


Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

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

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

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

Parallel Programming for Multiprocessor Distributed Memory Systems

08 Practice
Parallel Algorithms of Graph Processing

With the support of Microsoft

Sysoyev A.V.
Software department

Contents

- The Shortest Path Problem Statement
- Serial Floyd Algorithm Implementation
- Parallel Floyd Algorithm
- Parallel Floyd Program

THE SHORTEST PATH PROBLEM STATEMENT

Step 1. Problem Statement…

- Let *G* be a graph

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

V – the set of vertices

$$v_i, 1 \le i \le n$$

R – the set of arcs

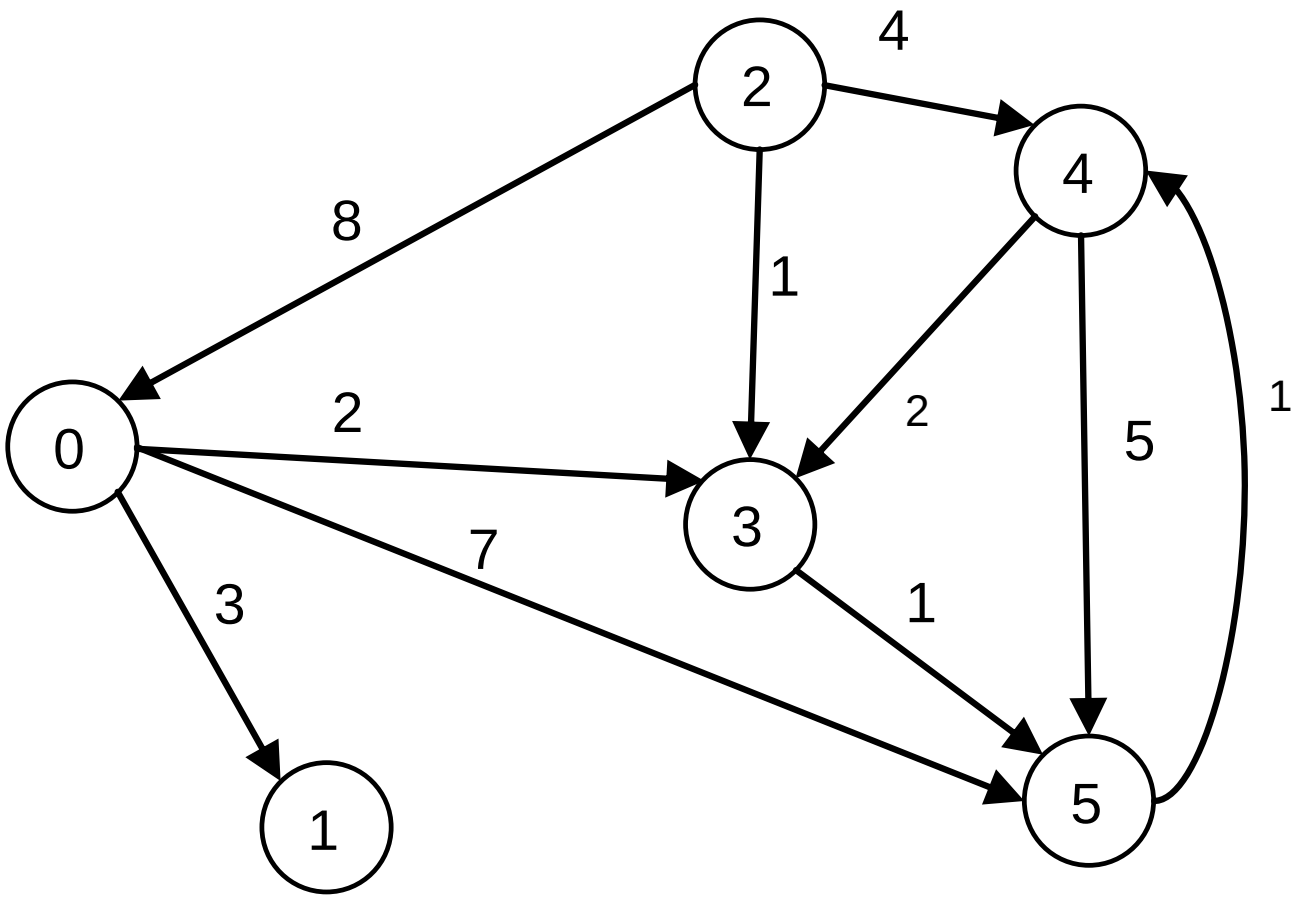
$$r_j = (v_{s_j}, v_{t_j}), 1 \le j \le m$$

