


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

07 Practice
Parallel Algorithms of Solving the Linear Equation Systems

With the support of Microsoft

Sysoyev A.V.
Software department

Contents

- The Problem Statement of Solving the Linear Equation Systems
- Gauss Algorithm Studying
- Serial Implementation
- Parallel Algorithm
- Parallel Program

THE PROBLEM STATEMENT OF SOLVING THE LINEAR EQUATION SYSTEMS

Step 1. Problem Statement…

- Linear equation with *n* independent unknowns

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

- Set of n linear equations is termed a system of linear equations or a linear system

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

- In matrix form

$$Ax = b$$

Step 1. Problem Statement

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

GAUSS ALGORITHM STUDYING

Step 2. Gauss Algorithm Studying…

- The main concept of the method is a modification of matrix A by means of equivalent transformations to a triangle form
- After that the values of the desired unknown variables may be obtained directly in an explicit form
- Equivalent transformations:
  - the multiplication of any equation by a nonzero constant
  - the permutation of equations
  - the addition of any system equation to other equation

Step 2. Gauss Algorithm Studying…

- At the first stage (the **Gaussian elimination** stage) the initial system of linear equations is reduced to the upper triangle form with the use of sequential elimination of unknowns

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

- At the **back substitution** (the second stage of the algorithm) the values of unknown variables are determined

SERIAL IMPLEMENTATION

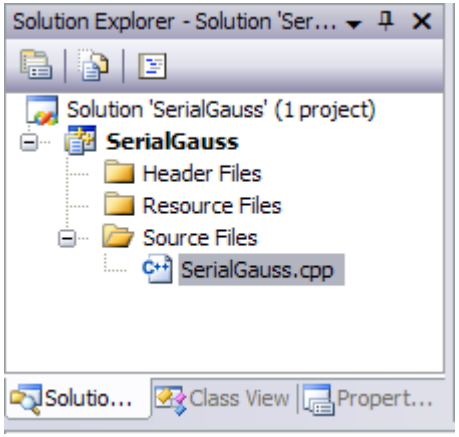
Step 3. Serial Implementation…

- Task 1 – Open the Project SerialGauss
- Task 2 – Input the Matrix and Vector Sizes
- Task 3 – Input the Initial Data
- Task 4 – Terminate the Program Execution
- Task 5 – Implement the Gaussian Elimination
- Task 6 – Implement the Back Substitution
- Task 7 – Carry out Computational Experiments

Step 3. Serial Implementation…

Task 1 – Open the Project SerialGauss

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



Step 3. Serial Implementation…

Task 1 – Open the Project SerialGauss

- Next variables will be used in the program

```
double* pMatrix; // The matrix of linear system
double* pVector; // The right parts of the linear system
double* pResult; // The result vector
int Size;     // Sizes of the initial matrix and the vector
```

- The program code, which follows the declarations of the variables, is the output of the initial message and the waiting for pressing any key before the application exit

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

Step 3. Serial Implementation…

Task 2 – Input the Matrix and Vector Sizes

- In order to input the initial data of the Gauss algorithm implement the function **`ProcessInitialization()`**
  - determine the sizes of the objects
  - allocate the memory for the objects involved in multiplication (**`pMatrix`**, **`pVector`** and **`pResult`**)
  - sets the values of the initial matrix and vector elements

```
// Function for process initialization
void ProcessInitialization(double* &pMatrix,
  double* &pVector, double* &pResult, int &Size);
```

Step 3. Serial Implementation…

Task 2 – Input the Matrix and Vector Sizes

- Determine the sizes of the objects with correct input control

```
// Function for process initialization
void ProcessInitialization(double* &pMatrix,
  double* &pVector, double* &pResult, int &Size) {
  // Setting the size of the initial matrix and the vector
  do {
    printf("\nEnter size of the initial objects: ");
    scanf("%d", &Size);
    printf("\nChosen objects' size = %d", Size);
    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);
}
```

Step 3. Serial Implementation…

Task 2 – Input the Matrix and Vector Sizes

- Add the call of the function **`ProcessInitialization()`** to the **`main()`** function after the initial message line

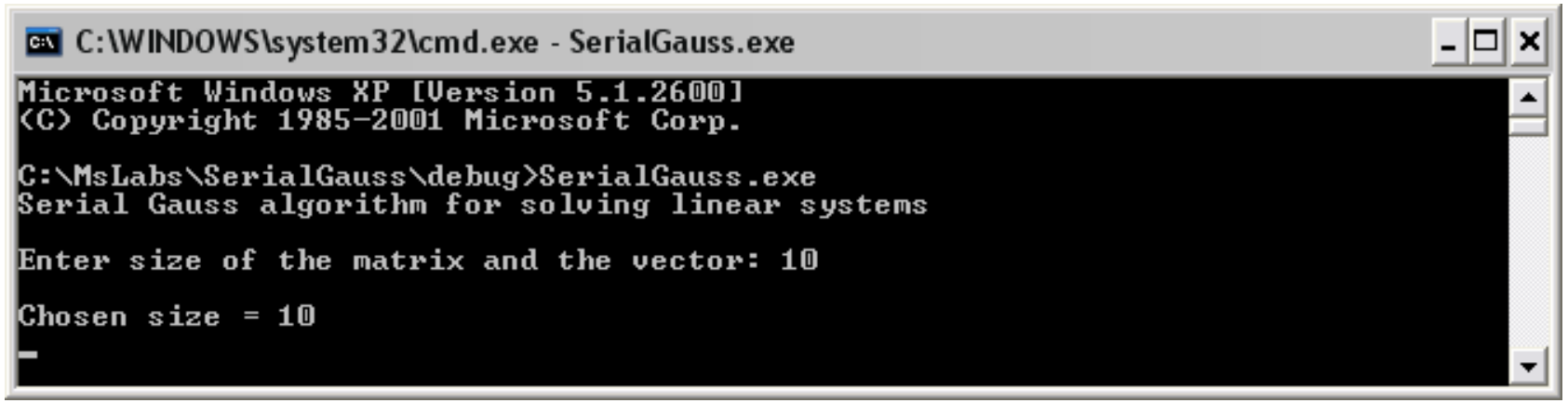
```
void main() {
  double* pMatrix;  // Initial matrix
  double* pVector;  // Initial vector
  double* pResult;  // Result vector
  int Size;         // Sizes of initial matrix and vector

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

Step 3. Serial Implementation…

Task 2 – Input the Matrix and Vector Sizes

- Compile and run the application



Step 3. Serial Implementation…

Task 3 – Input the Initial Data

- Memory allocation

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

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

Step 3. Serial Implementation…

Task 3 – Input the Initial Data

- Implement the function **`DummyDataInitialization()`** to set the matrices elements by the following template

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

