


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

05 Practice
Parallel Algorithms of Matrix-Vector Multiplication

With the support of Microsoft

Sysoyev A.V.
Software department

Contents

- Matrix-Vector Multiplication Problem Statement
- Serial Implementation
- Parallel Algorithm
- Parallel Program

MATRIX-VECTOR MULTIPLICATION PROBLEM STATEMENT

Step 1. Problem Statement…

- Matrix-vector multiplication

$$c = A \cdot b$$

- or

$$\begin{pmatrix} c_0 \\ c_1 \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} a_{0,0}, & a_{0,1}, & \ldots, & a_{0,n-1} \\ & & \ldots & \\ a_{m-1,0}, & a_{m-1,1}, & \ldots, & a_{m-1,n-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_{n-1} \end{pmatrix}$$

- Each *i*-th element of the vector *c* is the result of scalar multiplications of *i*-th matrix *A* row (let us denote this row as a_i) and the vector *b*

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, \quad 0 \le i < m$$

Step 1. Problem Statement

- Obtaining the result vector *c* assumes the execution of *m* operations of the same type
- The pseudo code for the given matrix-vector multiplication algorithm may be as follows

```
// Serial algorithm of matrix-vector multiplication
for (i = 0; i < m; i++)
{
  c[i] = 0;
  for (j = 0; j < n; j++)
    c[i] += A[i][j] * b[j]
}
```

- Computational complexity is $O(mn)$

SERIAL IMPLEMENTATION

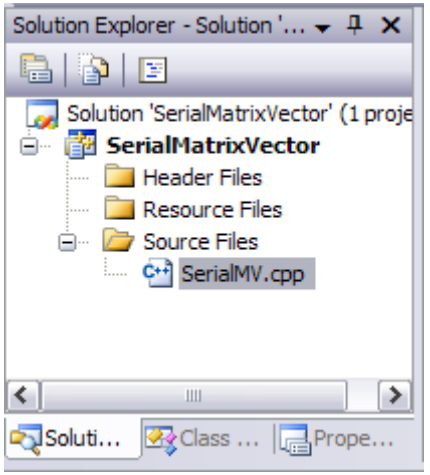
Step 2. Serial Implementation…

- Task 1 – Open the Project SerialMatrixVectorMult
- Task 2 – Input the Matrix and Vector
- Task 3 – Input the Initial Data
- Task 4 – Terminate the Program Execution
- Task 5 – Implement the Matrix-Vector Multiplication
- Task 6 – Carry out the Computational Experiments

Step 2. Serial Implementation…

Task 1 – Open the Project SerialMatrixVectorMult

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



Step 2. Serial Implementation…

Task 1 – Open the Project SerialMatrixVectorMult

- Next variables will be used in the program

```
double *pMatrix;  // Initial matrix
double *pVector;  // Initial vector
double *pResult;  // Result vector
int Size;         // Sizes of initial matrix and vector
```

- Notice that the matrix **`pMatrix`** is stored rowwise in an one-dimensional array
- 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 matrix-vector multiplication program\n");
getch();
```

Step 2. Serial Implementation…

Task 2 – Input the Matrix and Vector

- In order to set the initial data of the matrix-vector multiplication program 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 2. Serial Implementation…

Task 2 – Input the Matrix and Vector

- Determine the sizes of the objects