- Generally, the arcs may be assigned certain numerical characteristics (weights)

$$w_j, 1 \le j \le m$$

Step 1. Problem Statement…

- Example of the Weighted Oriented Graph



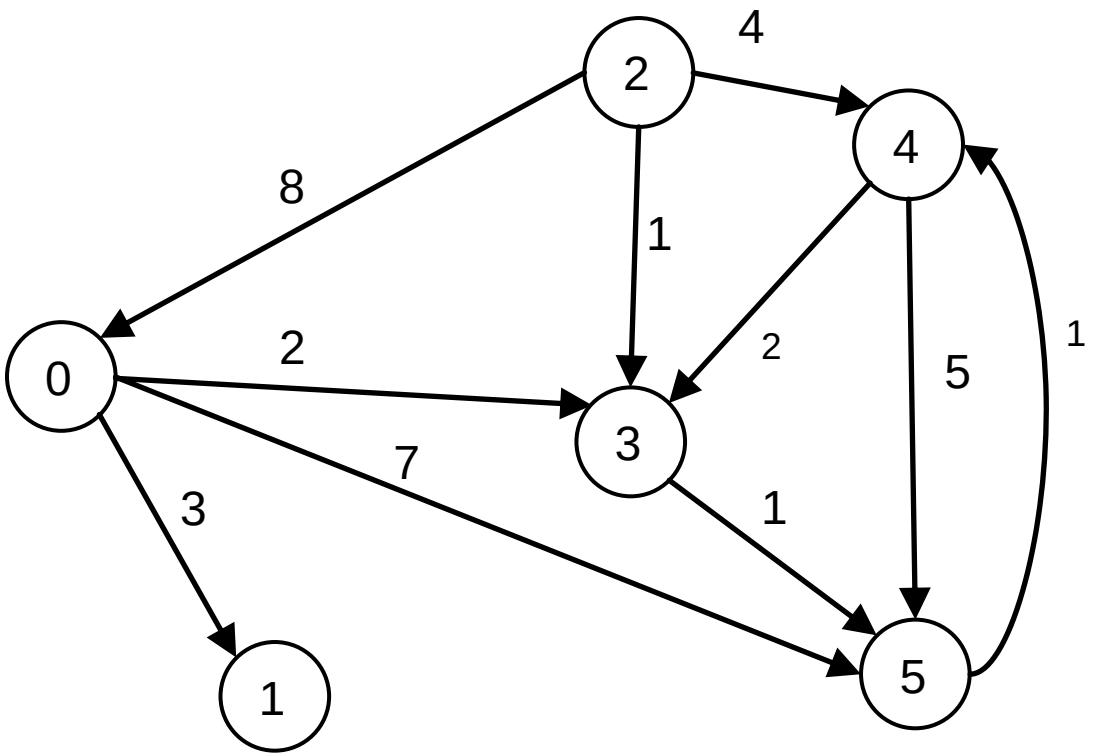
Step 1. Problem Statement…

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

$$A = (a_{ij}),\ 1 \le i, j \le n, \qquad a_{ij} = \begin{cases} w(v_i, v_j), & \text{если } (v_i, v_j) \in R, \\ 0, & \text{если } i = j, \\ \infty, & \text{иначе}. \end{cases}$$

Step 1. Problem Statement…

- The Adjacency Matrix



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

Step 1. Problem Statement…

The problem of searching the shortest paths

- The initial information for the problem is the weighted graph
- Each arc of the graph is assigned non-negative weight
- The graph is assumed to be oriented
- The problem is to find out the shortest paths among all the pairs of destination vertices for the given graph *G*
- As a method for solving the problem, we will further use Floyd algorithm

Step 1. Problem Statement

- 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 complexity of the Floyd algorithm is of n^3 order

