Learning Lab 5: Parallel Algorithms of Graph Processing

Mathematical models in the form of graphs are widely used for modeling various phenomena, processes and systems. As a result, many theoretical and applied problems may be solved by means of various procedures of graph model analysis. It is possible to select a set of typical algorithms of graph processing among all those procedures. The problems of graph theory, modeling algorithms, analizing and solving problems based on graphs are discussed in a number of books – see Section 11 "Parallel Algorithms of Graph Processing" of the training materials.

Lab Objective

The objective of the lab is to develop a parallel program, which solves the problem of searching the shortest path with the use of Floyd algorithm. The lab assignments include:

- Exercise 1 – State the shortest path problem.
- Exercise 2 – Code the serial Floyd program.
- Exercise 3 – Develop the parallel Floyd algorithm.
- Exercise 4 – Code the parallel Floyd program.

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

The lab students are assumed to be familiar with the related section of the training material: Section 4 "Parallel programming with MPI", Section 6 "Principles of parallel method development" and Section 11 "Parallel Algorithms of Graph Processing". Besides, the preliminary lab "Parallel programming with MPI" and Lab "Parallel algorithm of matrix vector multiplication" are assumed to have been done.

Exercise 1 – State the Shortest Path Problem

In order to do this Exercise it is necessary to study the Floyd algorithm.

Let ***G*** be a graph

$$G = (V, R),$$

for which the set ***V*** of vertices v_i, $1 \le i \le n$, is defined , and the set of arcs

$$r_j = (v_{s_j}, v_{t_j}),\ 1 \le j \le m,$$

is defined by the set ***R***. Generally, the arcs may be assigned certain numerical characteristics (weights) w_j, $1 \le j \le m$ (*the weighted graph*). An example of the weighted graph is given in Figure 5.1.

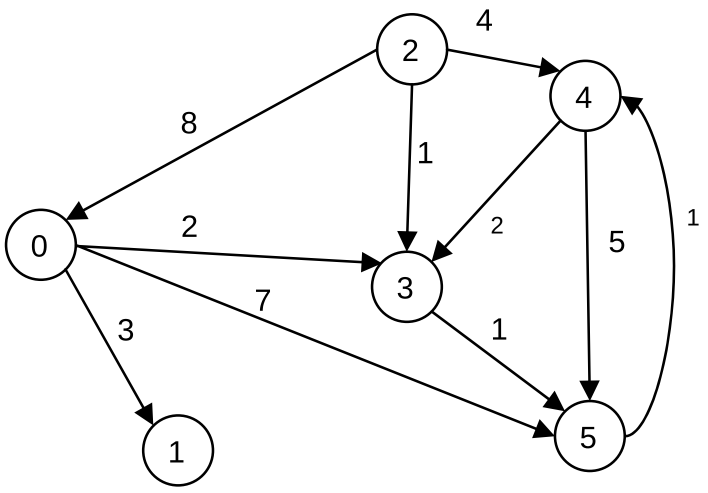


Figure. 5.1. Example of the Weighted Oriented Graph

Presenting dense graphs, almost all the nodes of which are linked by arcs (i.e. $m \sim n^2$), may be efficiently described by means of the adjacency matrix

$$A = (a_{ij}),\ 1 \le i,\ j \le n,$$

the nonzero values of which correspond to the arcs of the graph

$$a_{ij} = \begin{cases} w(v_i, v_j), & \text{if} \quad (v_i, v_j) \in R, \\ 0, & \text{if} \quad i = j, \\ \infty, & \text{otherwise.} \end{cases}$$

(the infinity sign is used in a corresponding position to denote the absence of an arc between the vertices in the adjacency matrix. In computations the infinity sign may be replaced by, for instance, a negative number). For instance, the adjacency matrix, which corresponds to the graph in Figure 5.1, is shown in Figure5.2

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Figure. 5.2. The Adjacency Matrix for the Graph in Figure 5.1

We will discuss further the aspects of parallel algorithm development on graphs using the problem of searching the shortest paths among all the pairs of destination vertices for the given graph *G*. As a practical example we may use the problem of working out the transport route between various cities, if the distances between them are given, and all the problems alike.

The initial information for *the problem of searching the shortest paths* is the weighted graph $G = (V, R)$, which contains *n* vertices ($|V| = n$). Each arc of the graph is assigned non-negative weight. The graph is assumed to be oriented. If there is an arc from the vertex *i* to the vertex *j*, it should not be supposed that there is an arc from *j* to *i*. In case when the vertices are connected by inverse arcs, the weights assigned to the arcs may not coincide.

As a method for solving the problem of searching all the shortest paths, we will further use Floyd algorithm (see Section 11 "Parallel Algorithms of Graph Processing" of the training materials).

Generally, the algorithm may be presented in the following way:

```
// Serial Floyd algorithm
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      A[i,j] = min(A[i,j],A[i,k]+A[k,j]);
}
```

(the implementation of operation *min* choosing the minimum value must be performed with regard to the method of describing nonexistent graph arcs in the adjacency matrix). As it can be noted, while the algorithm is performed the adjacency matrix *A* changes. After the termination of computations, the required result (the length of the minimum path) will be stored in matrix *A*.

As it can be noted, the complexity of the Floyd algorithm is of n^3 order.

Additional information on the Floyd algorithm may be obtained in Section 11 "Parallel Algorithms of Graph Processing" of the training materials.

Exercise 2 – Code the Serial Floyd Program

The Exercise implies the necessity to develop the serial Floyd algorithm. The initial version of the program to be developed is given in the project *SerialFloyd,* which contains a part of the initial code. In the course of doing the Exercise, it is necessary to add the available program version by the initial data input operations, the Floyd algorithm implementation and testing the correctness of the program results.

Task 1 – Open the project SerialFloyd

Open the project *SerialFloyd* 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\ SerialFloyd** in the dialog window **Open Project,**
- Make the double click on the file **SerialFloyd.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 the double click on the file of the initial code *SerialFloyd.cpp*, as it is shown in Figure 5.3. After that, the code, which has to be enhanced, will be opened in the workspace of Visual Studio.

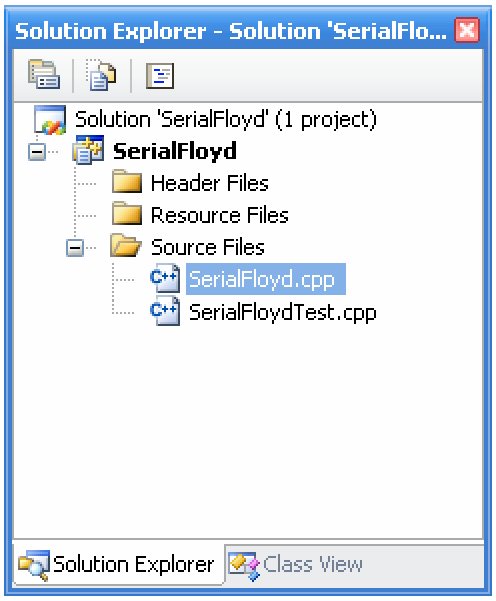


Figure. 5.3. Opening the File SerialFloyd.cpp

The file *SerialFloyd.cpp* provides access to the necessary libraries and also contains the initial version of the head function of the program – the function *main*. The available program variant contains the declaration of variables and printout of the initial program message. File *SerialFloydTest.cpp* contains the developed variant of the test functions, which will be necessary for testing the correctness of the program to be developed.

Let us consider the variables, which are used in the main function of the application. The variable *pMatrix* is the adjacency matrix. After the Floyd algorithm has finished its execution the variable will point to the result of the algorithm. The variable *Size* defines the adjacency matrix size (or, which is the same, the number of vertices).

```
  int *pMatrix;   // Adjacency matrix
  int  Size;      // Size of adjacency matrix
```

It should be noted that in order to store the matrix *pMatrix* we should use a one-dimensional array, where the matrix is stored element by element. 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 initial message input is provided by means of the following program lines:

```
  printf("Serial Floyd 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 program start the following message will appear in the command console:

"`Serial Floyd algorithm`".

In order to exit the program, press any key.

Task 2 – Input the Number of Vertices

In order to set the initial data of the serial Floyd algorithm, we will develop the function *ProcessInitialization*. This function is intended for the initialization of all the variables used in the program, in particularly for the input of the number of the processed vertices (the adjacency matrix size), the allocation of the memory for the adjacency matrix and for filling this memory with the initial values. Thus, the function should have the following heading:

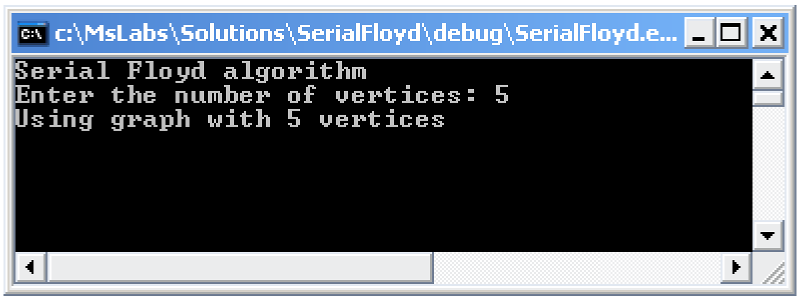
```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pMatrix, int& Size);
```

Let us enhance the program with the code, which allows to input the adjacency matrix size (to set the value of the variable *Size*) and check the correctness of the input. For this purpose add the bold marked code given below to the function *ProcessInitialization* and add the call of the function to the main function:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int& Size) {
  do {
    printf("Enter the number of vertices: ");
    scanf("%d", &Size);
    if(Size <= 0)
      printf("The number of vertices should be greater than zero\n");
  } while(Size <= 0);

  printf("Using graph with %d vertices\n", Size);
}
```

The user is given the opportunity to enter the number of vertices in the adjacency matrix, which is read from the standard input stream *stdin* and stored in the integer variable *Size*. The value of the variable *Size* is tested further (it should be greater than zero), in case of the invalid input the latter is repeated and at last the input value is printed out (Figure 5.4).