```
// Function for simple initialization of the matrix
// and the vector elements
void DummyDataInitialization (double* pMatrix,
  double* pVector, int Size);
```

Step 3. Serial Implementation…

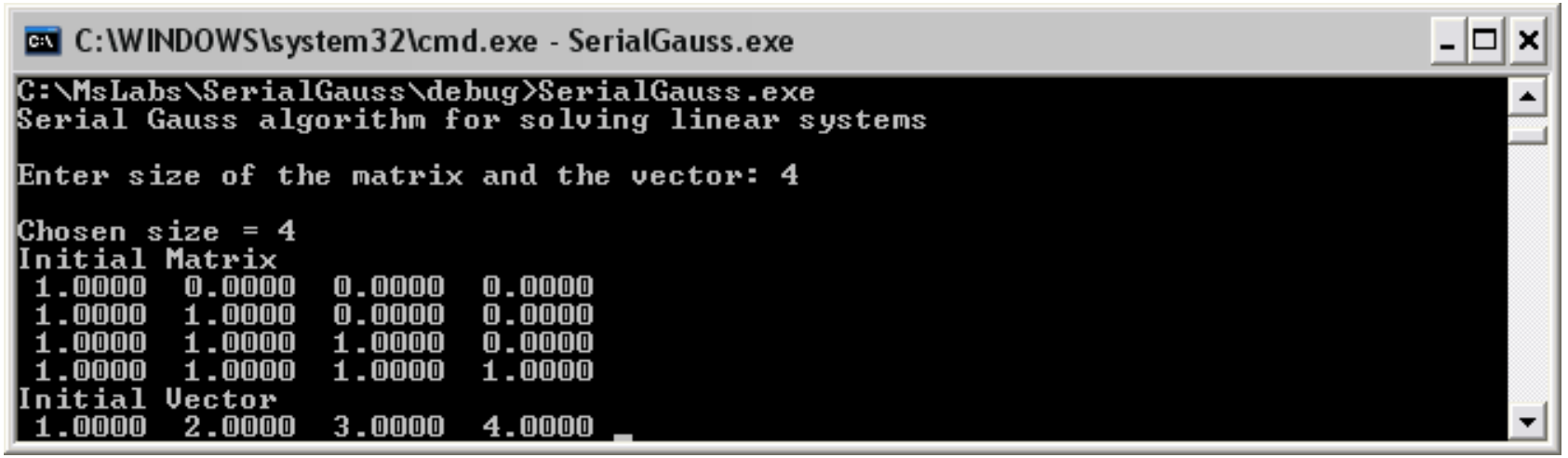
Task 3 – Input the Initial Data

- Call the function **`DummyDataInitialization()`** after allocating memory inside the function **`ProcessInitialization()`**
- Print out the matrix **`pMatrix`** and the vector **`pVector`** in the main function after calling the function **`ProcessInitialization()`**
- Use of the formatted matrix output function **`PrintMatrix()`** and formatted vector output function **`PrintVector()`**, which was developed in the Practice 05

Step 3. Serial Implementation…

Task 3 – Input the Initial Data

- Compile and run the application
- Check the correctness of data input



Step 3. Serial Implementation…

Task 4 – Terminate the Program Execution

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

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

Step 3. Serial Implementation…

Task 4 – Terminate the Program Execution

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

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

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

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

Step 3. Serial Implementation…

Task 5 – Implement the Gaussian Elimination

- To solve the linear equation system by means of the Gauss algorithm develop the function **`SerialResultCalculation()`**

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

Step 3. Serial Implementation…

Task 5 – Implement the Gaussian Elimination

- Declare two array variables
  - **`pSerialPivotPos`** – to store the order of choosing the pivot rows
  - **`pSerialPivotIter`** – to store the number of the iteration where the row with the number *i* was chosen as the pivot one