SERIAL FLOYD ALGORITHM IMPLEMENTATION

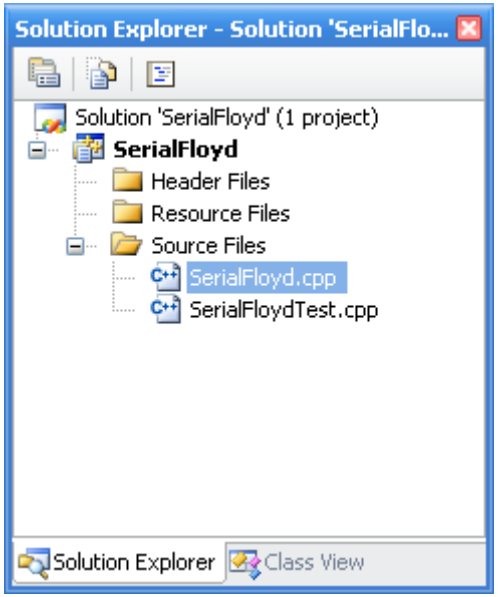
Step 2. Serial Implementation…

- Task 1 – Open the project SerialFloyd
- Task 2 – Input the Number of Vertices
- Task 3 – Terminate the Program Execution
- Task 4 – Implement the Floyd Algorithm
- Task 5 – Carry out the Computational Experiments

Step 2. Serial Implementation…

Task 1 – Open the project SerialFloyd

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



Step 2. Serial Implementation…

Task 2 – Input the Number of Vertices

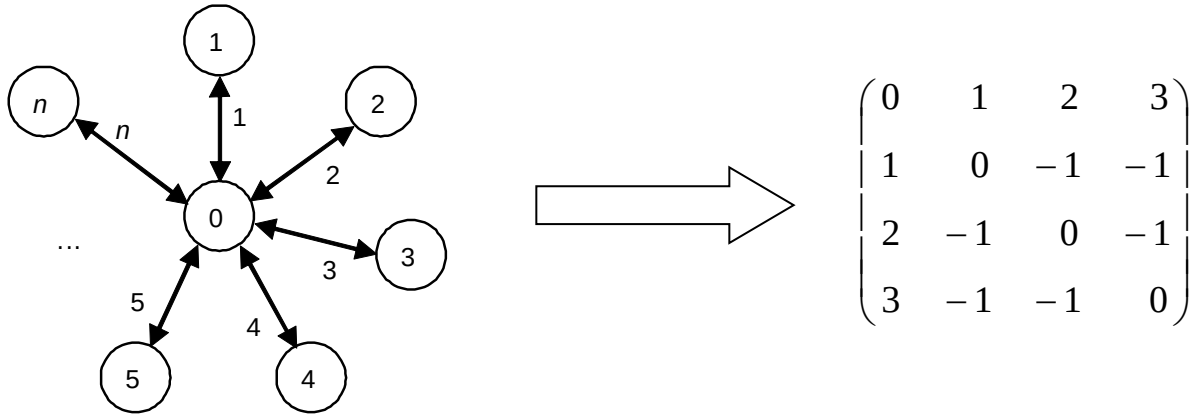
- To set the initial data of the serial Floyd algorithm develop the function **`ProcessInitialization()`**
- This function is intended for
  - the initialization of all the variables used in the program
  - the input of the number of the processed vertices (the adjacency matrix size)
  - the allocation of the memory for the adjacency matrix
  - filling the memory with the initial values

```
// Function for process initialization
void ProcessInitialiazation(int *&pMatrix, int& Size);
```

Step 2. Serial Implementation…

Task 2 – Input the Number of Vertices

- Implement the functions **`DummyDataInitialization()`**
- Use the simple method of implementation, containing the set of data, the correctness of which can be easily checked



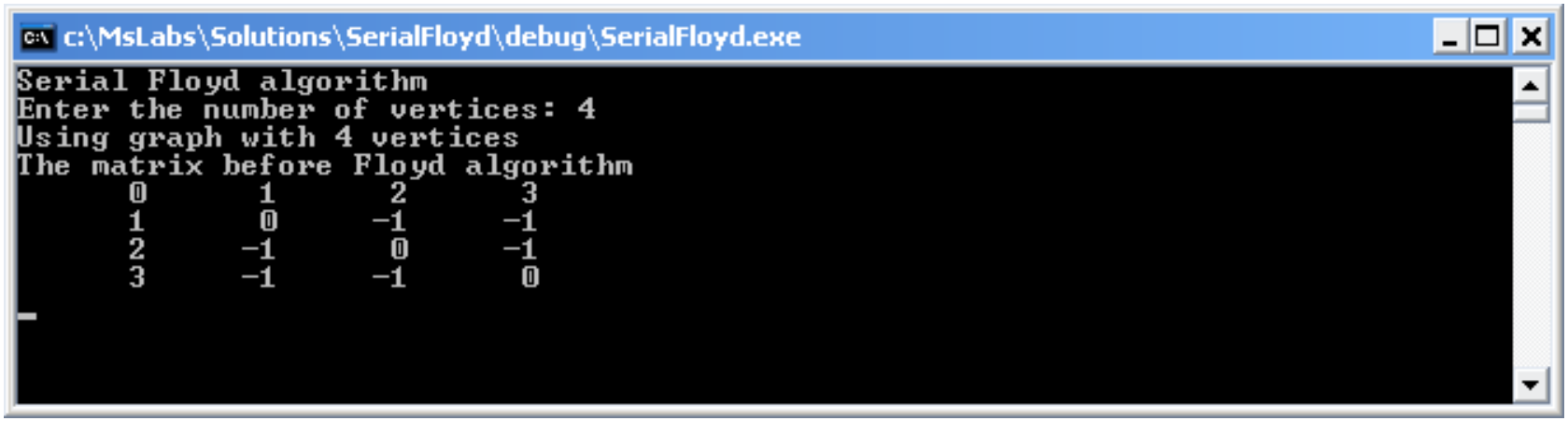
$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & -1 & -1 \\ 2 & -1 & 0 & -1 \\ 3 & -1 & -1 & 0 \end{pmatrix}$$

```
// Function for simple setting the initial data
void DummyDataInitialization(int *pMatrix, int Size);
```