Compile and run the application. Make sure that in case of inputting a negative value of the variable *Size*, the program gives the diagnostic message: "`The number of vertices should be greater than zero`".



```
Serial Floyd algorithm
Enter the number of vertices: 5
Using graph with 5 vertices
```

Figure. 5.4. Setting the Number of the Vertices

Then it is necessary to add memory allocation for the adjacency matrix to the function *ProcessInitialization* and initialize the matrix. For this purpose the bold marked code must be added to the function *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int& Size) {
  <…>
  // Allocate memory for the adjacency matrix
  pMatrix = new int[Size * Size];

  // Data initalization
  DummyDataInitialization(pMatrix, Size);
}
```

The immediate adjacency matrix initialization will be executed by a special function. At the first stage it is possible to use the simple method of implementation, containing the set of data, the correctness of which can be easily checked. There is a stub version of this function in the file (*DummyDataInitialization*), it is only necessary to add the following code to the function:

```
// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {
  for(int i = 0; i < Size; i++)
    for(int j = i; j < Size; j++) {
      if(i == j) pMatrix[i * Size + j] = 0;
      else
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;

      pMatrix[j * Size + i] = pMatrix[i * Size + j];
  }
}
```

As it can be seen from the given code, this function fills the adjacency matrix in rather a simple way: the main matrix diagonal is filled with zeros, the elements located in the first matrix row are filled the column number, the rest of the elements located higher that the main diagonal are filled with minus ones (which signifies the absence of arcs among the vertices). The elements located lower than the main diagonal are filled with the values of the elements symmetric about the diagonal. Thus, if the user has entered the adjacency matrix size equal to 4, the matrix will be defined the following way:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & -1 & -1 \\ 2 & -1 & 0 & -1 \\ 3 & -1 & -1 & 0 \end{pmatrix}$$

(setting the matrix by means of a random number generator will be discussed in Task 5).

This way of setting the elements simplifies the testing of the program execution as it sets the graph with a simple structure: the vertex 0 is connected with the bidirectional arcs with all the rest of the vertices in the graph, the weight of the arc is equal to the number of the vertex, to which the vertex number 0 is connected by the arc (see Figure 5.5).

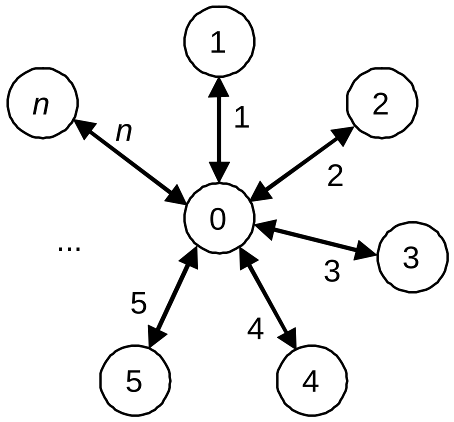


Figure. 5.5. Test Graph for the Floyd Algorithm

Compile and run the application (execute the command **Rebuild Solution** in the menu **Build).** If there are any errors in the application, compare your code to the code given in the Exercise. After all the debugging is completed, run the application.

Let us develop one more function, which will further help to control the program execution. This is the function of the formatted adjacency matrix output *PrintMatrix*. The preliminary version of the function may be obtained in the file *SerialFloydTest.cpp*. It is possible to pass over to editing the file on the analogy with choosing the file *SerialFloyd.cpp* in Task 1. The one-dimensional array, where the adjacency matrix is stored rowwise and the matrix sizes both vertically (the number of rows *RowCount*) and horizontally (the number of columns *RowCount*) are given as arguments to the function of the formatted matrix output *PrintMatrix*:

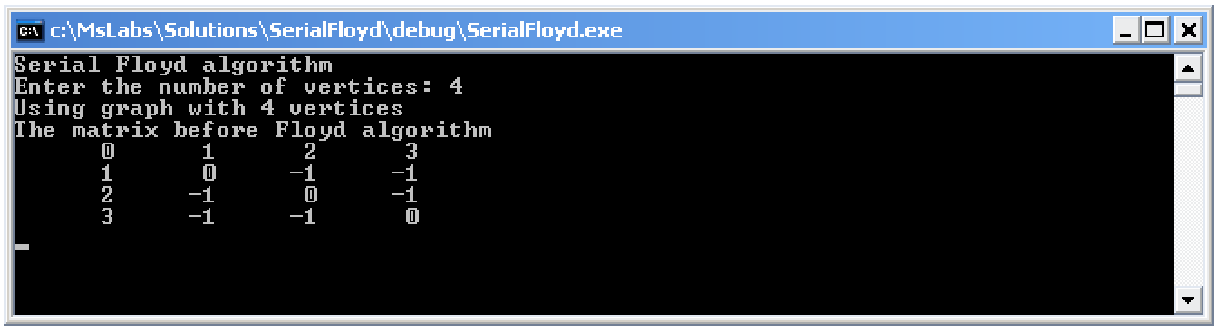
```
// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
    for(int j = 0; j < ColCount; j++)
      printf("%7d", pMatrix[i * ColCount + j]);
    printf("\n");
  }
}
```

Let us add the call of the function *PrintMatrix* to the function *main* of the application immediately after the call of the function *ProcessInitialization.* It should be done in order to test the correctness of setting the initial data:

```
...
```

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

  printf("The matrix before Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);
...
```

Compile and run the application. Make sure that the data input is performed according to the above-described rules (See Figure 5.6). Run the application several times, set different adjacency matrix sizes.



```
Serial Floyd algorithm
Enter the number of vertices: 4
Using graph with 4 vertices
The matrix before Floyd algorithm
      0       1       2       3
      1       0      -1      -1
      2      -1       0      -1
      3      -1      -1       0
```

Figure. 5.6. The Result of the Program Execution on the Completion of Task 2

Task 3 –Terminatee the Program Execution

In this Task before the implementation of the Floyd algorithm we should develop a function for correct program termination. 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 adjacency matrix *pMatrix* must be given to the function *ProcessTermination* as argument:

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

The call of the function *ProcessTermination* should be added to the main function immediately before the termination of the main function *main*, when the adjacency matrix is not needed any longer:

```
...
  // Process termination
  ProcessTermination(pMatrix);

  return 0;
}
```

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

Task 4 – Implement the Floyd Algorithm

Let us develop the main computational part of the program. In order to execute the Floyd algorithm, we will develop the function *SerialFloyd*, which receives the adjacency matrix *pMatrix* of size *Size* as input parameters.

According to the algorithm given in Exercise 1 the code of the function should be as follows:

```
// Function for the serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];
          pMatrix[i * Size + j] = Min(t1, t2);
      }
}
```

As it has already been noted, the function of choosing the minimum value should be in agreement with the selected way of designating the infinity. The value -1 has been chosen to denote the infinitely in this lab. Let us implement this function and call it *Min*:

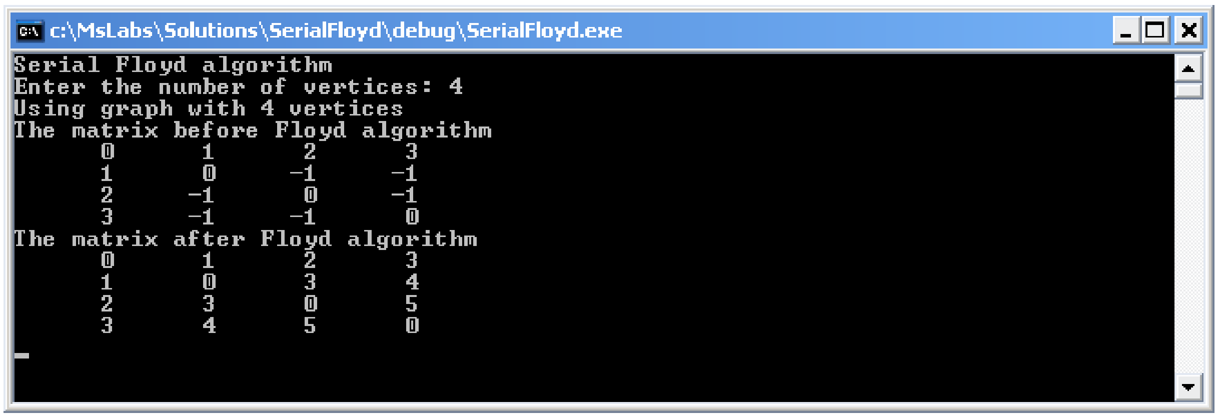
```
int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}
```