```
// Function for process initialization
void ProcessInitialization(double* &pMatrix,
  double* &pVector, double* &pResult, int &Size) {
  // Setting the size of the initial matrix and the vector
  printf("\nEnter the size of the initial objects: ");
  scanf("%d", &Size);
  printf("\nChosen objects' size = %d", Size);
}
```

Step 2. Serial Implementation…

Task 2 – Input the Matrix and Vector

- 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 2. Serial Implementation…

Task 2 – Input the Matrix and Vector

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

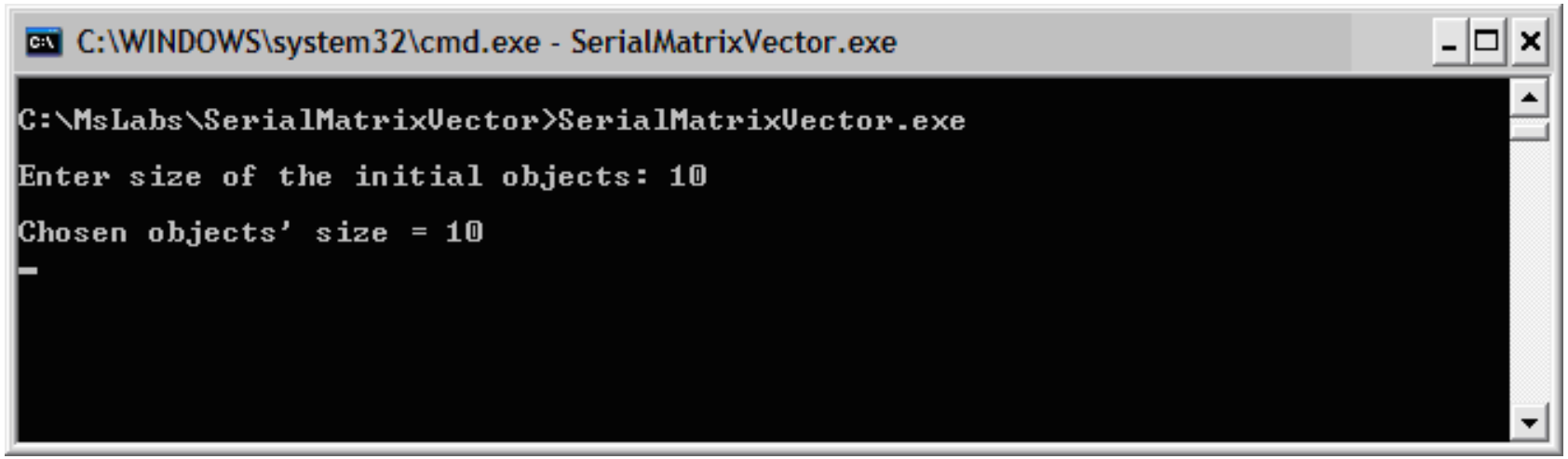
```
void main() {
  double* pMatrix;  // Initial matrix
  double* pVector;  // Initial vector
  double* pResult;  // Result vector
  int Size;         // Sizes of initial matrix and vector

  printf ("Serial matrix-vector multiplication program\n");
  // Process initializattion
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

Step 2. Serial Implementation…

Task 2 – Input the Matrix and Vector

- Compile and run the application



Step 2. 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 2. Serial Implementation…

Task 3 – Input the Initial Data

- Setting matrix and vector elements by the following template

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

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Function to set the matrix and vector elements in rather a simple way

```
// Function for simple data initialization
void DummyDataInitialization(double* pMatrix,
  double* pVector, int Size) {
  int i, j;
  for (i = 0; i < Size; i++) {
    pVector[i] = 1;
    for (j = 0; j < Size; j++)
      pMatrix[i * Size + j] = i;
  }
}
```

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Call the function **`DummyDataInitialization()`** after allocating memory inside the function **`ProcessInitialization()`**

```
// 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
  <…>

  // Initialization of matrix and vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}
```

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Develop two more functions, which help to control data input
  - The function of the formatted matrix output **`PrintMatrix()`**
    - The arguments of **`PrintMatrix()`** is the matrix **`pMatrix`**, the number of rows **`RowCount`** and the number of columns **`ColCount`**
  - The function of the formatted vector output **`PrintVector()`**
    - The arguments of **`PrintVector()`** is the vector **`pVector`** and the number of elements **`Size`**

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Formatted matrix output

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

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Formatted vector output

```
// Function for formatted vector output
void PrintVector(double* pVector, int Size) {
  int i;
  for (i = 0; i < Size; i++)
    printf("%7.4f ", pVector[i]);
  printf("\n");
}
```

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Add the call of these functions to the main application function