Step 2. Serial Implementation…

Task 2 – Input the Number of Vertices

- Develop the function **`ProcessInitialization()`**, **`DummyDataInitialization()`** and **`PrintMatrix()`**
- Call the functions **`ProcessInitialization()`** and **`PrintMatrix()`** from main function of the application
- Compile and run the application



Step 2. Serial Implementation…

Task 3 –Terminate the Program Execution

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

```
// Function for computational process termination
void ProcessTermination(int *pMatrix);
```

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

Step 2. Serial Implementation…

Task 4 – Implement the Floyd Algorithm

- To execute the Floyd algorithm develop the function **`SerialFloyd()`**

```
// Serial Floyd algorithm
void SerialFloyd(int *pMatrix, int Size);
```

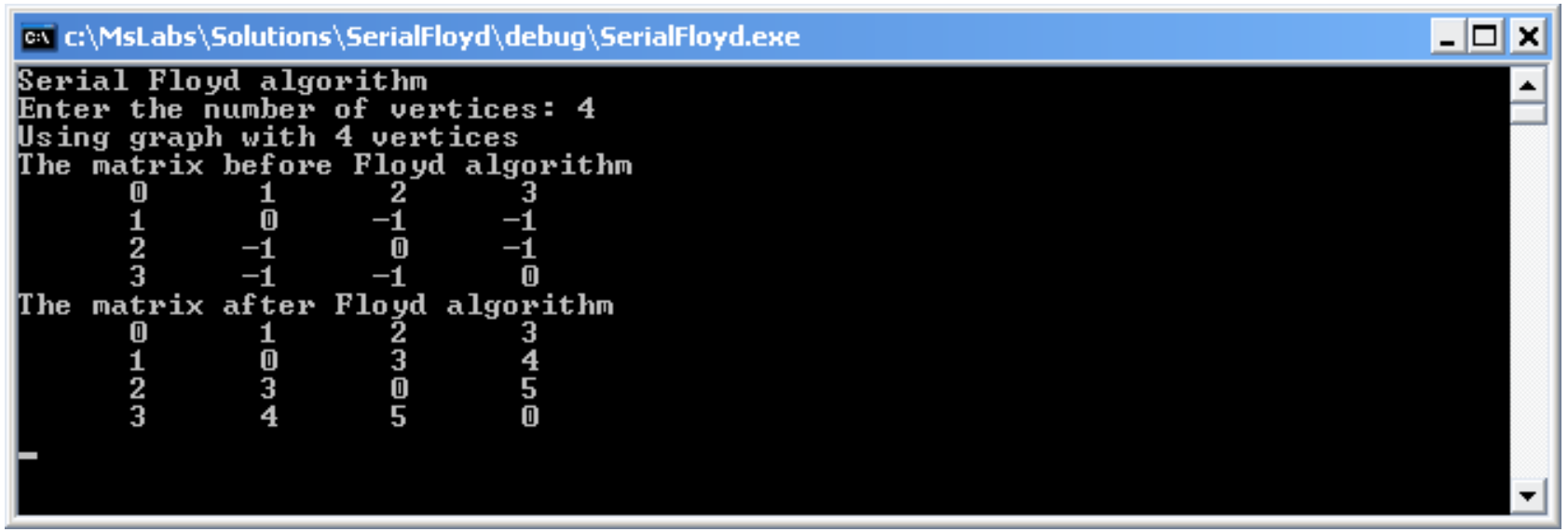
- The value -1 has been chosen to denote the infinitely. Implement the function **`Min()`**

```
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;
}
```