Let us call the function *SerialFloyd* from the main program. To test the correctness of the algorithm execution, we will print out the obtained matrix:

```
  <…>

  // Serial Floyd algorithm
  SerialFloyd(pMatrix, Size);

  printf("The matrix after Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);
...
```

Compile and run the application. Analyze the result of the program execution. If the algorithm has been implemented correctly, the result matrix must have the following structure: all the elements located on the main diagonal are equal to zero; the remaining elements are equal to the sum of the row number and the column number of the element; the rows and the columns are enumerated starting with zero. Thus, for the matrix described in Task 2 the result should be the following (see also Figure 5.7):

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 4 \\ 2 & 3 & 0 & 5 \\ 3 & 4 & 5 & 0 \end{pmatrix}$$



```
Serial Floyd algorithm
Enter the number of vertices: 4
Using graph with 4 vertices
The matrix before Floyd algorithm
      0      1      2      3
      1      0     -1     -1
      2     -1      0     -1
      3     -1     -1      0
The matrix after Floyd algorithm
      0      1      2      3
      1      0      3      4
      2      3      0      5
      3      4      5      0
```

Figure. 5.7. The Result of the Floyd Algorithm Execution for Test Matrix Consisting of 4 Elements

Task 5 – Carry out the Computational Experiments

In order to test the further speed up of the parallel program execution, it is necessary to carry out experiments on the computation of the execution time for the serial program. It is reasonable to analyze the execution time of the algorithm for large enough data amounts. The data will be set by means of random number generator. For this purpose we will develop the function of element setting *RandomDataInitialization* (the random number generator is initialized by the current clock):

```
// Function for initializing the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1);
      }
      else
        pMatrix[i * Size + j] = 0;
}
```

We will call this function instead of the previously developed function *DummyDataInitialization* in the function *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int& Size) {
..<…>
  // Allocate memory for the adjacency matrix
  pMatrix = new int[Size * Size];

  // Data initalization
  //DummyDataInitialization(pMatrix, Size);
  RandomDataInitialization(pMatrix, Size);
}
```

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

In order to determine the time, you should add the call of the functions, which allow you to find out the execution time of the program or its part, to the obtained program. We will use this function, as we have previously done in other labs:

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

To compute the execution time you will need three additional variables, which must be declared in the function *main*:

```
int main(int argc, char *argv[]) {
  int *pMatrix = 0;     // Adjacency matrix
  int  Size    = 0;     // Size of adjacency matrix

  time_t start, finish;
  double duration = 0.0;
...
```

Let us add the computation and output of the Floyd algorithm execution time to the program code. For this purpose let us clock in before and after the call of the function *SerialFloyd*:

```
..  <…>

  start = clock();
  // Parallel Floyd algorithm
  SerialFloyd(pMatrix, Size);
  finish = clock();

  printf("The matrix after Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  duration = (finish - start) / double(CLOCKS_PER_SEC);
  printf("Time of execution: %f\n", duration);
...
```

Compile and run the application. In order to carry out the computational experiments of large amounts of data, switch off the data print before and after the Floyd algorithm (transforming the corresponding code lines into comments). Carry out the computational experiments and fill the third column of the following table:

Table 5.1. The Results of the Computational Experiments for the Floyd Algorithm

Test Number	The Number of Vertices in the Graph	Algorithm Execution Time (sec.)
1	10	
2	500	
3	600	
4	700	
5	800	
6	900	
7	1,000	

Use the results of the experiments and estimate the nature of dependence of the Floyd algorithm execution time against the number of graph vertices. Make sure that this dependence is cubic (if the amount of data is increased twice, then the algorithm execution time increases eight times etc.).

Let us estimate the Floyd algorithm complexity (see Section 11 "Parallel Algorithms of Graph Processing" of the training materials). The algorithm assumes the execution of Size^3 operations:

$$T_1 = Size^3 \cdot \tau , \qquad (5.1)$$

where τ is the execution time of the operation of adjacency matrix element comparison carried out by the algorithm.

Complete the table of comparison of the experiment execution time to the time obtained according to formula (5.1). In order to compute the execution time of the basic comparison operation, we will use the following method: let us choose one of the experiments as a pivot (for instance, the experiment on processing the graph containing 800 vertices) and divide the execution time of this experiment by the number of the executed operations. Thus, we will compute the execution time of the operation τ. Then we will use this value and compute the theoretical execution time for all the other experiments.

Carry out the above-described computations and complete the Table 5.2.

Table 5.2. The Comparison of the Experimental and the Theoretical Floyd Algorithm Execution Time

The Execution Time of the Basic Comparison Operation τ (sec):			
Test Number	The Number of Vertices in the Graph	Execution Time (sec)	Theoretical Time (sec)
1	10		
2	500		
3	600		
4	700		
5	800		
6	900		
7	1,000		

It should be noted that the basic comparison operation execution time depends generally on the number of the processed vertices. This dependence can be explained by the computer architecture properties. If the amount of vertices is small, the data can be fully located in cache memory of the processor, and the access to the memory is fast. If the algorithm operates with medium size data, which can be fully located in RAM, but not in cache, the execution time for a basic comparison operation will be somewhat bigger, as the access time to the RAM is greater than the access time to cache memory. If the amount of data is large enough and they can not be located in the RAM, the swap file mechanism is involved. The data is stored on the external storage, and read and write time for the external storage exceeds significantly the recording time to the RAM area. Thus, choosing an experiment as a pivot (the experiment, for which the basic comparison operation execution time is calculated), we should be oriented at some average situation. That is why we have chosen the experiment on processing the graph with 800 vertices as a pivot.

Exercise 3 –Develop the Parallel Floyd Algorithm

It order to do the Exercise you should study the principles of parallelizing the Floyd algorithm. For this purpose you should decompose the problem, analyze the information interactions among the subtasks, and distribute subtasks among the processes.

Subtask Definition

As the general scheme of the Floyd algorithm suggests, the main computations in solving the problem of searching the shortest path consists in choosing the minimum values. It is not worthwhile to parallelize this rather simple operation, as it will not speed up the computation significantly. It is more efficient to update the values of matrix *A* simultaneously, as it will make parallel computations more effective (the efficiency of this approach to parallelizing is shown in Section 11 "Parallel Algorithms of Graph Processing" of the training materials).

As a result, the necessary conditions for parallel computations are provided. Thus, the update operation of matrix element may be used as the basic computational subtask (the indices of the updated elements will be used to point to the subtasks).

Choosing the Information Dependencies

Computations in the subtasks become possible only if each subtask (i,j) contains the elements A_{ij}, A_{ik}, A_{kj} of the matrix *A*, which are necessary for computations. To eliminate data doubling we will place the only element A_{ij} in the subtask (i,j). Obtaining all the other necessary values may be provided only by means of data transmission. Thus, each element A_{kj} of the row *k* of the matrix *A* must be transmitted to all the subtasks (k,j), $1 \le j \le n$, and each element A_{ik} of the column *k* of the matrix *A* must be transmitted to all the subtasks (i,k), $1 \le i \le n$, - see Figure 5.8.

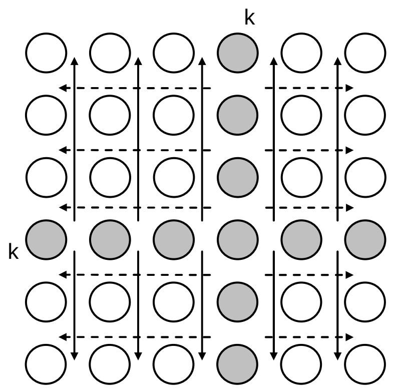


Figure. 5.8. The Information Dependencies of the Basic Computational Subtasks (the arrows show the direction of exchanging values at iteration *k*)

Scaling and Distributing the Subtasks among Processors

As a rule, the number of available processors *p* is considerably smaller than the number of basic subtasks n^2 ($p << n^2$). The use of block-striped scheme of the matrix *A* partitioning is a possible way to aggregate the computations. This approach corresponds to uniting in one basic subtask the computations connected with updating the elements of one or several rows (horizontal partitioning) or columns (vertical partitioning) of matrix *A*. These two partition types are practically equal. With regard to the fact that for the algorithmic language C arrays are located rowwise, we will further analyze only partitioning the matrix *A* into horizontal stripes.

It should be noted that in case of this method of data partitioning, it is necessary to transmit among the subtasks only the elements of one of matrix *A* rows. The network topology for efficient execution of this communication operation is a hypercube or a complete graph.

Exercise 4 – Code the Parallel Floyd Program

In order to perform the Tasks, you will have to implement the parallel Floyd algorithm. This Exercise is aimed at:

- Enhancing the practical knowledge on parallel program development,
- Developoffing the parallel program for implementing the Floyd algorithm.

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

- Initialization of the MPI environment,
- The main part of the program, where the necessary algorithm of solving the stated problem is implemented, and the message exchange among the processes executed in parallel is carried out,
- The termination of MPI program.

Lab students are supposed to be familiar with Section 4 "Parallel Programming with MPI".

Task 1 – Open the Project ParallelFloyd

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

After the project has been opened in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *ParallelFloyd.cpp*, as it is shown in Figure 5.9. After that, the code, which is to be enhanced, will be opened in the workspaceof Visual Studio.

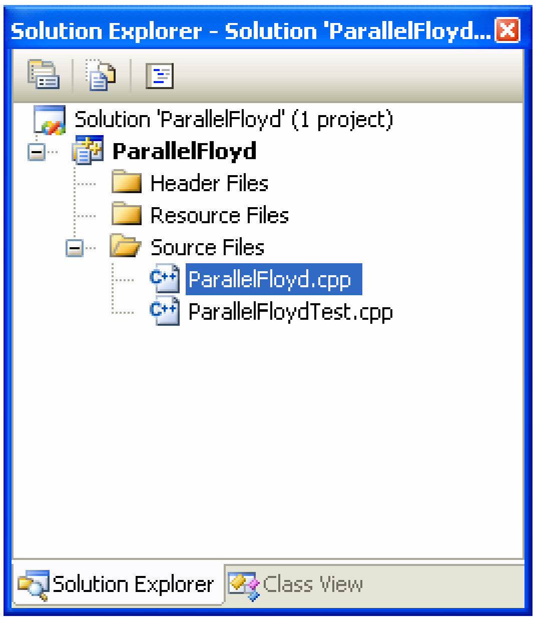


Figure. 5.9. Opening the File ParallelFloyd.cpp

The main function of the parallel algorithm to be developed, which contains the declaration of the necessary variables, is located in the file *ParallelFloyd.cpp***.** The following functions copied from the serial are also located in the file *ParallelFloyd.cpp*: *DummyDataInitialization*, *RandomDataInitialization* and *Min*. Besides, you can also see the function *PrintMatrix* responsible for test data output in the file *ParallelFloydTest.cpp* (the purpose of the function is considered in detail 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 process initialization (*ProcessInitialization*) and process termination (*ProcessTermination*) are also located there.

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