```
int* pSerialPivotPos;
int* pSerialPivotIter;
```

- Add memory allocation for declared array and initialize their elements

Step 3. Serial Implementation…

Task 5 – Implement the Gaussian Elimination

- It is necessary to determine the pivot matrix row at each iteration
- Implement the function **`FindPivotRow()`**

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

Step 3. Serial Implementation…

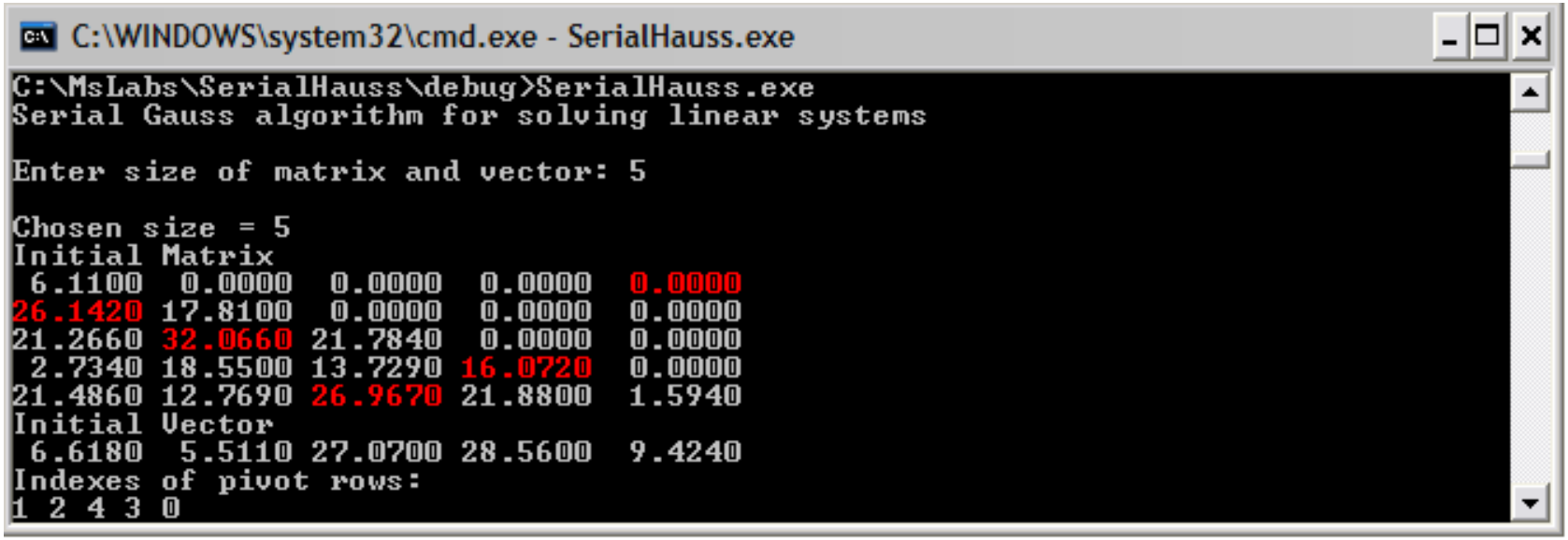
Task 5 – Implement the Gaussian Elimination

- Add the call of the function **`FindPivotRow()`** to the function, which carries out the Gaussian elimination – **`SerialGaussianElimination()`**
- Store the obtained value in corresponding element of the array **`pPivotPos`**
- Print the numbers of the selected pivot rows to check the computation correctness
- Comment the call of the function **`SerialBackSubstitution()`**

Step 3. Serial Implementation…

Task 5 – Implement the Gaussian Elimination

- Add the call of the function **`SerialResultCalculation()`** to the main function
- Compile the application and run. Make sure that the pivot rows are chosen correctly (marked by the red color in example)



Step 3. Serial Implementation…

Task 5 – Implement the Gaussian Elimination

- After selecting the pivot rows, these rows multiplied by the corresponding multipliers are subtracted from the rows, which have not yet been chosen as the pivot ones
- To carry out the subtraction develop the function **`SerialColumnElimination()`**

