 0) {
        printf("Initial Matrix: \n");
        PrintMatrix(pMatrix, Size, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    for (int i=0; i<ProcNum; i++) {
        if (ProcRank == i) {
            printf("\nProcRank = %d \n", ProcRank);
            // fprintf(" Matrix Stripe:\n");
            PrintMatrix(pProcRows, RowNum, Size);
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
}

// Function for simple setting the grid node values
void DummyDataInitialization (double* pMatrix, int Size) {
    int i, j; // Loop variables
    double h = 1.0 / (Size - 1);
    // Setting the grid node values
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
                pMatrix[i*Size+j] = 100;
            else
                pMatrix[i*Size+j] = 0;
    }
}

// Function for memory allocation and initialization of grid nodes
void ProcessInitialization (double* &pMatrix, double* &pProcRows,int &Size,
    int &RowNum, double &Eps) {

```

```

int RestRows; // Number of rows, that haven't been distributed yet
// Setting the grid size
if (ProcRank == 0) {
    do {
        printf("\nEnter the grid size: ");
        scanf("%d", &Size);
        if (Size <= 2) {
            printf("\n Size of grid must be greater than 2! \n");
        }
        if (Size < ProcNum) {
            printf("Size of grid must be greater than"
                "the number of processes! \n ");
        }
    }
    while ( (Size <= 2) || (Size < ProcNum));

    // Setting the required accuracy
    do {
        printf("\nEnter the required accuracy: ");
        scanf("%lf", &Eps);
        printf("\nChosen accuracy = %lf", Eps);
        if (Eps <= 0)
            printf("\nAccuracy must be greater than 0!\n");
    }
    while (Eps <= 0);
}
MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Define the number of matrix rows stored on each process
RestRows = Size;
for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
RowNum = (RestRows-2)/(ProcNum-ProcRank)+2

// Memory allocation
pProcRows = new double [RowNum*Size];
// Define the values of initial objects' elements
if (ProcRank == 0) {
    // Initial matrix exists only on the pivot process
    pMatrix = new double [Size*Size];
    // Values of elements are defined only on the pivot process
    DummyDataInitialization(pMatrix, Size);
}
}

// Function for exchanging the boundary rows of the process stripes
void ExchangeData(double* pProcRows, int Size, int RowNum) {
    MPI_Status status;
    int NextProcNum = (ProcRank == ProcNum-1)? MPI_PROC_NULL : ProcRank + 1;
    int PrevProcNum = (ProcRank == 0)? MPI_PROC_NULL : ProcRank - 1;
    // Send to NextProcNum and receive from PrevProcNum
    MPI_Sendrecv(pProcRows + Size * (RowNum - 2), Size, MPI_DOUBLE,
        NextProcNum, 4, pProcRows, Size, MPI_DOUBLE, PrevProcNum, 4,
        MPI_COMM_WORLD, &status);
    // Send to PrevProcNum and receive from NextProcNum
    MPI_Sendrecv(pProcRows + Size, Size, MPI_DOUBLE, PrevProcNum, 5,
        pProcRows + (RowNum - 1) * Size, Size, MPI_DOUBLE, NextProcNum, 5,
        MPI_COMM_WORLD, &status);
}

// Function for the parallel Gauss - Seidel method
void ParallelResultCalculation (double *pProcRows, int Size, int RowNum,
    double Eps, int &Iterations) {

```

```

double ProcDelta,Delta;
Iterations=0;
do {
    Iterations++;
    // Exchanging the boundary rows of the process stripe
    ExchangeData(pProcRows, Size,RowNum);

    // The Gauss-Seidel method iteration
    ProcDelta = IterationCalculation(pProcRows, Size, RowNum);

    // Calculating the maximum value of the deviation
    MPI_Allreduce(&ProcDelta, &Delta, 1,MPI_DOUBLE, MPI_MAX,
        MPI_COMM_WORLD);
} while ( Delta > Eps);
}