```
"Parallel Floyd algorithm"
```

is output into the command console.

Task 2 –Initialize and Terminate the Parallel Program

As is has been noted previously, there should be the header file "mpi.h" in the parallel program

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include <cmath>
#include <algorithm>
```

```
#include <mpi.h>
```

It is necessary to initialize the environment of the MPI program execution in the main function, to determine the number of processes available for MPI program, to determine the process rank in communicator MPI_COMM_WORLD, and also to set global variables for storing these values (*ProcNum* and *ProcRank* correspondingly). Add the following selected code:

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

void main(int argc, char* argv[]) {
  int *pMatrix;     // Adjacency matrix
  int  Size;        // Size of adjacency matrix
  int *pProcRows;   // Process rows
  int  RowNum;      // Number of process rows

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

  if(ProcRank == 0)
    printf("Parallel Floyd 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 then **Run**,
2. Type the name of the program **cmd** in the dialog window, which appears on the screen (Figure 5.10)

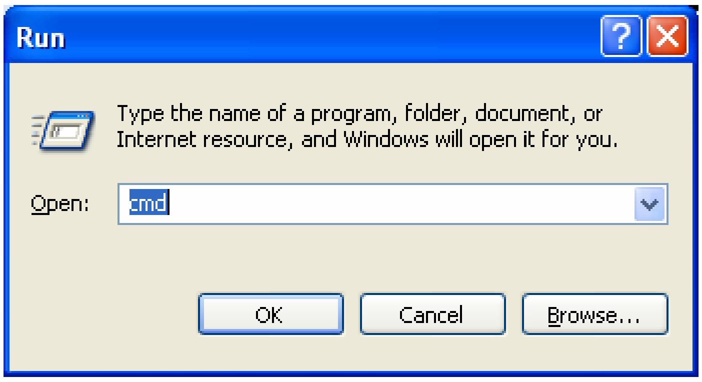
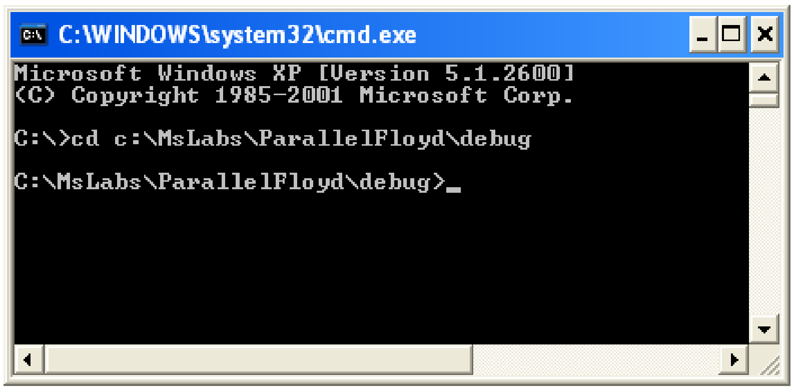

Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
cmd
OK
Cancel
Browse...

Figure. 5.10. The Start of the Command Prompt

In the command line go to the folder, which contains the developed program (Figure 5.11):

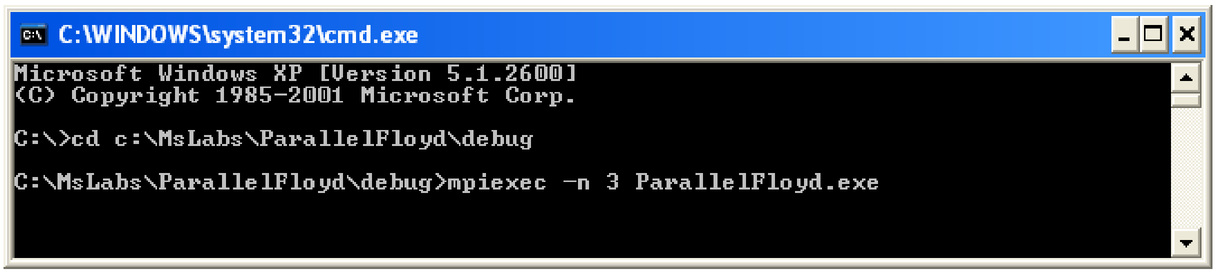


```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\>cd c:\MsLabs\ParallelFloyd\debug

C:\MsLabs\ParallelFloyd\debug>_
```

Figure. 5.11. Setting the Folder, which Contains the Parallel Program

On the analogy to the program start from the previously labs, type the command (Figure 5.12) in order to execute the program using 4 processes:

```
mpiexec –n 3 ParallelFloyd.exe
```



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\>cd c:\MsLabs\ParallelFloyd\debug

C:\MsLabs\ParallelFloyd\debug>mpiexec –n 3 ParallelFloyd.exe
```

Figure. 5.12. Starting the Parallel Program

Make sure that initial message

```
"Parallel Floyd algorithm"
```

is output to the command console.

Task 3 – Input the Initial Data

Let us implement the data input. We should enhance our program with the code, which provides setting the number of vertices of the processed graph and allocates memory for the adjacency matrix. As in the other labs, setting the initial data will be executed by one of the processes (let it be the process with the rank 0). Then according to the scheme of parallel computations given in Exercise 3, the adjacency matrix is distributed among all the processes so that each of them processes a continuous sequence (stripe) of data. It should be noted that the first version of the program to be developed is developed for the case when the number of vertices in the graph is divisible by the number of processes without remainder, i.e. the stripes on all the processes contain the same number of matrix rows. This number will be stored in the variable *RowNum*. The stripe of each of the processes will be stored in the variable *pProcRows*. As a result of the Floyd algorithm execution, each process obtains *RowNum* result matrix rows. Then the data should be collected on the root process again (the process with the rank 0).

In order to initialize the computations we will develop the function *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size,
  int& RowNum);
```

Thus, first we should input the amount of the number of vertices in the graph, i.e. set the value of the variable *Size*. In order to input this number it is necessary to implement the dialog with the user. As in the previous labs, we should check the correctness of the input value up. Add the bold marked code to the function *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size,
  int& RowNum) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    do {
      printf("Enter the number of vertices: ");
      scanf("%d", &Size);
      if(Size < ProcNum)
        printf("The number of vertices should be greater than "
          "the number of processes\n");
      if(Size % ProcNum != 0)
        printf("The number of vertices should be divisible by "
          "the number of processes\n");
    } while((Size < ProcNum) || (Size % ProcNum != 0));

    printf("Using the graph with %d vertices\n", Size);
  }
}
```

Now it is necessary to broadcast the number of vertices to the other processes. For this purpose we should use the function, which is familiar to those who have done the previous labs, 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):

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size,
  int& RowNum) {
  if(ProcRank == 0) {
    <…>
    printf("Using graph with %d vertices\n", Size);
  }

  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  RowNum = Size / ProcNum;
}
```

Add the call of the initialization function to the main function:

```
int main(int argc, char* argv[]) {
  <…>);
  if(ProcRank == 0)
    printf("Parallel Floyd algorithm program\n");

  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  MPI_Finalize();

  return 0;
}
```

Compile and run the application. Make sure that all the invalid situations are processed correctly. For this purpose, start the application several times setting the various number of parallel processes (by means of the utility **mpiexec**) and the various number of vertices in the processed graph.

After the number of vertices has been set, it is possible to allocate memory for the adjacency matrix and the stripes assigned to the processes. Add the bold marked code to the function *ProcessInitialization*:

```
  <…>
  // Allocate memory for the current process rows
  pProcRows = new int[Size * RowNum];

  if(ProcRank == 0) {
    // Allocate memory for the adjacency matrix
    pMatrix = new int[Size * Size];

    // Data initalization
    DummyDataInitialization(pMatrix, Size);
  }
}
```

It should be noted that for the adjacency matrix on the root process, we have used the function of data generation *DummyDataInitialization*, which was developed for the implementation of the serial application. Pay attention to the fact that the function fills the array with the predictable values, which allow us to easily test the program execution.