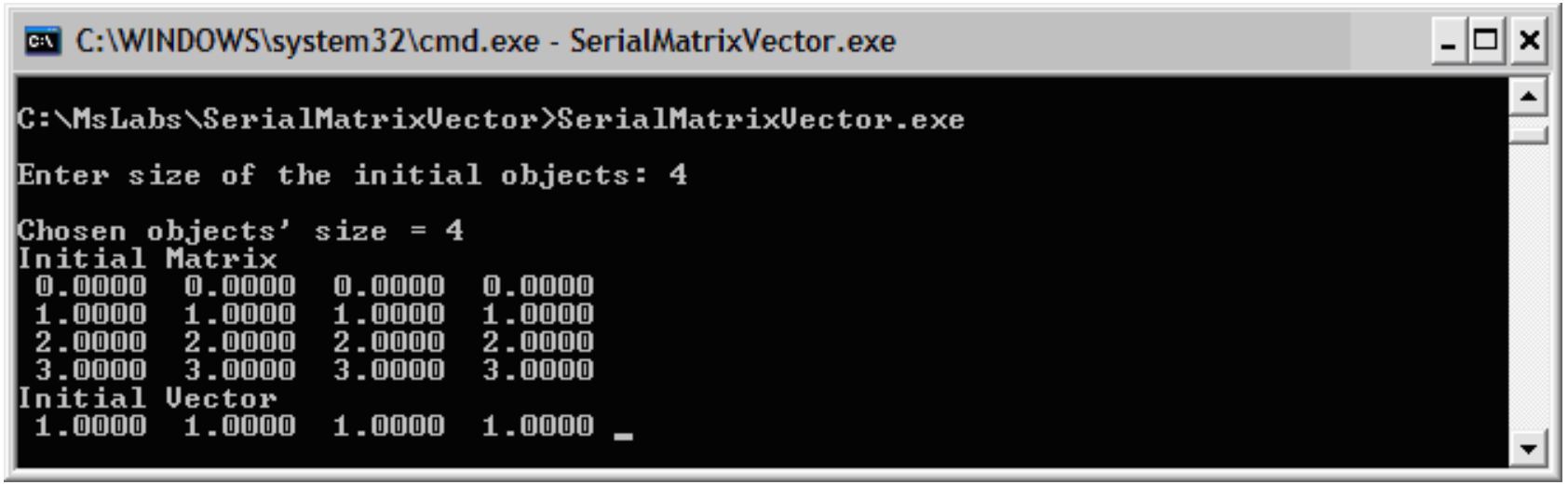
```
// Process initialization
ProcessInitialization(pMatrix, pVector, pResult, Size);

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

Step 2. Serial Implementation…

Task 3 – Input the Initial Data

- Compile and run the application



Step 2. 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 2. Serial Implementation…

Task 4 – Terminate the Program Execution

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

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

Step 2. Serial Implementation…

Task 5 – Implement the Matrix-Vector Multiplication

- To multiply the matrix by the vector develop the function **`ResultCalculation()`**, which gets the initial matrix **`pMatrix`** and the vector **`pVector`**, the size **`Size`** and the result vector **`pResult`**

```
// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int i, j;
  for (i = 0; i < Size; i++) {
    pResult[i] = 0;
      for (j = 0; j < Size; j++)
        pResult[i] += pMatrix[i * Size + j] * pVector[j];
  }
}
```

Step 2. Serial Implementation…

Task 5 – Implement the Matrix-Vector Multiplication

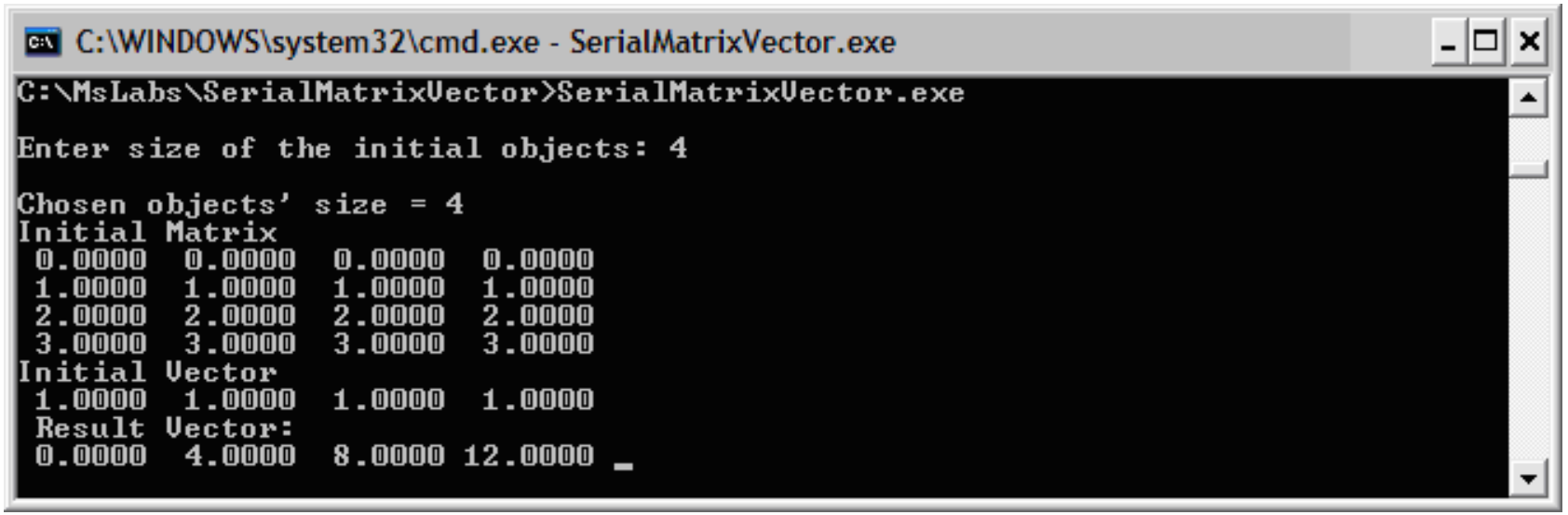
- Call the function of matrix-vector multiplication computation from the **`main()`** function
- In order to control the correctness of the function implementation print out the result vector