```
// Column elimination
void SerialColumnElimination(double* pMatrix,
  double* pVector, int Pivot, int Iter, int Size);
```

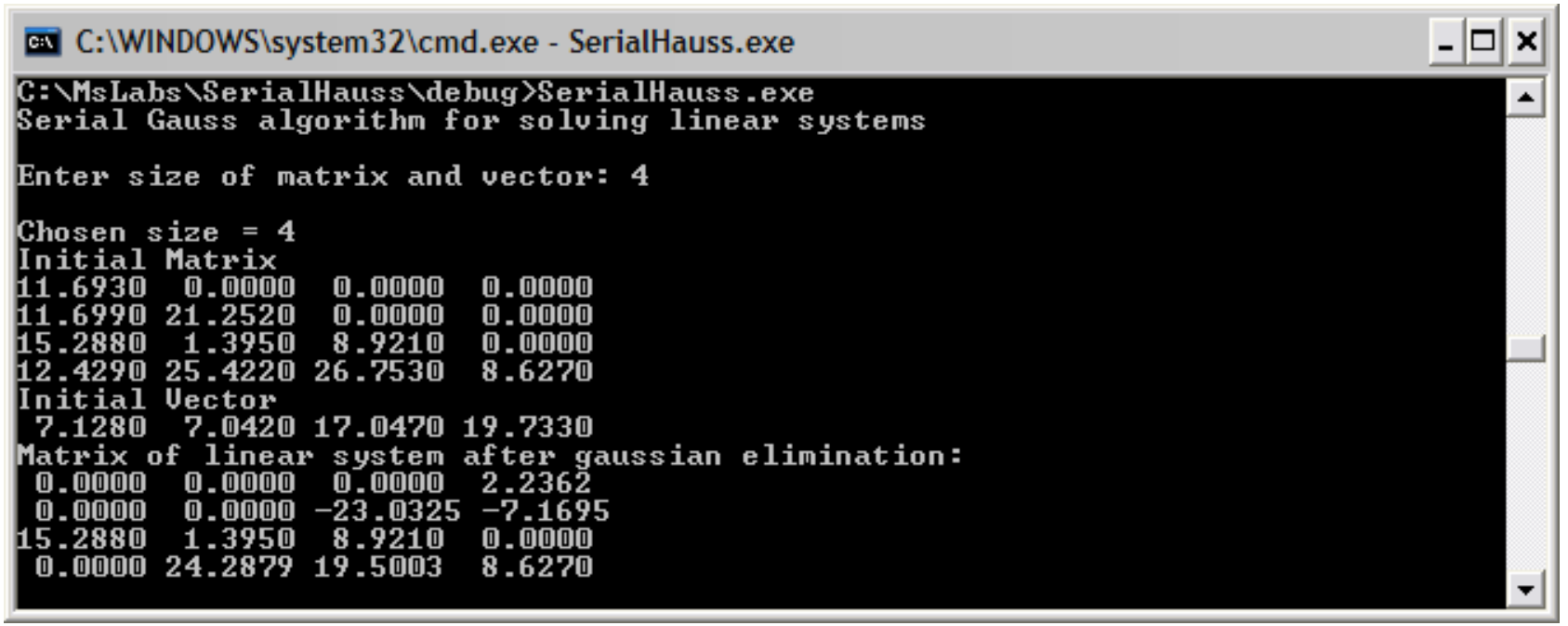
Step 3. Serial Implementation…

Task 5 – Implement the Gaussian Elimination

- Implement the function **`SerialColumnElimination()`**
- Call the function **`SerialColumnElimination()`** on each iteration of Gauss Elimination
- Print out the matrix after all iterations using the function **`PrintMatrix()`**
- Compile and run the application
- Make sure that the Gaussian elimination is executed correctly

Step 3. Serial Implementation…

Task 5 – Implement the Gaussian Elimination



Step 3. Serial Implementation…

Task 6 – Implement the Back Substitution

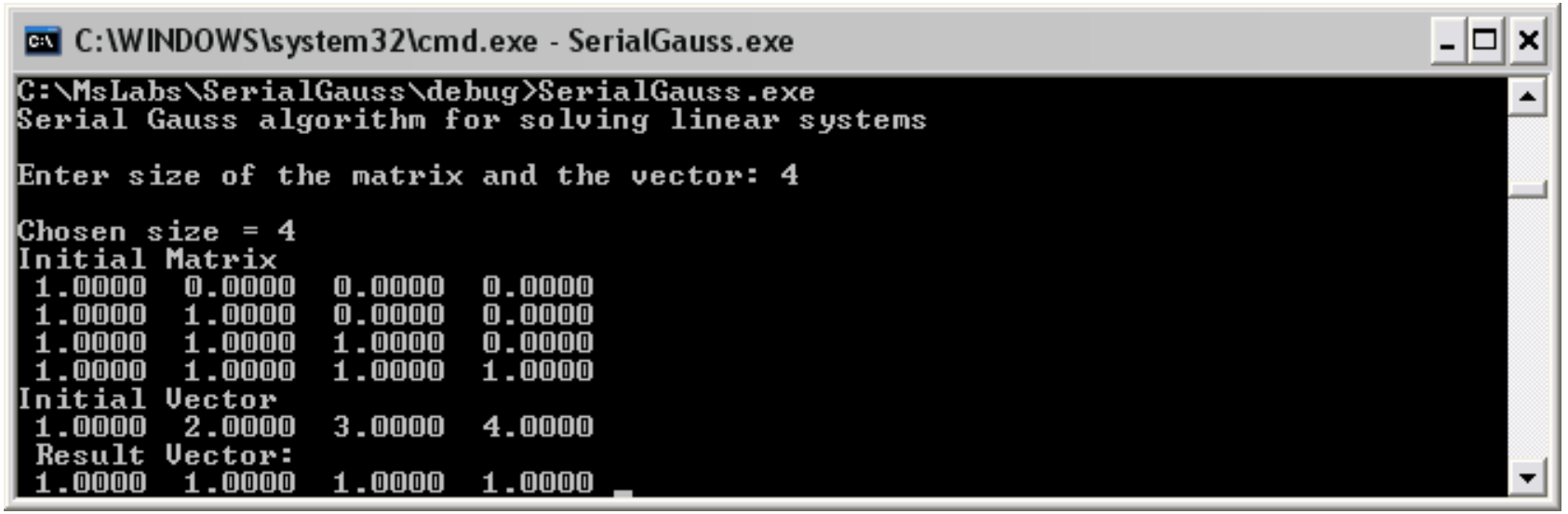
- Implement the function **`SerialBackSubstitution()`** to execute the back substitution