Step 2. Serial Implementation…

Task 4 – Implement the Floyd Algorithm

- Implement the function **`SerialFloyd()`**
- Add debugging print to the function **`main()`**
- Compile and run the application



Step 2. Serial Implementation

Task 5 – Carry out the Computational Experiments

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

```
// Function for random generating the initial data
void RandomDataInitialization(int *pMatrix, int Size);
```

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

PARALLEL FLOYD ALGORITHM

Step 3. Parallel Algorithm…

Subtask definition

- The effective way of parallel scheme of the Floyd algorithm is to update the values of matrix A simultaneously

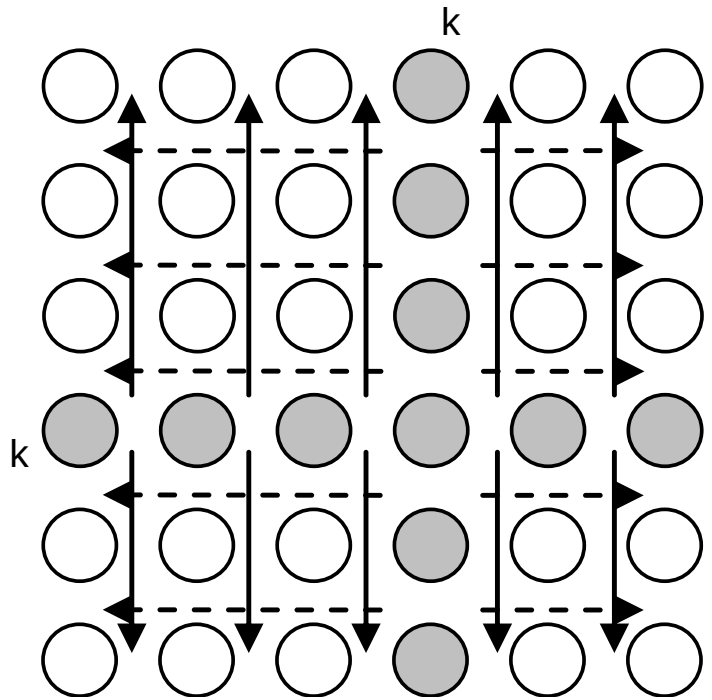
Analysis of Information Dependencies

- 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
- 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), $1 \le i \le n$

Step 3. Parallel Algorithm…

Analysis of Information Dependencies

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



Step 3. Parallel Algorithm…

Scaling and Distributing the Subtask among the Processors

- A possible way to aggregate the computations is to use of block-striped scheme of the matrix *A* partitioning
- 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*
- We will further analyze only partitioning the matrix *A* into **horizontal stripes**

PARALLEL FLOYD PROGRAM

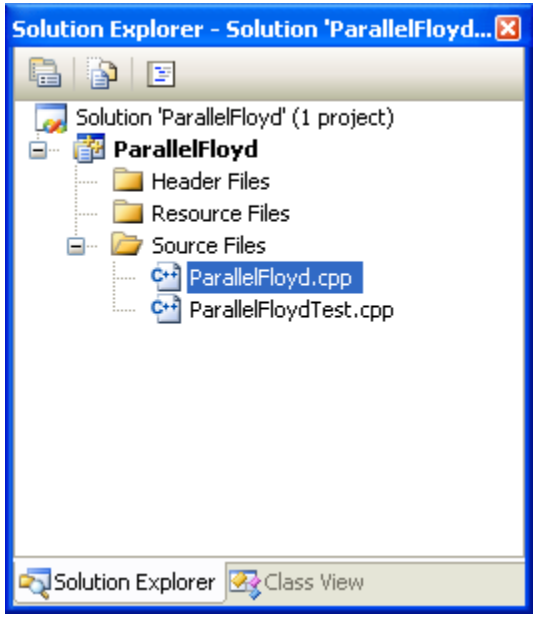
Step 4. Parallel Program…

- Task 1 – Open the Project ParallelFloyd
- Task 2 – Initialize and Finalize the Parallel Program
- Task 3 – Input the Initial Data
- Task 4 – Terminate the Calculations
- Task 5 – Distribute the Data among the Processes
- Task 6 – Implement the Parallel Floyd Algorithm
- Task 7 – Implement the Floyd Algorithm Iterations
- Task 8 – Collect the Result Matrix
- Task 9 – Test the Parallel Program Correctness
- Task 10 – Carry out the Computational Experiments