```
// Matrix and vector output
<…>
// Matrix-vector multiplication
ResultCalculation(pMatrix, pVector, pResult, Size);
// Printing the result vector
printf("\n Result Vector: \n");
PrintVector(pResult, Size);
```

Step 2. Serial Implementation…

Task 5 – Implement the Matrix-Vector Multiplication

- Compile and run the application



Step 2. Serial Implementation…

Task 6 – 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) {
  int i, j;
  srand(unsigned(clock()));
  for (i = 0; i < Size; i++) {
    pVector[i] = rand() / double(1000);
    for (j = 0; j < Size; j++)
      pMatrix[i * Size + j] = rand() / double(1000);
  }
}
```

Step 2. Serial Implementation

Task 6 – Carry out the Computational Experiments

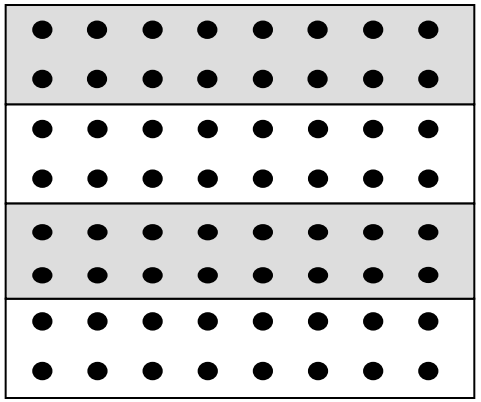
- Call this function instead of the function **`DummyDataInitialization()`**, which has been developed previously
- Add time measurement and printing
- Carry out the computational experiments with large objects
- Fill the table with results of experiments

PARALLEL ALGORITHM

Step 3. Parallel Algorithm…

Analysis of Information Dependencies

- Let's use the matrix presentation as **continuous sets** (**horizontal stripes**) of rows



- In this case **base subtask** is scalar multiplication operation (matrix row by vector)