Task 4 –Terminate the Calculations

In order to terminate the parallel program correctly, 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 the adjacency matrix *pMatrix* was allocated on the root process; besides, memory was allocated on all the processes for the stripes *pProcRows*. These two arrays must be given to the function *ProcessTermination* as arguments:

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

The call of the termination function must be added to the function *main* immediately before the call of the function *MPI_Finalize*:

```
...
  // Process termination
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
}
```

The commands for printing the adjacency matrix on the root process should be added to the code of the main function (use the function *PrinMatrix*, which was implemented in the course of serial application development). Compile and run the application. Make sure that the initial data is being set correctly.

Task 5 – Distribute the Data among the Processes

In accordance with the parallel computation scheme given in the previous Exercise, the adjacency matrix must be distributed among the processes in equal stripes.

The function *DataDistribution* is responsible for data distribution. The adjacency matrix *pMatrix*, the number of graph vertices (the matrix size) and the process stripe *pProcRows*, and also the size of the stripe *RowNum* must be given to the function as arguments:

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

After that it is necessary to use the generalized operation of data transmission from one process to all processes (data distribution) and to distribute the adjacency matrix among the processes:

```
// Function for the data distribution among the processes
void DataDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Scatter(pMatrix, RowNum * Size, MPI_INT, pProcRows, RowNum * Size,
    MPI_INT, 0, MPI_COMM_WORLD);
}
```

Correspondingly, it is necessary to call the function *DataDistribution* from the main program right after the call of the initialization function *ProcessInitialization*:

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

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

Now let us check the correctness of the data distribution among the processes up. For this purpose we should print the initial adjacency matrix and the stripes 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. This function will be referred to as *TestDistribution*. For this purpose we should develop one more function for debugging print besides the previously developed function *PrintMatrix.* This function referred to as *ParallelPrintMatrix* will provide the sequential output of the matrix stripes by the processes with the use of the function *PrintMatrix.* Add the following code to the program being developed:

```
// Function for formatted output of all stripes
void ParallelPrintMatrix(int *pProcRows, int Size, int RowNum) {
  for(int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Proc rows:\n");
      PrintMatrix(pProcRows, RowNum, Size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

Finally, it is possible to implement the function *TestDistribution* using all the previously developed functions of debugging print the following way:

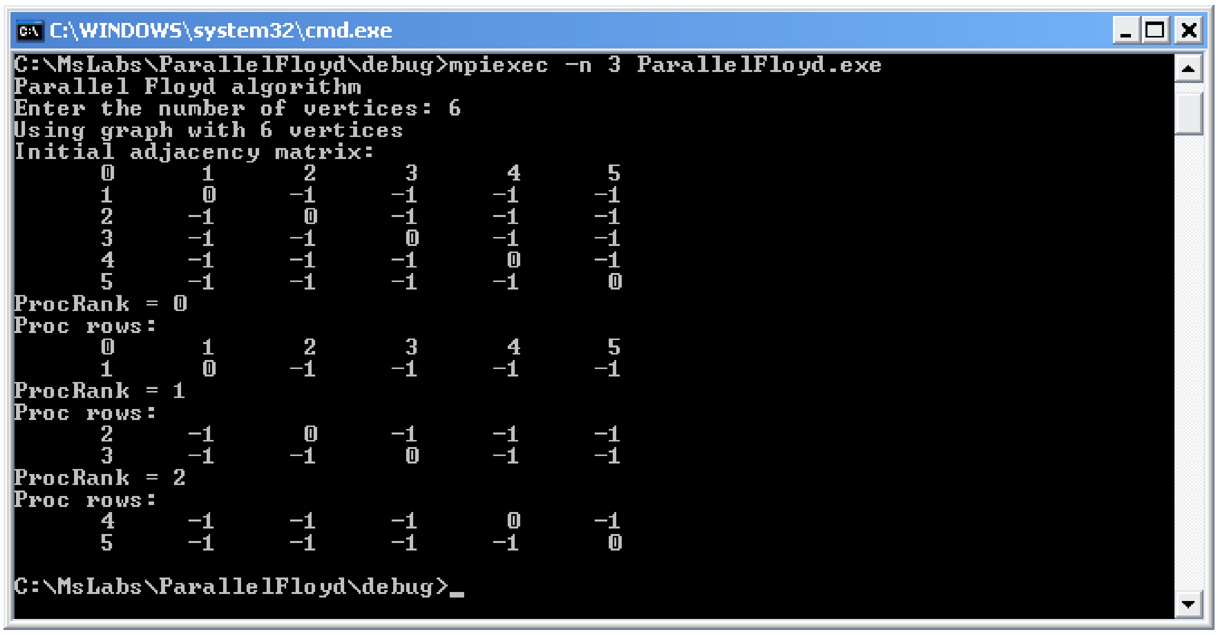
```
// Function for testing the data distribution
void TestDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial adjacency matrix:\n");
    PrintMatrix(pMatrix, Size, Size);
  }

  MPI_Barrier(MPI_COMM_WORLD);

  ParallelPrintMatrix(pProcRows, Size, RowNum);
}
```

The function *TestDistribution* resembles the previously developed functions, which have the analogous purpose: the root process prints all the data, then the parallel program processes print their data one after another (first the process with the rank 0, then the process with the rank 1 etc.)

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

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

  // Testing the distribution
  TestDistribution(pMatrix, pProcRows, Size, RowNum);
...
```

Compile the application. In case any errors are identified, check them comparing your code to the code given here. Start the application, which uses three processes, and set the number of vertices in the processed graph equal to 6. Make sure that data distribution is performed correctly (Figure 5.13).



```
C:\MsLabs\ParallelFloyd\debug>mpiexec -n 3 ParallelFloyd.exe
Parallel Floyd algorithm
Enter the number of vertices: 6
Using graph with 6 vertices
Initial adjacency matrix:
      0      1      2      3      4      5
      1      0     -1     -1     -1     -1
      2     -1      0     -1     -1     -1
      3     -1     -1      0     -1     -1
      4     -1     -1     -1      0     -1
      5     -1     -1     -1     -1      0
ProcRank = 0
Proc rows:
      0      1      2      3      4      5
      1      0     -1     -1     -1     -1
ProcRank = 1
Proc rows:
      2     -1      0     -1     -1     -1
      3     -1     -1      0     -1     -1
ProcRank = 2
Proc rows:
      4     -1     -1     -1      0     -1
      5     -1     -1     -1     -1      0

C:\MsLabs\ParallelFloyd\debug>_
```

Figure. 5.13. Data Distribution for the Parallel Program Using Three Processes

Task 6 – Implement the Parallel Floyd Algorithm

Let us implement the parallel Floyd algorithm in the course of several sequential stages. Each of the stages is simple enough and its correctness is easy to check.

Let us define the heading of the function, which implements the algorithm. It is necessary to have process stripe *pProcRows,* the matrix size *Size* and the stripe size *RowNum*. As a result, the developed function will have the following heading:

```
// Function for the parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum);
```

In accordance with the general scheme of the parallel Floyd algorithm it is necessary to carry out *Size* times the operation, which updates the adjacency matrix. To carry out this operation all the processes need the matrix row, the number of which coincides with the iteration number. This row does not have to be stored by the current process. It means that it is necessary to broadcast this row among the processes in the course of the operation execution. Besides, it is necessary to allocate memory for storing this row. Consequently, the program for the parallel Floyd algorithm will look the following way at the first stage:

```
// Function for the parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];

  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);
  }
  delete []pRow;
}
```

The function *RowDistribution* used above must use the row number *k* in order to find the process, to which the *k*-th adjacency matrix row belongs, and broadcast the row to the other processes. As previously, we will use the function *MPI_Bcast* for such kind of exchange:

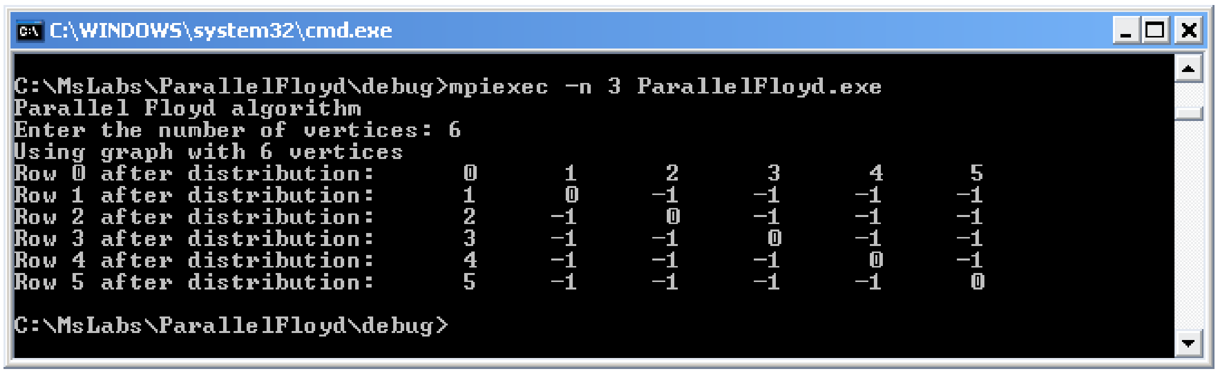
```
// Function for row broadcasting among all processes
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank = k / RowNum;  // Process rank with the row k
  int ProcRowNum  = k - ProcRowRank * RowNum; // Process row number

  if(ProcRowRank == ProcRank)
    // Copy the row to pRow array
    copy(&pProcRows[ProcRowNum*Size],&pProcRows[(ProcRowNum+1)*Size],pRow);

  // Broadcast row to all processes
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}
```

Let us test the developed part of the parallel Floyd algorithm. For this purpose we will use the debugging print, which outputs the row received by the process with the rank 0. Add the following code to the function *ParallelFloyd*:

```
...
  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    if(ProcRank == 0) {
      printf("Row %d after distribution:", k);
      PrintMatrix(pRow, Size, 1);
    }
  }
...
```

It should be noted that to print each row of the adjacency matrix the function *PrintMatrix* is used as the third parameter of this function is equal to 1 (this parameter determines the number of the matrix rows to be printed). As a result of the code execution the rows of the adjacency matrix must appear on the console (the structure of the test adjacency matrix was described in Task 2 of Exercise 2).