```
// Back substution
void SerialBackSubstitution(double* pMatrix,
  double* pVector, double* pResult, int Size);
```

- Uncomment the call of the function **`SerialBackSubstitution()`** in the function **`SerialResultCalculation()`**
- Print out the matrix and result vector
- Make sure all the result vector elements must be equal to 1

Step 3. Serial Implementation…

Task 6 – Implement the Back Substitution



Step 3. Serial Implementation

Task 7 – 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 initialization of object elements
void RandomDataInitialization(double* pMatrix,
  double* pVector, 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 ALGORITHM

Step 4. Parallel Algorithm…

Subtask definition

- All the computations are reduced to the same computational operations on the rows of the coefficient matrix of the linear equation system
- The data parallelism principle may be applied as the basis of the Gauss algorithm parallel implementation
- All the computations connected with processing a row of the matrix *A* and the corresponding element of the vector *b* may be taken as **the basic computational subtask**

Step 4. Parallel Algorithm…

Analysis of Information Dependencies

- Each iteration with number *i* of the Gaussian elimination stage includes the following stages
  - **The pivot row selection** – the subtasks with the numbers *k*, $k > i$, should exchange their coefficients of the eliminated variable x_i for the maximum value search
  - **Broadcast** – the pivot subtask has to broadcast its pivot row of the matrix *A* and the corresponding element of the vector *b* to all the other subtasks with the numbers *k*, $k > i$
  - Subtraction – after receiving the pivot row the subtasks perform the subtraction of rows

Step 4. Parallel Algorithm…

Analysis of Information Dependencies

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

Step 4. Parallel Algorithm

Scaling and Distributing the Subtask among the Processors

- When the number of processors p is less than the number of basic subtasks m ($p < n$), we can combine the basic subtasks in such a way that each processor would execute several of these tasks
- One-to-all broadcast is the main form of the information communication of the subtasks
- The data transmission network topology must be a hypercube or a complete graph in order to implement the desired information communications among the basic subtasks efficiently

PARALLEL PROGRAM

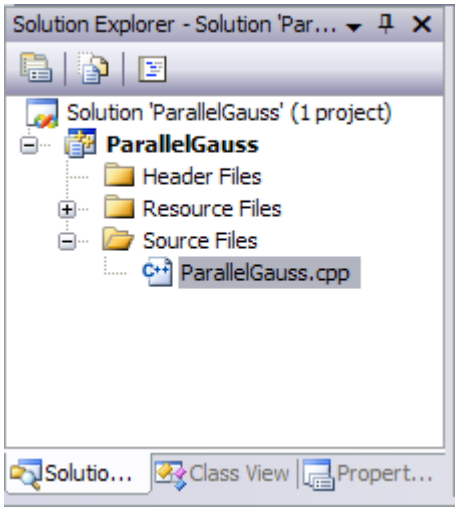
Step 5. Parallel Program…

- Task 1 – Open the Project ParallelGauss
- Task 2 – Input the Initial Data
- Task 3 – Terminate the Parallel Program
- Task 4 – Distribute the Data among the Processes
- Task 5 – Implement the Gaussian Elimination
- Task 6 – Implement the Back Substitution
- Task 7 – Gather the Result
- Task 8 – Test the Parallel Program Correctness
- Task 9 – Carry out the Computational Experiments

Step 5. Parallel Program…

Task 1 – Open the Project ParallelGauss

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



Step 5. Parallel Program…

Task 1 – Open the Project ParallelMatrixMult

- The project contains the following functions
  - **`DummyDataInitialization()`** – simple data initialization
  - **`RandomDataInitialization()`** – random data initialization
  - **`SerialResultCalculation()`** – serial algorithm implementation
  - **`PrintMatrix()`**, **`PrintVector()`** – matrix and vector printing
- **`main()`** function contains declarations of variables **`ProcNum`**, **`ProcRank`**, **`pMatrix`**, **`pVector`**, **`pResult`**, **`Size`**
- The environment of the MPI program is initialized, number of processes **`ProcNum`** and the rank of each process **`ProcRank`** is determined
- Compile and run the applications. Make sure that the initial message is output into the command console

```
"Parallel Gauss algorithm for solving linear systems"
```

Step 5. Parallel Program…

Task 2 – Input the Initial Data

- Determine the variables for storing the blocks and the block sizes

```
double *pProcRows;   // The rows of matrix A on the process
double *pProcVector; // The elements of vector b
                     // on the process
double *pProcResult; // The elements of vector x
                     // on the process
int     RowNum;      // The Number of  the matrix rows on
                     // the  current process
```

Step 5. Parallel Program…

Task 2 – Input the Initial Data

- Develop the function **`ProcessInitialization()`**
  - Input the matrix size and the vector size (on the root process)
  - Broadcast sizes
  - Calculate the number of matrix rows, which will be processed by a given process
  - Allocate the memory for storing the matrix, the vectors and their blocks
  - Generate the initial matrix and vector elements