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, \quad 0 \le i < m$$

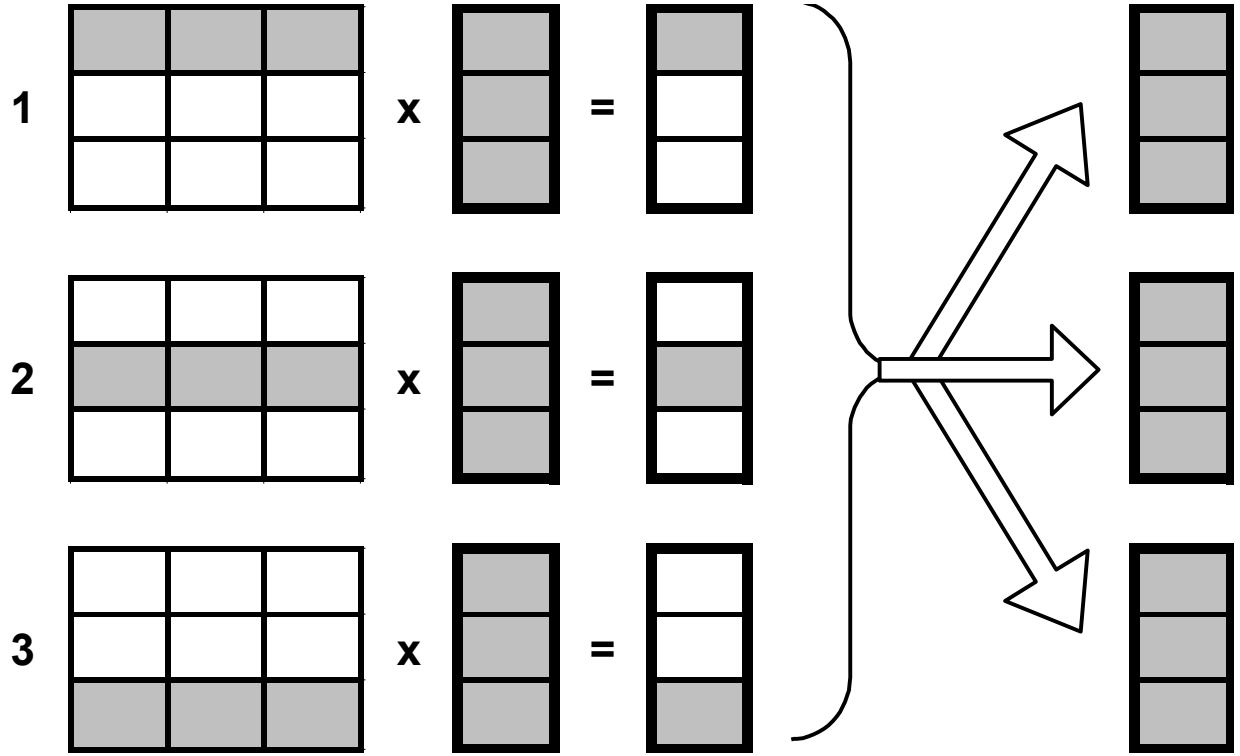
Step 3. Parallel Algorithm…

Analysis of Information Dependencies

- In order to execute the basic subtasks of scalar multiplication the processor should contain the corresponding row of the matrix **`pMatrix`** and a copy of the vector **`pVector`**
- After the termination of the computations, each basic subtask determines one of the elements of the result vector **`pResult`**.
- In order to combine the calculation results and obtaining the whole vector **`pResult`** on each processor of the computational system, it is necessary to execute the all gather operation, when each processor transmits its computed element of the vector **`c`** to the rest of the processors

Step 3. Parallel Algorithm…

Computation Organization



Step 3. 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 ($p < m$), we can combine the basic subtasks in such a way that each processor would execute several of these tasks
- In this case upon the completion of computations, each extended basic subtask determines several elements of the result vector **`pResult`**
- Subtasks distribution among the processors of the computational system may be executed in an arbitrary way

PARALLEL PROGRAM

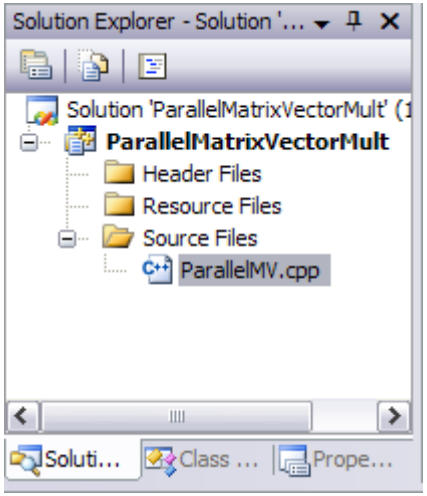
Step 4. Parallel Program…

- Task 1 – Open the Project ParallelMatrixVectorMult
- Task 2 – Initialize and Terminate the Parallel Program
- Task 3 – Determine the Number of Processes
- Task 4 – Input the Matrix and Vector Size
- Task 5 – Input the Initial Data
- Task 6 – Terminate the Calculations
- Task 7 – Distribute the Data among the Processes
- Task 8 – Implement the Parallel Matrix-Vector Multiplication
- Task 9 – Gather the Results
- Task 10 – Test the Parallel Program Correctness
- Task 11 – Implement the Computations for Any Matrix Sizes
- Task 12 – Carry out the Computational Experiments

Step 4. Parallel Program…

Task 1 – Open the Project ParallelMatrixVectorMult

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



Step 4. Parallel Program…

Task 1 – Open the Project ParallelMatrixVectorMult

- The project contains the following functions
  - **`DummyDataInitialization()`** – simple data initialization
  - **`RandomDataInitialization()`** – random data initialization
  - **`ResultCalculation()`** – serial algorithm implementation
  - **`PrintMatrix()`** and **`PrintVector()`** – matrix and vector printing
- **`main()`** function contains declarations of variables **`pMatrix`**, **`pVector`**, **`pResult`**, **`Size`**