Step 4. Parallel Program…

Task 1 – Open the Project ParallelFloyd

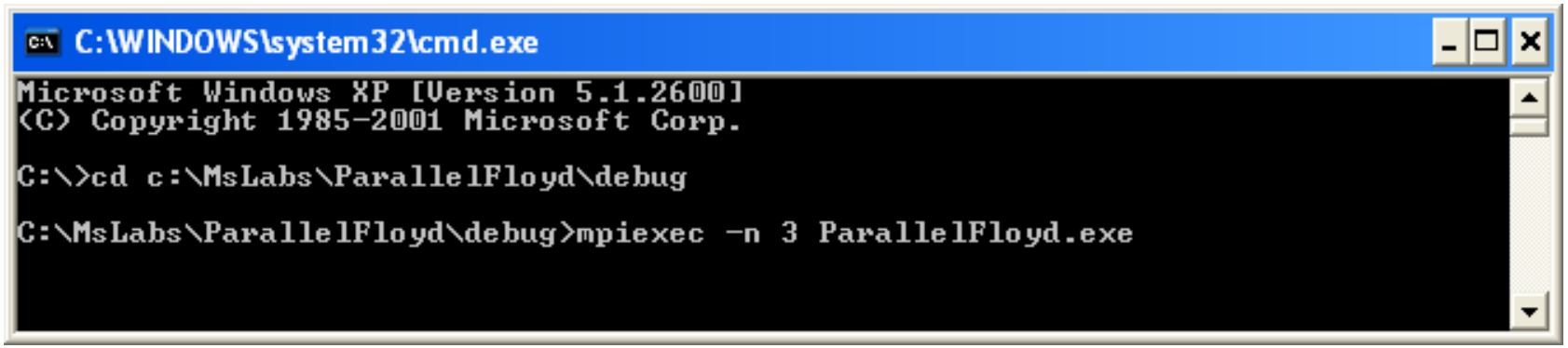
- Start **Microsoft Visual Studio**
- Open solution ParallelFloyd.sln from the folder **c:\ParLabs\ParallelFloyd**
- Open file **ParallelFloyd.cpp** in the window Solution Explorer (Ctrl+Alt+L)



Step 4. Parallel Program…

Task 2 – Initialize and Finalize the Parallel Program

- Initialize the environment of the MPI program execution in the main function
- Determine the number of processes available for MPI program
- Determine the process rank in communicator **`MPI_COMM_WORLD`**
- Set global variables for storing these values (**`ProcNum`** and **`ProcRank`** correspondingly)
- Compile and run the parallel application



Step 4. Parallel Program…

Task 3 – Input the Initial Data

- To initialize the computations develop the function ProcessInitialization
  - input the amount of the number of vertices in the graph
  - broadcast the number of vertices to the other processes
  - allocate memory for the adjacency matrix and the stripes assigned to the processes

```
// Function for process initialization
void ProcessInitialiazation(int *&pMatrix,int *&pProcRows,
  int& Size, int& RowNum);
```

- Implement the function **`ProcessInitialization()`**
- Call the function from the main function of application
- Compile and run the application

Step 4. Parallel Program…

Task 4 –Terminate the Calculations

- Develop the function for correct program termination **`ProcessTermination()`**
- Deallocate the memory for storing the adjacency matrix **`pMatrix`** (on the root process), and the memory for storing the matrix stripes **`pProcRows`**

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

- Call the function from the main function of application
- Compile and run the application

Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

- In accordance with the parallel computation scheme the adjacency matrix must be distributed among the processes in equal stripes
- To distribute the matrix **`pMatrix`** use the function **`MPI_Scatterv()`**
- Implement the function **`DataDistribution()`**

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

Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

- Call the function **`DataDistribution()`** from the main program
- To test the correctness of the data distribution among the processes implement the “debugging print” function **`TestDistribution()`**
  - Print the adjacency matrix **`pMatrix`** on the root process
  - Print the matrix stripes, which are distributed on each of the processes

```
// Data distribution among the processes
void TestDistribution(int *pMatrix, int *pProcRows,
  int Size, int RowNum);
```

Step 4. Parallel Program…

Task 5 – Distribute the Data among the Processes

- Make sure that the data is distributed correctly

```
C:\WINDOWS\system32\cmd.exe
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>_
```