```
void ProcessInitialization(double* &pMatrix,
  double* &pVector, double* &pResult, double* &pProcRows,
  double* &pProcVector, double* &pProcResult, int &Size,
  int &RowNum);
```

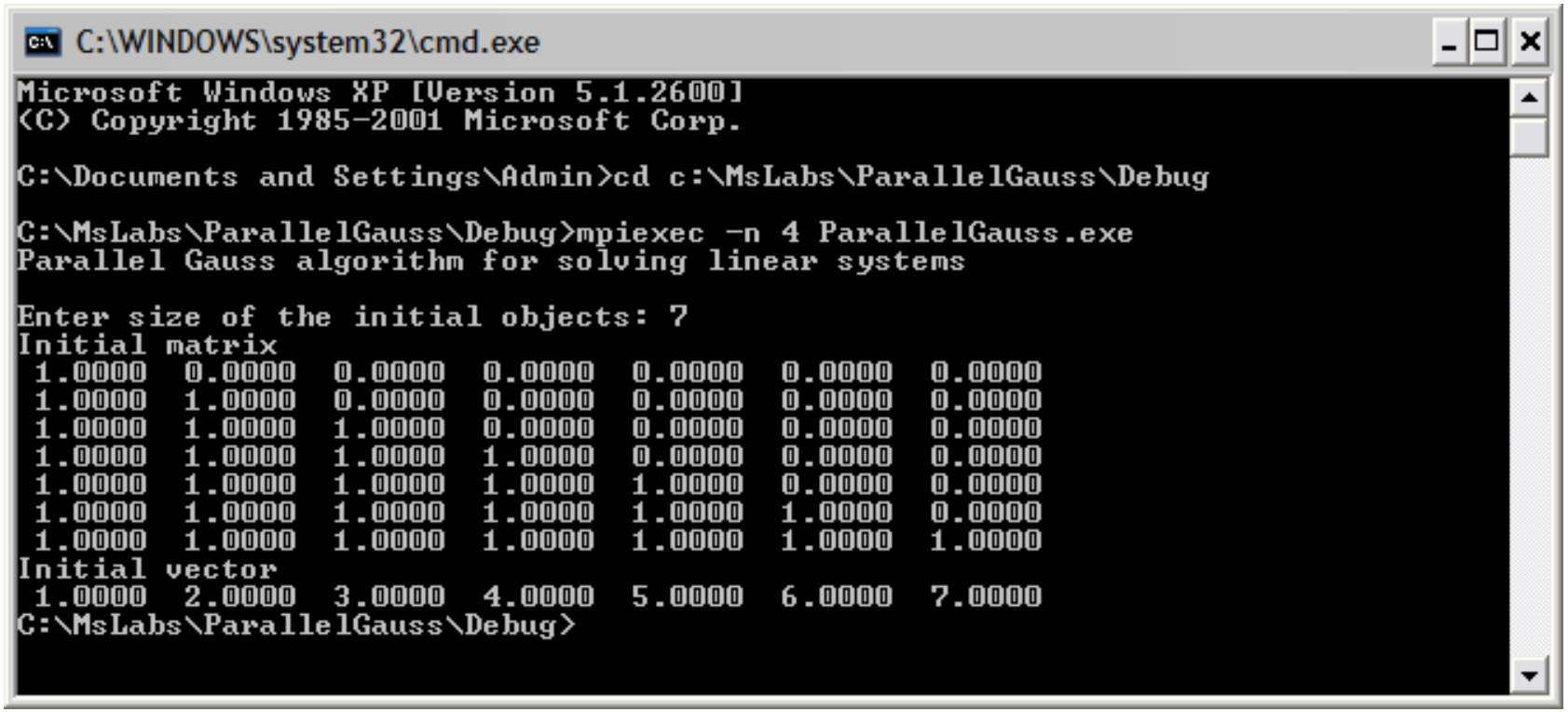
Step 5. Parallel Program…

Task 2 – Input the Initial Data

- Implement the function **`ProcessInitialization()`**
- Call the function from the main function of application
- To control the correctness of the initial data input use the function of the formatted matrix output **`PrintMatrix()`** and the vector **`PrintVector()`**
- Print out the linear equation system matrix and the right part vector on the root process
- Compile and run the application

Step 5. Parallel Program…

Task 2 – Input the Initial Data



Step 5. Parallel Program…

Task 3 – Terminate the Parallel Program

- Modify the function for correct program termination **`ProcessTermination()`**
- Deallocate the memory for storing the initial matrix **`pMatrix`** (on the root process), and the memory for storing the initial vector **`pVector`**, the result vector **`pResult`**, matrix stripe **`pProcRows`**, the blocks of the right part vector **`pProcVector`** and the result vector block **`pProcResult`**

```
// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector,
  double* pResult, double* pProcRows, double* pProcVector,
  double* pProcResult);
```

Step 5. Parallel Program…

Task 4 – Distribute the Data among the Processes

- In accordance with the parallel computation scheme the system of linear equations must be distributed among the processes in horizontal stripes (divided into continuous sequences of rows)
- To distribute the matrix **pMatrix** and the vector **pVector** use the function **MPI_Scatterv()**
- Implement the function **DataDistribution()**

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

Step 5. Parallel Program…

Task 4 – Distribute the Data among the Processes

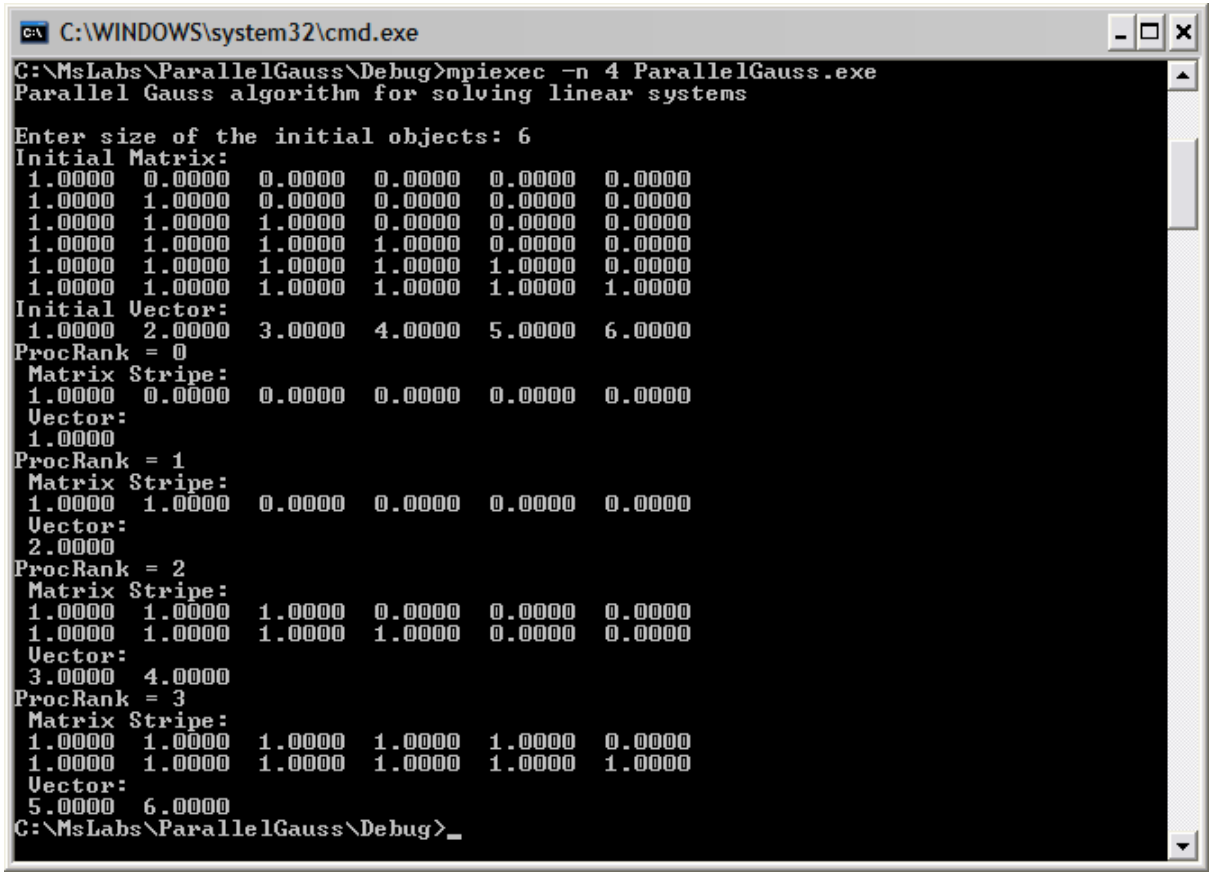