Step 4. Parallel Program…

Task 2 – Initialize and Terminate the Parallel Program

- Add the header file mpi.h to the program
- It is necessary to initialize the environment of the MPI program execution in the **`main()`** function of the parallel program and to terminate its use of the environment on the program exit

```
void main (int argc, char* argv[]) {
  //Variable declaration
  <…>
  MPI_Init(&argc, &argv);
  printf("Parallel matrix-vector multiplication program\n");
  MPI_Finalize();
}
```

Step 4. Parallel Program…

Task 2 – Initialize and Terminate the Parallel Program

- Press the button **Start**, and then execute the command **Run**,
- Type the name of the program **cmd** in the dialog window, which appears on the screen
- In the command line of the cmd program window go to the folder, which contains the developed program
- Run the MPI program. The command line may looks like following

```
mpiexec -n <NumOfProcesses> <NameOfExecutive> <Arguments>
```

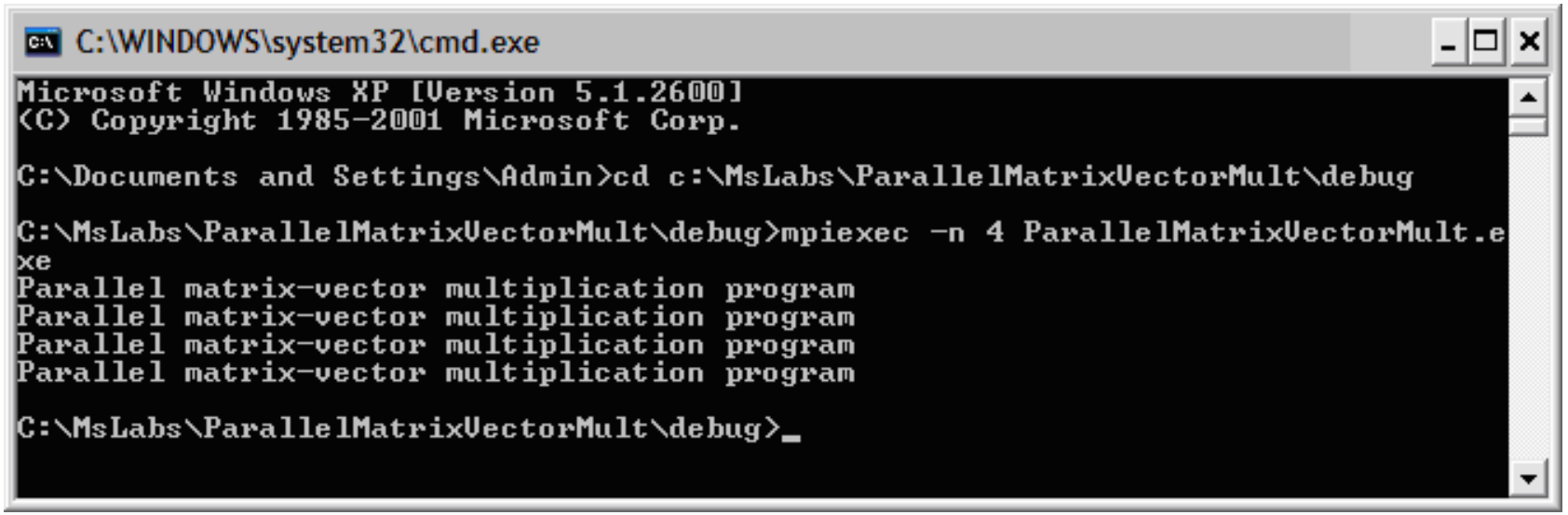
- To execute the parallel program using 4 processes type the command line

```
mpiexec -n 4 ParallelMatrixVectorMult.exe
```

Step 4. Parallel Program…

Task 2 – Initialize and Terminate the Parallel Program

- Parallel program execution



Step 4. Parallel Program…

Task 3 – Determine the Number of Processes

- Declare **ProcNum** and **ProcRank** as global variables
- Determine the Number of Processes and the rank of each process

```
int ProcNum;       // Number of available processes
int ProcRank;      // Rank of current process
void main(int argc, char* argv[]) {
  <…>
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  printf("Parallel matrix-vector multiplication program\n");
  MPI_Finalize();
}
```

Step 4. Parallel Program…

Task 3 – Determine the Number of Processes

- Change the code in **`main()`** function to
  - Print out the initial message and number of processes by Determine the Number of Processes by the process with the rank 0 only
  - Print out the rank by each process

Step 4. Parallel Program…

Task 4 – Input the Matrix and Vector Size

- The function **`ProcessInitialization()`** serves as previously for initializing the computation