Step 4. Parallel Program…

Task 6 – Implement the Parallel Floyd Algorithm

- 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
  - All the processes need the matrix row, the number of which coincides with the iteration number
  - It is necessary to broadcast this row among the processes
  - Implement the function **`RowDistribution()`**
  - It 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

```
// Function for row broadcasting
void RowDistribution(int *pProcRows, int Size, int RowNum,
  int k, int *pRow);
```

Step 4. Parallel Program…

Task 6 – Implement the Parallel Floyd Algorithm

- The code for the parallel Floyd algorithm will look the following way at the first stage

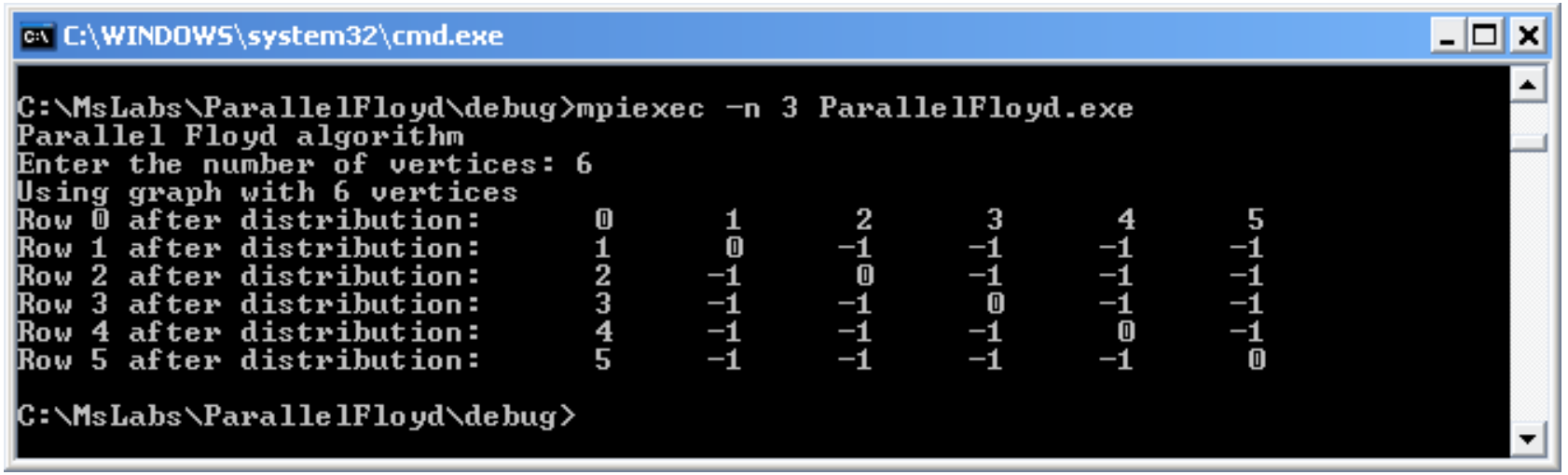
```
// Function for the parallel Floyd algorithm
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];

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

Step 4. Parallel Program…

Task 6 – Implement the Parallel Floyd Algorithm

- Implement the function **`RowDistribution()`**
- Call the function **`ParallelFloyd()`** from the main function of the application
- Compile and run the application
- Make sure that the data rows are distributed correctly



Step 4. Parallel Program…

Task 7 – Implement the Floyd Algorithm Iterations

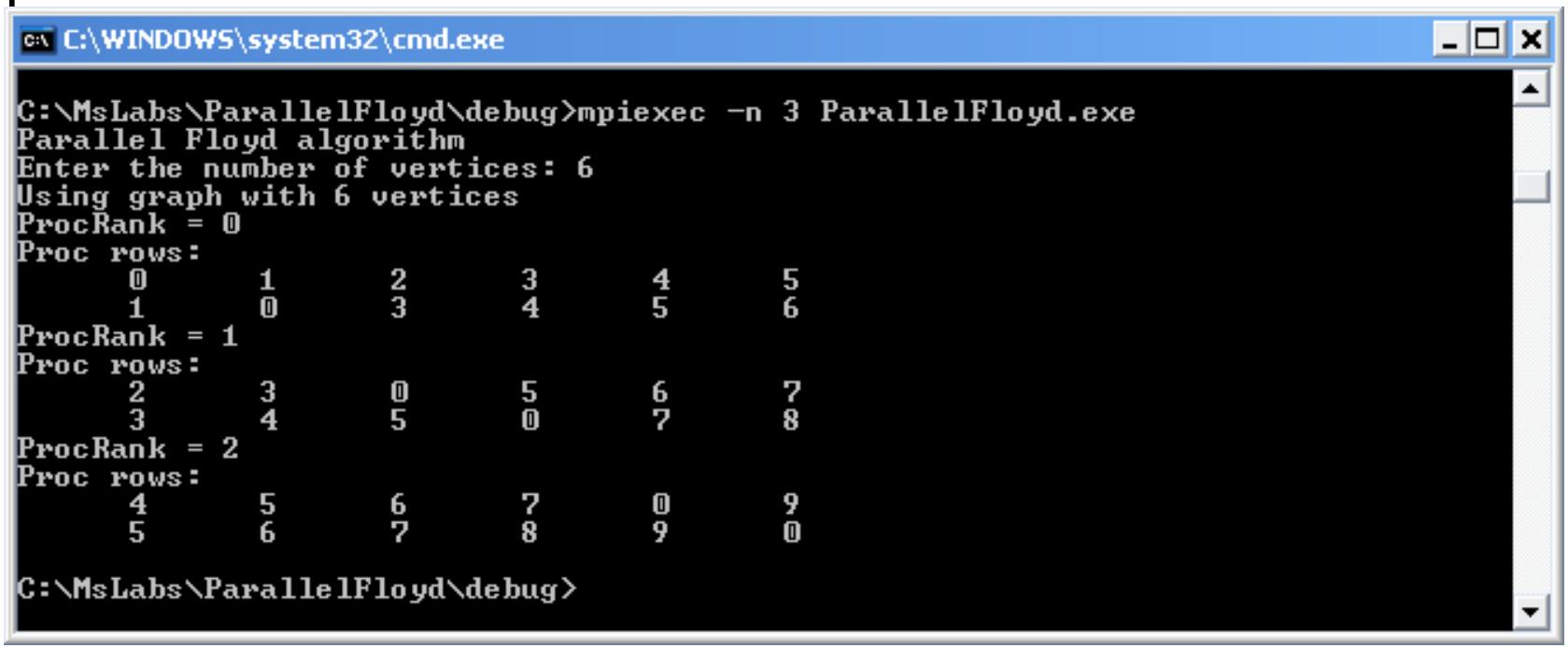
- It is necessary to execute the adjacency matrix update after broadcasting the next adjacency matrix row among the processes
- Add to the function **`ParallelFloyd()`** the following code on each iteration