Transform the call of the function *TestDistribution* into the comment line in the function *main*, compile the application. Run the application and make sure that the data rows are distributed correctly. The program output for the test adjacency matrix of size 6x6 is given below in Figure 5.14.



```
C:\MsLabs\ParallelFloyd\debug>mpiexec -n 3 ParallelFloyd.exe
Parallel Floyd algorithm
Enter the number of vertices: 6
Using graph with 6 vertices
Row 0 after distribution:      0      1      2      3      4      5
Row 1 after distribution:      1      0     -1     -1     -1     -1
Row 2 after distribution:      2     -1      0     -1     -1     -1
Row 3 after distribution:      3     -1     -1      0     -1     -1
Row 4 after distribution:      4     -1     -1     -1      0     -1
Row 5 after distribution:      5     -1     -1     -1     -1      0

C:\MsLabs\ParallelFloyd\debug>
```

Figure. 5.14. The Debugging Print for Testing the Function *RowDistribution*

Task 7 – Implement the Floyd Algorithm Iterations

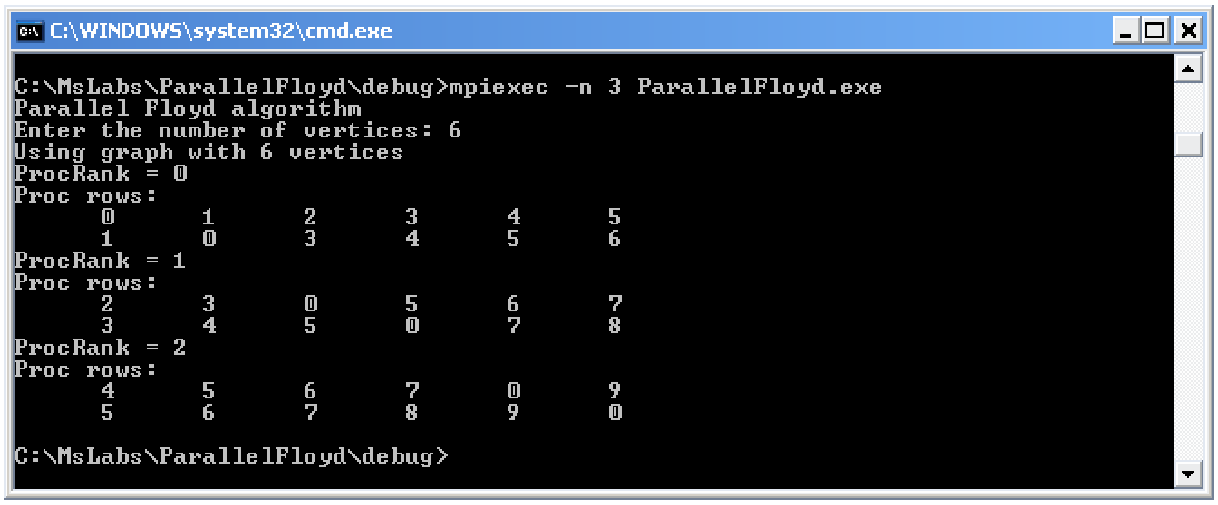
In accordance with the general scheme of the parallel Floyd algorithm, it is necessary to execute the adjacency matrix update after broadcasting the next adjacency matrix row among the processes. The function *ParallelFloyd* should look the following way:

```
// Function for the parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;

  for(int k = 0; k < Size; k++) {

    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Update adjacency matrix elements
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) &&
            (pRow     [j]            != -1)) {
          t1 = pProcRows[i * Size + j];
          t2 = pProcRows[i * Size + k] + pRow[j];
          pProcRows[i * Size + j] = Min(t1, t2);
      }
  }
  delete []pRow;
}
```

This stage should be tested as well as all the previous ones. Let us make use of the debugging print with the help of the function *ParallelPrintMatrix* again. Put comment signs before the previous calls of the function and add a new call of the function immediately after the call of the algorithm *ParallelFloyd*:

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

  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);
  ParallelPrintMatrix(pProcRows, Size, RowNum);
```

For the data set by means of the function *DummyDataInitialization*, the result is known beforehand and described in Task 4 of Exercise 2. Thus, for instance, if the size of the test adjacency matrix is equal to 6 and the application uses three processes, the result should be the following (Figure 5.15).



```
C:\MsLabs\ParallelFloyd\debug>mpiexec -n 3 ParallelFloyd.exe
Parallel Floyd algorithm
Enter the number of vertices: 6
Using graph with 6 vertices
ProcRank = 0
Proc rows:
       0       1       2       3       4       5
       1       0       3       4       5       6
ProcRank = 1
Proc rows:
       2       3       0       5       6       7
       3       4       5       0       7       8
ProcRank = 2
Proc rows:
       4       5       6       7       0       9
       5       6       7       8       9       0

C:\MsLabs\ParallelFloyd\debug>
```

Figure. 5.15. The Result of Testing the Parallel Floyd Algorithm Using Three Processes and the Number of Graph Vertices Equal to Six

Compile and run the application. Check the correctness of the obtained partial results setting different number of processes and different number of the test graph vertices.

Task 8 – Collect the Result Matrix

As the final stage we have to collect the obtained matrix on the root process (the process with the rank 0). It should be noted that this stage is not mandatory in case of algorithm execution, as the amount of data to be processed may appear to be so significant that it would be impossible to locate it in the RAM of a computer. In this lab this stage is discussed as an additional example of a training Exercise and also for the purpose of final comparison of the parallel and serial calculation results.

In order to collect the data let us develop the function *ResultCollection*, which consists practically of the call of the function *MPI_Gather*:

```
// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Gather(pProcRows, RowNum * Size, MPI_INT, pMatrix, RowNum * Size,
    MPI_INT, 0, MPI_COMM_WORLD);
```

The call of the function from the main program:

```
  <…>
  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);
  ParallelPrintMatrix(pProcRows, Size, RowNum);

  // Process data collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);
```

After the collection execution, add print of the obtained matrix by means of the function *PrintMatrix* on the root process of the parallel application to the code of the main application function. Compile and run the application. Check the correctness of the program execution.

Task 9 – Test the Parallel Program Correctness

After the completion of the program development, it is necessary to test the correctness of the program execution. Let us develop the function *TestResult* for this purpose. This function will compare the results of the serial program to the results of the parallel one. In order to execute the serial algorithm you may use the previously developed function *SeriaFloyd*, which is located in the file *ParallelFloydTest.cpp*.

To make the serial algorithm *SerialFloyd* operate the same data as the developed parallel algorithm *ParallelFloyd*, it is necessary to produce a copy of the data using the function *CopyMatrix* (which is also located in the file *ParallelFloyd.cpp*):

```
// Function for copying the matrix
void CopyMatrix(int *pMatrix, int Size, int *pMatrixCopy) {
  copy(pMatrix, pMatrix + Size * Size, pMatrixCopy);
}
```

In order to check the correctness, let us compare the results of the serial Floyd algorithm to the result of the developed parallel algorithm with the help of the function *CompareMatrices* which is also located in the file *ParallelFloydTest.cpp*:

```
// Function for comparing the matrices
bool CompareMatrices(int *pMatrix1, int *pMatrix2, int Size) {
  return equal(pMatrix1, pMatrix1 + Size * Size, pMatrix2);
}
```

Let us add the calls of these functions to the source code. It is necessary to declare the variable *pSerialMatrix* for storing the copy of the adjacency matrix, which parties processed in the serial Floyd algorithm, in the function *main*. It is also necessary to make ready this copy:

```
...
  int  RowNum;  // Number of process rows
  int *pSerialMatrix = 0;

  <…>

  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  if (ProcRank == 0) {
    // Matrix copying
    pSerialMatrix = new int[Size * Size];
    CopyMatrix(pMatrix, Size, pSerialMatrix);
  }
...
```

Besides, it is necessary to delete the allocated memory when it is not necessary any more:

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

  MPI_Finalize();

  return 0;
}
```

Then, add the function *TestResult* to the program code:

```
// Function for testing the result of parallel Floyd algorithm
void TestResult(int *pMatrix, int *pSerialMatrix, int Size) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialFloyd(pSerialMatrix, Size);
    if(!CompareMatrices (pMatrix, pSerialMatrix, Size)) {
      printf("The results of serial and parallel algorithms are "
        "NOT identical. Check your code\n");
    }
    else {
      printf("The results of serial and parallel algorithms are "
        "identical\n");
    }
  }
}
```

The result of the function execution is printing a diagnostic message. You can test the result of the parallel execution regardless of the number of graph vertices with the help of this function.