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

- Input the matrix and vector sizes by the process with rank 0
- Assume that the size of the objects is divisible without remainder by the number of processes
- Assume that the size of the objects is greater then the number of processes

Step 4. Parallel Program…

Task 4 – Input the Matrix and Vector Size

```
void ProcessInitialization(double* &pMatrix,
  double* &pVector, double* &pResult, int &Size) {
  if (ProcRank == 0) {
    do {
      printf("\nEnter size of the initial objects: ");
      scanf("%d", &Size);
      if (Size < ProcNum)
        printf("Size of the objects must be greater than
                number of processes! \n ");
      if (Size % ProcNum != 0)
        printf("Size of objects must be divisible by
                number of processes! \n");
    } while ((Size < ProcNum) || (Size % ProcNum != 0));
  }
}
```

Step 4. Parallel Program…

Task 4 – Input the Matrix and Vector Size

- After the value of the variable **`Size`** is defined correctly, it is necessary to transfer the value to other processes
- For this purpose use the MPI broadcast function

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type,
  int root, MPI_Comm comm);
```

- **`buf`** – the address of the memory buffer, which contains the data of the message to be transmitted
- **`count`** – the number of the data elements in the message
- **`type`** – the type of the data elements in the message
- **`root`** – the rank of the process, which carries out data broadcasting
- **`comm`** – the communicator, within of which the data is transmitted

Step 4. Parallel Program…

Task 4 – Input the Matrix and Vector Size

- Add the call of the function of the computation initialization instead of the lines, which print the number of the processes and their ranks
- Compile and run the application
- Make sure that all the invalid situations are processed correctly
- For this purpose run the application several times setting different number of the parallel processes and various matrix and vector sizes

Step 4. Parallel Program…

Task 5 – Input the Initial Data

- According to the scheme of the parallel computations the initial matrix is distributed among all the processes by continuous set of rows (a horizontal stripe)
- The horizontal stripes of rows on each of the processes will be stored in the variable **`pProcRows`** (**`pProcRows`** is the matrix, which contains **`RowNum`** rows and **`Size`** columns rowwise)
- The vector **`pVector`** is copied from the root process to the other processes
- As a result of matrix stripe by vector multiplication, each process obtains **`RowNum`** elements of the result vector. These elements will be stored in the array **`pProcResult`**

Step 4. Parallel Program…

Task 5 – Input the Initial Data

```
void main (int argc, char* argv[]) {
  double* pMatrix; // The first argument - initial matrix
  double* pVector; // The second argument - initial vector
  double* pResult; // Result vector for matrix-vector
                   // multiplication
  int Size;        // Sizes of initial matrix and vector
  double* pProcRows;   // Matrix stripe on current process
  double* pProcResult; // Block of result vector
  int RowNum;          // Number of rows in matrix stripe
  <…>
}
```

Step 4. Parallel Program…

Task 5 – Input the Initial Data

- Change the list of arguments of the function **`ProcessInitialization`** so that this function can determine the value of the variable **`RowNum`** and allocate memory for storing new objects

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

- Only the root process allocate the memory for initial matrix and set the matrix and vector elements (using function **`DummyDataInitialization()`** or function **`RandomDataInitialization()`**)

Step 4. Parallel Program…

Task 6 – Terminate the Calculations

- 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`** and the result vector block **`pProcResult`**

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

Step 4. Parallel Program…

Task 7 – Distribute the Data among the Processes

- In accordance with the parallel computation scheme the matrix must be distributed among the processes in equal horizontal stripes, and the initial vector must be copied onto all the processes
- Use the function **`MPI_Bcast()`** to copy the vector **`pVector`**
- To distribute the matrix **`pMatrix`** let's use the "Scatter" function

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype,
                void *rbuf, int rcount, MPI_Datatype rtype,
                int root, MPI_Comm comm);
```

- **`sbuf, scount, stype`** – the parameters of the transmitted message (scount defines the number of elements transmitted to each process)
- **`rbuf, rcount, rtype`** – the parameters of the received message
- **`root`** – the rank of the process, on which the result must be obtained
- **`comm`** – the communicator, within of which the operation is executed