- Call the function **`DataDistribution()`** from the main program
- To test the correctness of the data distribution among the processes implement the “debugging print” function **`TestDistribution()`**
  - Print the initial matrix **`pMatrix`** and vector **`pVector`** on the root process
  - Print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

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

Step 5. Parallel Program…

Task 4 – Distribute the Data among the Processes

- Make sure that the data is distributed correctly



Step 5. Parallel Program…

Task 5 – Implement the Gaussian Elimination

- The computational scheme of the Gauss algorithm consists of the two stages: the Gaussian elimination and the back substitution
- Implement the **`ParallelResultCalculation()`** function

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

Step 5. Parallel Program…

Task 5 – Implement the Gaussian Elimination

- To develop the parallel version of Gauss algorithm we will need two auxiliary arrays **`pParallelPivotPos`** and **`pProcPivotIter`**

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

- Allocate the memory for storing these objects before the execution of the parallel Gauss method stages in the function **`ParallelResultCalculation()`** function and initialize them
- Deallocate the memory After the termination of the back substitution

Step 5. Parallel Program…

Task 5 – Implement the Gaussian Elimination

- To reduce the matrix of the linear equation system to the upper triangle form using equivalent transformations implement the function **`ParallelGaussianElimination()`**

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

Step 5. Parallel Program…

Task 5 – Implement the Gaussian Elimination

- The function **`ParallelGaussianElimination()`**
  - Select the local pivot rows on each process
  - Choose the maximum element among the obtained pivot elements and determine, at which process it is located. Use the function **`MPI_Allreduce()`**