Transform into comments the call of the functions using the debugging print, which have been previously used for testing the correctness of the parallel application (the function *TestDistribution*, *ParallelPrintMatrix,*

PrintMatrix). Instead of the function *DummyDataInitialization*, which generates the initial data of the simple type, call the function *RandomDataInitialization*, which generates the initial data by means of the random data generator. Compile and run the application. Carry out experiment for different initial data amounts. Make sure that the program operates correctly.

Task 10 – Implement the Floyd Algorithm for Any Given Graph

The parallel application, which was developed in the course of executing the previous Tasks, was created for the case when the number of vertices *Size* in graph is divisible by the number of processors *ProcNum*. In this case the adjacency 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 case when the number of vertices *Size* is not multiple of the number of processes *ProcNum*. In this case the value *RowNum* of the number of rows processed on each process may be different: some processes will get $\lfloor Size/ProcNum \rfloor$, and the rest of them - $\lceil Size/ProcNum \rceil$ matrix rows of the adjacency matrix (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 graph vertices is not divisible by the number of processes. Now it is necessary to determine how many adjacency matrix rows each process should process. One of the simplest methods may consist in the following: all the processes except the last one (the process with the rank *ProcNum*-1) are allocated $\lfloor Size/ProcNum \rfloor$ matrix rows. The last process is allocated all the other rows ($Size - \lfloor Size/ProcNum \rfloor \cdot (ProcNum-1)$ rows. However, in this case the computational load may be distributed among the processes unequally. Thus, for instance, if the adjacency matrix size is equal to five, and the parallel application is run using three processes, the first two processes will be allocated a matrix row each, and the last process will get three rows.

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

Let us change the program code for calculating the value of the variable *RowNum*:

```
// Function for memory allocation and setting the initial values
void ProcessInitiliazation(int *&pMatrix, int *&pProcRows, int& Size,
  int& RowNum) {
...
  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  int RestRows = Size;
  for(int i = 0; i < ProcRank; i++)
    RestRows = RestRows - RestRows / (ProcNum - i);
  RowNum = RestRows / (ProcNum - ProcRank);

  // Allocate memory for the current process rows
  pProcRows = new int[Size * RowNum];
...
}
```

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

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

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

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

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

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

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

Very much in the same way we will use the more general function *MPI_Gatherv* for data gathering instead of the function *MPI_Gather* oriented at gathering the data of the same size from all the communicator processes. As in case of using the function *MPI_Scatterv*, the use of the function *MPI_Gatherv* requires two additional arrays:

```
// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  // Offset for storing the data from current process
  int RestRows = Size; // Number of rows, that haven’t been gathered yet

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

  // Determine the disposition of the result data block of current process
  RowNum = Size / ProcNum;
  pReceiveInd[0] = 0;
  pReceiveNum[0] = RowNum * Size;

  for(int i = 1; i < ProcNum; i++) {
    RestRows      -= RowNum;
    RowNum = RestRows / (ProcNum - i);
    pReceiveNum[i] = RowNum * Size;
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  // Gather the whole matrix on the process 0
  MPI_Gatherv(pProcRows, pReceiveNum[ProcRank], MPI_INT,
    pMatrix, pReceiveNum, pReceiveInd, MPI_INT, 0, MPI_COMM_WORLD);

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

The necessary changes must be also implemented for the function *RowDistribution*, as the stripe sizes may differ and the computation of the rank of the process, where the matrix row with the number *k* is located, becomes more complicated. Let us change the function *RowDistribution* with regard to this fact:

```
// Function for row broadcasting among all processes
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank;  // Process rank with the row k
  int ProcRowNum;   // Process row number

  // Finding the process rank with the row k
  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {
    if(k < Ind + Num ) break;
    RestRows -= Num;
    Ind      += Num;
    Num       = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
    // Copy the row to pRow array
    copy(&pProcRows[ProcRowNum*Size],&pProcRows[(ProcRowNum+1)*Size],pRow);

  // Broadcast row to all processes
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}
```

Compile and run the application. Check the correctness of the parallel program by means of the function *TestResult*.

Task 11 – Carry out the Computational Experiments

The main challenge in the implementation 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 program execution time. For this purpose we will add clocking to the program code. As the parallel algorithm includes the stages of data distribution, the Floyd algorithm execution and collecting the processed values, then clocking should start immediately before the call of the function *DataDistribution*, and stop right after the execution of the function *DataCollection*:

```
...
  double start, finish;
  double duration = 0.0;
...
  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

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

  // Testing the distribution
  //TestDistribution(pMatrix, pProcRows, Size, RowNum);

  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);
  //ParallelPrintMatrix(pProcRows, Size, RowNum);

  // Process data collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);
  finish = MPI_Wtime();

  //TestResult(pMatrix, pSerialMatrix, Size);

  duration = finish - start;
  if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);
...
```

It is obvious that this way we will print the time spent on the execution of calculations by the process with the rank 0. 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 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 5.3:

Table 5.3. The Results of the Computational Experiments for the Parallel Floyd Algorithm

Test Number	Number of Vertices	Execution Time			
		Serial Floyd Algorithm	Parallel Floyd Algorithm. Number of Processors		
			2	4	8
1	10				
2	500				
3	600				
4	700				
5	800				
6	900				
7	1,000				

The column "Serial Floyd Algorithm" is assigned for writing the execution times of the serial Floyd algorithm measured in the course of testing the serial program in Exercise 2. Calculate the obtained computation speed up as the ration of the serial algorithm time and the parallel algorithm time and give the results in Table 5.4.

Table 5.4. The Computation Speed Up Obtained for the Parallel Floyd Algorithm

Test Number	Speed Up		
	2 processors	4 processors	8 processors
1			
2			
3			
4			
5			
6			
7			

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

$$T_p = n^2 \cdot \lceil n/p \rceil \cdot \tau + n \cdot \lceil \log_2(p) \rceil (\alpha + w \cdot n/\beta) \tag{5.2}$$

(the derivation of this expression is considered in detail in subsection 11.1.5 of Section 11 "Parallel Algorithms of Graph Processing" of the training materials). Here *n* is the number of graph vertices, *p* is the number of processes, *τ* is the execution time of the basic computational operation of choosing the minimum values (this value was calculated in the course of testing the serial algorithm), *α* is the latency and *β* is the data communication network bandwidth. You should use the values obtained in the course of executing the Compute Cluster Server Lab 2 "Carrying out Jobs under Microsoft Compute Cluster Server 2003", as the values of the latency and the bandwidth.

Calculate the theoretical time of the parallel algorithm execution using formula (5.2). Tabulate the results in Table 5.5.

Table 5.5. The Comparison of the Experimental and the Theoretical Time of the Parallel Floyd Algorithm

Test Number	Number of Vertices	Parallel Algorithm Execution Time					
		2 processors		4 processors		8 processors	
		Model	Experiment	Model	Experiment	Model	Experiment
1	10						
2	500						
3	600						
4	700						
5	800						
6	900						
7	1,000						

Discussions

- How great is the difference between the execution time of the serial Floyd algorithm and the parallel algorithm? Why?
- Has there any speed up been obtained in the course of processing a graph containing 10 vertices? Why?
- Were the theoretical and the experiment execution time congruent? What might be the reason for incongruity?

Exercises

- Study other parallel algorithms of graph processing (the Prim algorithm for finding the minimum spanning tree, the Dejkstra method for solving the problem of finding the shortest path from one of the graph vertices to the other ones – see Section 11 "Parallel Algorithms of Graph Processing" of the training materials). Develop the programs, which implement these algorithms.

Appendix 1. The Program Code of the Serial Floyd Algorithm

File SerialFloyd.cpp

```
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <algorithm>

#include "SerialFloyd.h"
#include "SerialFloydTest.h"

using namespace std;

const double InfinitiesPercent = 50.0;
const double RandomDataMultiplier = 10;

int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

int main(int argc, char* argv[]) {
  int *pMatrix;    // Adjacency matrix
  int  Size;       // Size of adjacency matrix

  time_t start, finish;
  double duration = 0.0;

  printf("Serial Floyd algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, Size);

  printf("The matrix before Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  start = clock();
  // Parallel Floyd algorithm
  SerialFloyd(pMatrix, Size);
  finish = clock();

  printf("The matrix after Floyd algorithm\n");
  PrintMatrix(pMatrix, Size, Size);

  duration = (finish - start) / double(CLOCKS_PER_SEC);
  printf("Time of execution: %f\n", duration);

  // Process termination
  ProcessTermination(pMatrix);

  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int& Size) {
  do {
    printf("Enter the number of vertices: ");
    scanf("%d", &Size);

    if(Size <= 0)
      printf("The number of vertices should be greater then zero\n");
  } while(Size <= 0);

  printf("Using graph with %d vertices\n", Size);