```
// 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);
    }
```

Step 4. Parallel Program…

Task 7 – Implement the Floyd Algorithm Iterations

- 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



Step 4. Parallel Program…

Task 8 – Collect the Result Matrix

- To collect the obtained matrix on the root process implement the function **`ResultCollection()`**

```
// Function for process result collection
void ResultCollection(int *pMatrix, int *pProcRows,
  int Size, int RowNum);
```

- Call the function from main function of the application
- Add print of the obtained matrix by means of the function **`PrintMatrix()`** on the process with rank 0
- Compile and run the application
- Check the correctness of the program execution

Step 4. Parallel Program…

Task 9 – Test the Parallel Program Correctness

- To test the correctness of the program develop the function **`TestResult()`**

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

- The function should compare the results of the serial program to the results of the parallel one
- To execute the serial algorithm use the function **`SeriaFloyd()`**
- To make the serial algorithm **`SerialFloyd()`** operate the same data as the developed parallel algorithm **`ParallelFloyd()`**, produce a copy of the data

Step 4. Parallel Program…

Task 9 – Test the Parallel Program Correctness

- Implement the function **`TestResult()`**
- Call the function from main function of the application
- Instead of the function **`DummyDataInitialization()`**, call the function **`RandomDataInitialization()`**
- Compile and run the application
- Set various amounts of the initial data
- Make sure that the application is functioning properly

Step 4. Parallel Program

Task 10 – Carry out the Computational Experiments

- Determine the parallel algorithm execution time
- Carry out the computational experiments with large objects
- Determine the given speedup
- Fill the table with results of experiments

Summary

- Method of solving the shortest path problem (Floyd algorithm) is considered
- Serial and parallel versions of Floyd algorithm are implemented
- Computational experiments are performed, comparison of serial and parallel algorithms is made

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
- Develop the programs, which implement these algorithms

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein C. (2009). Introduction to Algorithms, 3rd Edition. – The MIT Press.
2. Schloegel, K., Karypis, G., Kumar, V. (2000). Graph Partitioning for High Performance Scientific Simulations.
3. Quinn, M.J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
4. Kumar V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc. (2nd edn., 2003)
5. Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.
6. Foster, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.