// Function for gathering the result vector
void ResultCollection(double *pMatrix, double* pProcResult, int Size,
    int RowNum) {
    int *pReceiveNum; // Number of elements, that current process sends
    int *pReceiveInd; // Index of the first element of the received block
    int RestRows = Size;
    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;
    RowNum = (Size-2)/ProcNum+2;
    pReceiveNum[0] = RowNum*Size;
    for ( i=1; i < ProcNum; i++){
        RestRows = RestRows - RowNum + 1;
        RowNum = (RestRows-2)/(ProcNum-i)+2;
        pReceiveNum[i] = RowNum*Size;
        pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1]-Size;
    }

    // Gather the whole result vector on every processor
    MPI_Allgatherv(pProcRows, pReceiveNum[ProcRank], MPI_DOUBLE, pMatrix,
        pReceiveNum, pReceiveInd, MPI_DOUBLE, MPI_COMM_WORLD);

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

// Function for the serial Gauss - Seidel method
void SerialResultCalculation(double *pMatrixCopy, int Size, double Eps,
    int &Iter){
    int i, j; // Loop variables
    double dm, dmax,temp;
    Iter = 0;
    do {
        dmax = 0;
        for (i = 1; i < Size - 1; i++)
            for(j = 1; j < Size - 1; j++) {
                temp = pMatrixCopy[Size * i + j];
                pMatrixCopy[Size * i + j] = 0.25 * (pMatrixCopy[Size * i + j + 1] +
                    pMatrixCopy[Size * i + j - 1] +
                    pMatrixCopy[Size * (i + 1) + j] +
                    pMatrixCopy[Size * (i - 1) + j]);
                dm = fabs(pMatrixCopy[Size * i + j] - temp);
            }
    } while (dmax > Eps);
}

```



```

        if (dmax < dm) dmax = dm;
    }
    Iter++;
}
while (dmax > Eps);
}

// Function to copy the initial data
void CopyData(double *pMatrix, int Size, double *pSerialMatrix) {
    copy(pMatrix, pMatrix + Size, pSerialMatrix);
}

// Function for testing the computation result
void TestResult(double* pMatrix, double* pSerialMatrix, int Size,
    double Eps) {
    int equal = 0; // =1, if the matrices are not equal
    int Iter;

    if (ProcRank == 0) {
        SerialResultCalculation(pSerialMatrix, Size, Eps, Iter);
        for (int i=0; i<Size*Size; i++) {
            if (fabs(pSerialMatrix[i]-pMatrix[i]) >= Eps)
                equal = 1; break;
        }
        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 setting the grid node values by a random generator
void RandomDataInitialization (double* pMatrix, int Size) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    // Setting the grid node values
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))
                pMatrix[i*Size+j] = 100;
            else
                pMatrix[i*Size+j] = rand()/double(1000);
    }
}

void main(int argc, char* argv[]) {
    double* pMatrix; // Matrix of the grid nodes
    double* pProcRows; // Stripe of the matrix on current process
    double* pSerialMatrix; // Result of the serial method
    int Size; // Matrix size
    int RowNum; // Number of rows in matrix stripe
    double Eps; // Required accuracy
    int Iterations; // Iteration number
    double currDelta, delta;

    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 - Seidel algorithm \n");
    fflush(stdout);
}
// Process initialization
ProcessInitialization (pMatrix,pProcRows,Size,RowNum,Eps);

// Creating the copy of the initial data
if (ProcRank == 0) {
    pSerialMatrix = new double[Size*Size];
    CopyData(pMatrix, Size, pSerialMatrix);
}

// Data distribution among the processes
DataDistribution(pMatrix, pProcRows, Size,RowNum);

// Paralle Gauss-Seidel method
ParallelResultCalculation(pProcRows, Size,RowNum,Eps, Iterations);
//TestDistribution(pMatrix, pProcRows, Size,RowNum);

// Gathering the calculation results
ResultCollection(pProcRows, pMatrix, Size, RowNum);
TestDistribution(pMatrix, pProcRows, Size,RowNum);

// Printing the result
printf("\n Iter %d \n", Iterations);
printf("\nResult matrix: \n");
if (ProcRank==0) {
    //TestResult(pMatrix,Size,pMatrixCopy,Eps);
    PrintMatrix(pMatrix,Size,Size);
}

// Process termination
if (ProcRank == 0) delete []pSerialMatrix;
ProcessTermination(pMatrix, pProcRows);
MPI_Finalize();
}

```