  // Allocate memory for the adjacency matrix
  pMatrix = new int[Size * Size];

  // Data initalization
  DummyDataInitialization(pMatrix, Size);
  //RandomDataInitialization(pMatrix, Size);
}

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

// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {
  for(int i = 0; i < Size; i++)
    for(int j = i; j < Size; j++) {
      if(i == j) pMatrix[i * Size + j] = 0;
      else
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;
      pMatrix[j * Size + i] = pMatrix[i * Size + j];
    }
}

// Function for initializing the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1;
      }
      else
        pMatrix[i * Size + j] = 0;
}

// Function for the serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];
          pMatrix[i * Size + j] = Min(t1, t2);
      }
}
```

File SerialFloydTest.cpp

```
#include <cstdio>
#include "SerialFloydTest.h"

using namespace std;

// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
    for(int j = 0; j < ColCount; j++)
      printf("%7d", pMatrix[i * ColCount + j]);
    printf("\n");
  }
}
```

Appendix 2. The Program Code of the Parallel Floyd Algorithm

File ParallelFloyd.cpp

```
#include <cstdlib>
#include <cstdio>
#include <ctime>
#include <algorithm>
#include <mpi.h>

#include "ParallelFloyd.h"
#include "ParallelFloydTest.h"
```

```
using namespace std;

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

const double InfinitiesPercent = 50.0;
const double RandomDataMultiplier = 10;

int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

int main(int argc, char* argv[]) {
  int *pMatrix;      // Adjacency matrix
  int  Size;         // Size of adjacency matrix
  int *pProcRows;    // Process rows
  int  RowNum;       // Number of process rows

  double start, finish;
  double duration = 0.0;
  int *pSerialMatrix = 0;

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

  if(ProcRank == 0)
    printf("Parallel Floyd algorithm\n");

  // Process initialization
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  if (ProcRank == 0) {
    // Matrix copying
    pSerialMatrix = new int[Size * Size];
    CopyMatrix(pMatrix, Size, pSerialMatrix);
  }

  start = MPI_Wtime();
  // Distributing the initial data between processes
  DataDistribution(pMatrix, pProcRows, Size, RowNum);
  // Testing the distribution
  //TestDistribution(pMatrix, pProcRows, Size, RowNum);

  // Parallel Floyd algorithm
  ParallelFloyd(pProcRows, Size, RowNum);
  //ParallelPrintMatrix(pProcRows, Size, RowNum);

  // Process data collection
  ResultCollection(pMatrix, pProcRows, Size, RowNum);
  //if(ProcRank == 0)
  //  PrintMatrix(pMatrix, Size, Size);
  finish = MPI_Wtime();

  //TestResult(pMatrix, pSerialMatrix, Size);
  duration = finish - start;
  if(ProcRank == 0)
    printf("Time of execution: %f\n", duration);

  if (ProcRank == 0)
    delete []pSerialMatrix;

  // Process termination
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitialization(int *&pMatrix, int *&pProcRows, int& Size,
  int& RowNum) {
  setvbuf(stdout, 0, _IONBF, 0);

  if(ProcRank == 0) {
    do {
      printf("Enter the number of vertices: ");
      scanf("%d", &Size);

      if(Size < ProcNum)
        printf("The number of vertices should be greater then"
          "the number of processes\n");
    } while(Size < ProcNum);

    printf("Using the graph with %d vertices\n", Size);
  }

  // Broadcast the number of vertices
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Number of rows for each process
  int RestRows = Size;
  for(int i = 0; i < ProcRank; i++)
    RestRows = RestRows - RestRows / (ProcNum - i);
  RowNum = RestRows / (ProcNum - ProcRank);

  // Allocate memory for the current process rows
  pProcRows = new int[Size * RowNum];

  if(ProcRank == 0) {
    // Allocate memory for the adjacency matrix
    pMatrix = new int[Size * Size];

    // Data initalization
    DummyDataInitialization(pMatrix, Size);
    //RandomDataInitialization(pMatrix, Size);
  }
}

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

// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size) {
  for(int i = 0; i < Size; i++)
    for(int j = i; j < Size; j++) {
      if(i == j) pMatrix[i * Size + j] = 0;
      else
        if(i == 0) pMatrix[i * Size + j] = j;
        else       pMatrix[i * Size + j] = -1;
      pMatrix[j * Size + i] = pMatrix[i * Size + j];
    }
}

// Function for setting the data by the random generator
void RandomDataInitialization(int *pMatrix, int Size) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < Size; i++)
    for(int j = 0; j < Size; j++)
      if(i != j) {
        if((rand() % 100) < InfinitiesPercent)
          pMatrix[i * Size + j] = -1;
        else
          pMatrix[i * Size + j] = rand() + 1;
      }
      else
        pMatrix[i * Size + j] = 0;
}

// Function for the data distribution among the processes
void DataDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  int *pSendNum; // Number of elements sent to the process
  int *pSendInd; // Index of the first data element sent to the process
  int RestRows = Size; // Number of rows, that havenTt been distributed yet

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

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

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

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

// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  int *pReceiveNum;  // Number of elements, that current process sends
  int *pReceiveInd;  // Offset for storing the data from current process
  int RestRows = Size; // Number of rows, that haven't been gathered yet

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

  // Determine the disposition of the result data block of current process
  RowNum = Size / ProcNum;
  pReceiveInd[0] = 0;
  pReceiveNum[0] = RowNum * Size;

  for(int i = 1; i < ProcNum; i++) {
    RestRows      -= RowNum;
    RowNum = RestRows / (ProcNum - i);
    pReceiveNum[i] = RowNum * Size;
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  // Gather the whole matrix on the process 0
  MPI_Gatherv(pProcRows, pReceiveNum[ProcRank], MPI_INT,
    pMatrix, pReceiveNum, pReceiveInd, MPI_INT, 0, MPI_COMM_WORLD);

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

// Function for the parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;
  for(int k = 0; k < Size; k++) {
    // Distribute row among all processes
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Update adjacency matrix elements
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) &&
            (pRow     [j]            != -1)) {
          t1 = pProcRows[i * Size + j];
          t2 = pProcRows[i * Size + k] + pRow[j];

        pProcRows[i * Size + j] = Min(t1, t2);
      }
  }

  delete []pRow;
}

// Function for row broadcasting among all processes
void  RowDistribution(int  *pProcRows,  int  Size,  int  RowNum,  int  k,  int
*pRow) {
  int ProcRowRank;  // Process rank with the row k
  int ProcRowNum;   // Process row number

  // Finding the process rank with the row k
  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {
    if(k < Ind + Num ) break;
    RestRows -= Num;
    Ind      += Num;
    Num       = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
    // Copy the row to pRow array
    copy(&pProcRows[ProcRowNum*Size],&pProcRows[(ProcRowNum+1)*Size],pRow);

  // Broadcast row to all processes
  MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
}

// Function for formatted output of all stripes
void ParallelPrintMatrix(int *pProcRows, int Size, int RowNum) {
  for(int i = 0; i < ProcNum; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      fflush(stdout);
      printf("Proc rows:\n");
      fflush(stdout);
      PrintMatrix(pProcRows, RowNum, Size);
      fflush(stdout);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for testing the data distribution
void TestDistribution(int *pMatrix, int *pProcRows, int Size, int RowNum) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial adjacency matrix:\n");
    PrintMatrix(pMatrix, Size, Size);
  }

  MPI_Barrier(MPI_COMM_WORLD);

  ParallelPrintMatrix(pProcRows, Size, RowNum);
}

// Testing the result of parallel Floyd algorithm
void TestResult(int *pMatrix, int *pSerialMatrix, int Size) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialFloyd(pSerialMatrix, Size);
    if(!CompareMatrices(pMatrix, pSerialMatrix, Size)) {
      printf("Results of serial and parallel algorithms are "
        "NOT identical. Check your code\n");
    }
    else {
      printf("Results of serial and parallel algorithms are "
        "identical\n");
    }
  }
}
```

File ParallelFloydTest.cpp

```
#include <cstdio>
#include <algorithm>
using namespace std;
```

```
#include "ParallelFloyd.h"
#include "ParallelFloydTest.h"

// Function for copying the matrix
void CopyMatrix(int *pMatrix, int Size, int *pMatrixCopy) {
  copy(pMatrix, pMatrix + Size * Size, pMatrixCopy);
}

// Function for comparing the matrices
bool CompareMatrices(int *pMatrix1, int *pMatrix2, int Size) {
  return equal(pMatrix1, pMatrix1 + Size * Size, pMatrix2);
}

// Function for the serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size) {
  int t1, t2;
  for(int k = 0; k < Size; k++)
    for(int i = 0; i < Size; i++)
      for(int j = 0; j < Size; j++)
        if((pMatrix[i * Size + k] != -1) &&
           (pMatrix[k * Size + j] != -1)) {
          t1 = pMatrix[i * Size + j];
          t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];
          pMatrix[i * Size + j] = Min(t1, t2);
      }
}

// Function for formatted matrix output
void PrintMatrix(int *pMatrix, int RowCount, int ColCount) {
  for(int i = 0; i < RowCount; i++) {
    for(int j = 0; j < ColCount; j++) {
      printf("%7d", pMatrix[i * ColCount + j]);
      fflush(stdout);
    }
    printf("\n");
    fflush(stdout);
  }
}
```