```
int MPI_AllReduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype type, MPI_Op op, MPI_Comm comm);
```

- **`sendbuf`** – memory buffer with the transmitted message
- **`recvbuf`** – memory buffer with the resulting message (only for the root process)
- **`count`** – the number of the data elements in the message
- **`type`** – the type of the data elements in the message
- **`op`** – the operation, which should be carried out over the data
- **`comm`** – the communicator, within of which the operation is executed

Step 5. Parallel Program…

Task 5 – Implement the Gaussian Elimination

- The function **`ParallelGaussianElimination()`**
  - Broadcast the pivot row
  - Carry out the subtraction of rows on each process
- Implement the subtraction with the help of the function **`ParallelEliminateColumns()`**

```
// Fuction for column elimination
void ParallelEliminateColumns(double* pProcRows,
  double* pProcVector, double* pPivotRow, int Size,
  int RowNum, int Iter);
```

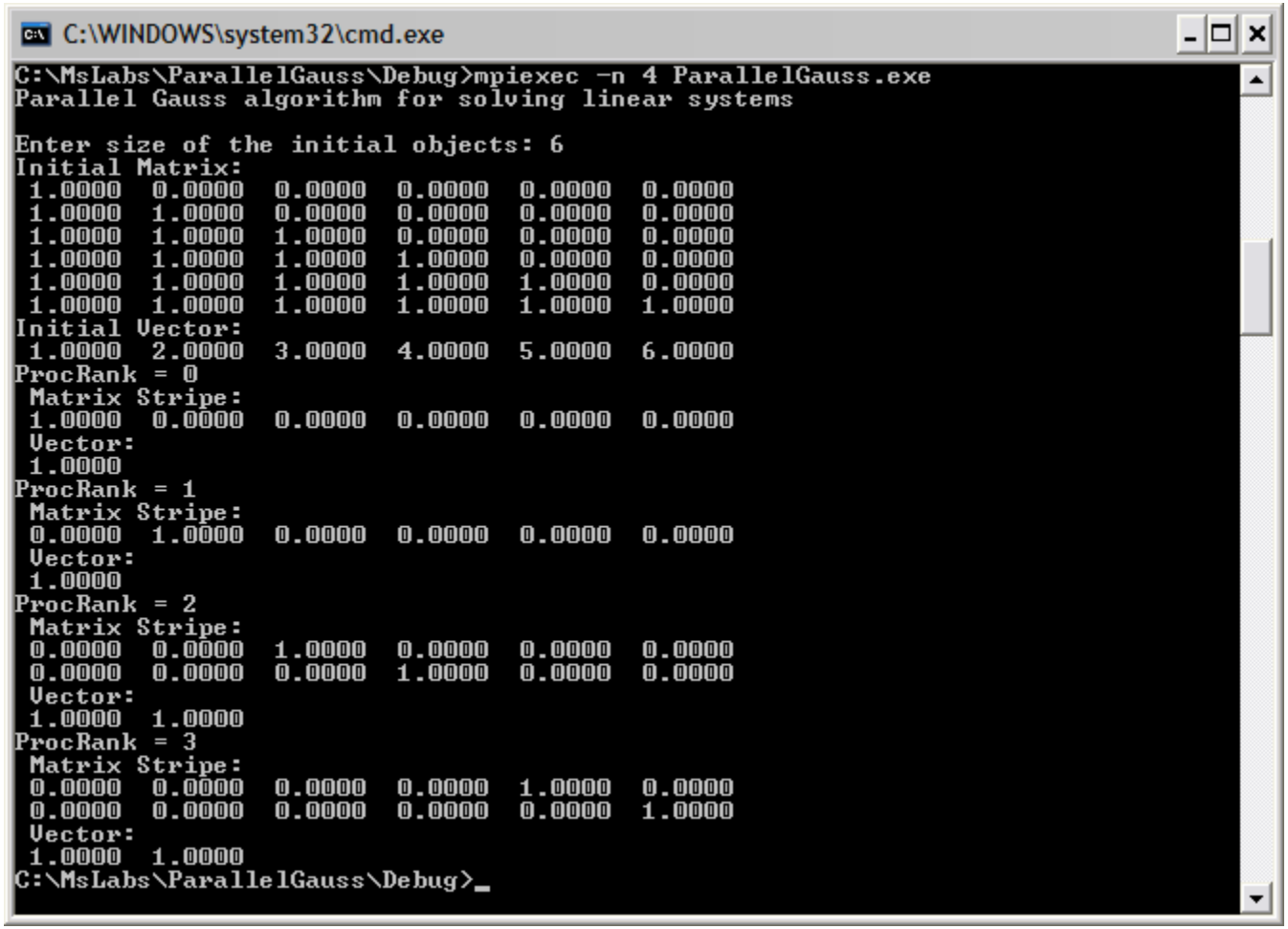
Step 5. Parallel Program…

Task 5 – Implement the Gaussian Elimination

- Implement the function **`ParallelEliminateColumns()`**
- Call the function **`ParallelEliminateColumns()`** on each iteration of Gauss Elimination
- Call the function **`ParallelResultCalculation()`** from the main function of the application
- To check the correctness of executing the Gaussian elimination, call the function **`TestDistribution()`**
- Compile and run the application
- Make sure the developed functions are operated correctly

Step 5. Parallel Program…

Task 5 – Implement the Gaussian Elimination



Step 5. Parallel Program…

Task 6 – Implement the Back Substitution

- The processes carry out the calculations necessary for obtaining the values of the unknown variables
- As soon as any process determines the value of its variable, this variable must be broadcast to all the processes
- The processes substitute the obtained value of the new unknown variable and correct the values for the elements of the right part vector

Step 5. Parallel Program…

Task 6 – Implement the Back Substitution

- The back substitution execution consists of **`Size`** iterations
- At each iteration it is necessary to determine the row, which makes possible to calculate the value of the next result vector element
  - The row number is stored in the array **`pParallelPivotIter`**
  - Using the row number determine the number of process, where the row is stored, and the number of the row in the stripe **`pProcRows`** of the process
- Implement the function **`FindBackPivotRow()`**

```
// Function to find the pivot row of the back substitution
void FindBackPivotRow(int RowIndex, int Size,
  int &IterProcRank, int &IterPivotPos);
```

Step 5. Parallel Program…

Task 6 – Implement the Back Substitution

- Implement the functions **`FindBackPivotRow()`** and **`ParallelBackSubstitution()`**
- After the execution of the parallel Gauss algorithm, print the result vector blocks on each parallel process
- Compile and run the application
- Test the correctness of the program execution

Step 5. Parallel Program…

Task 7 – Gather the Result

- After the execution of the back substitution of the Gauss algorithm the result vector blocks are located on each process
- It is necessary to collect the result vector on the root process
- Implement the function **`ResultCollection()`**
- Use the function **`MPI_Gatherv()`**

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

Step 5. Parallel Program…

Task 7 – Gather the Result

- Add the call of the function for gathering the result vector into the main function of the application
- Implement the function **`PrintResultVector()`** to print the result vector
  - the order of the unknowns in **`pResult`** vector is the same with the order of pivot rows selection
  - this order is stored in the **`pParallelPivotPos`** array

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

Step 5. Parallel Program…

Task 7 – Gather the Result

- Add the call of the function **`ResultCollection()`** to the main function of the application
- Compile and run the application
- Check the correctness of the algorithm execution: if the function **`DummyDataInitialization()`** is used to generate the initial data, all the result vector elements must be equal to 1

Step 5. Parallel Program…

Task 8 – Test the Parallel Program Correctness

- To test the correctness of the program execution develop the function **`TestResult()`**.
- It will perform the multiplication of the linear system matrix by the vector of unknowns, that has been obtained by the means of Gauss method
- The result of the multiplication will be stored in the variable **`pRightPartVector`**. Then, the function will compare the vector of right parts **`pVector`** and the result of multiplication **`pRightPartVector`** element by element
- The result of the function is the print of the diagnostic message

Step 5. Parallel Program…

Task 8 – Test the Parallel Program Correctness

- Comment on the calls of the functions, using the debugging print, which have been previously used
- Implement the function **`TestResult()`** and call it in main program
- 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 5. Parallel Program

Task 9 – 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

- Parallel Gauss method of solving the linear equation systems is considered
- Serial and parallel Gauss algorithm are implemented
- Computational experiments are performed, comparison of serial and parallel algorithms is made

Exercises

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

References

1. Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V. (1999). Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc for Industrial & Applied Math.
2. Quinn, M.J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
3. Kumar V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc. (2nd edn., 2003)
4. Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.
5. Foster, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.