Step 4. Parallel Program…

Task 7 – Distribute the Data among the Processes

- Add the call of the functions **`MPI_Bcast()`** and **`MPI_Scatter()`** to the function **`DataDistribution()`**

```
// Function for data distribution between the processes
void DataDistribution(double* pMatrix, double* pProcRows,
  double* pVector, int Size, int RowNum) {
  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(pMatrix, RowNum*Size, MPI_DOUBLE, pProcRows,
    RowNum*Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Step 4. Parallel Program…

Task 7 – Distribute the Data among the Processes

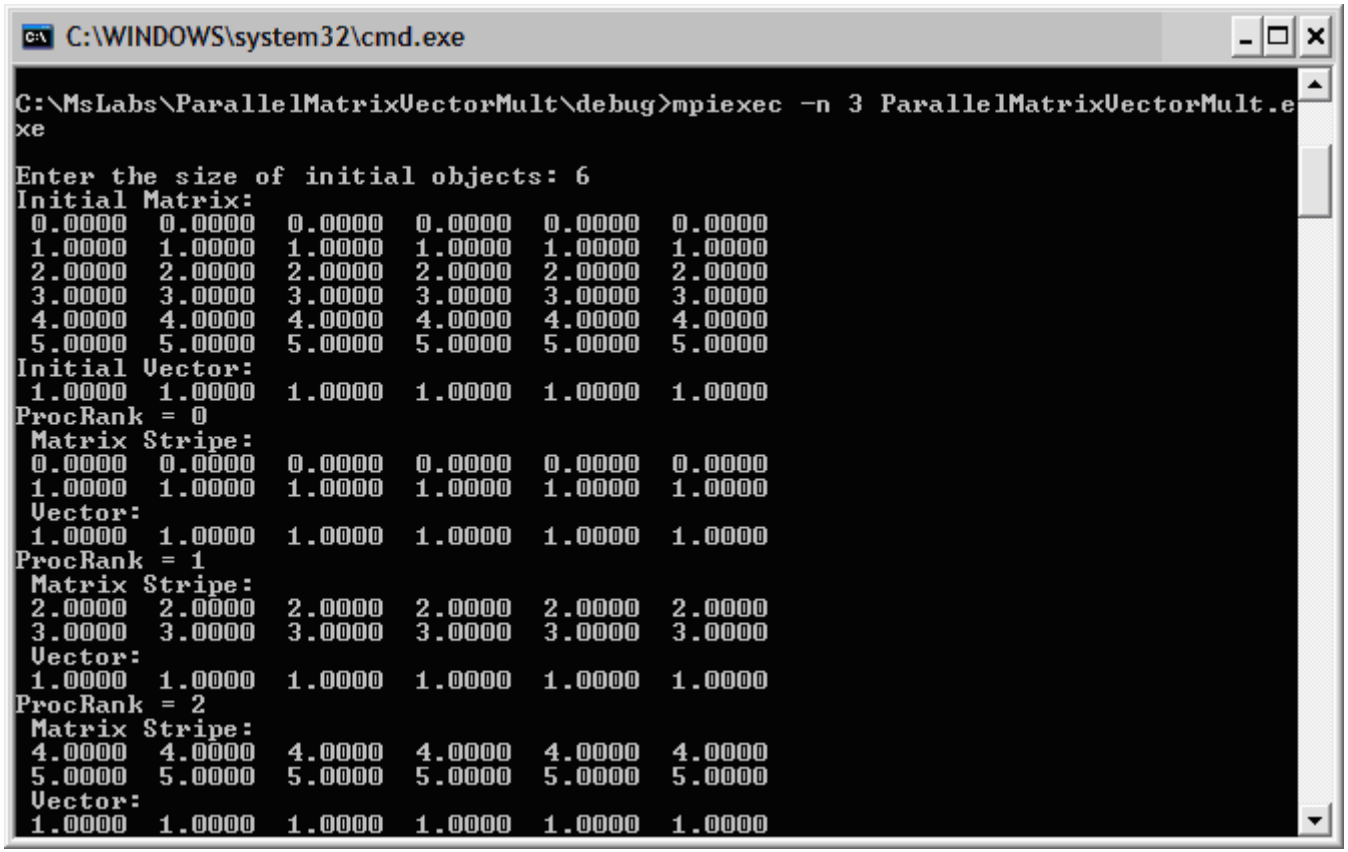
- Call the function **`DataDistribution()`** from the main program
- To test the correctness of the data distribution among the processes implement the “debugging print” function **`TestDistribution()`**
  - Print the initial matrix **`pMatrix`** and vector **`pVector`** on the root process
  - Print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

```
void TestDistribution(double* pMatrix, double* pVector,
  double* pProcRows, int Size, int RowNum);
```

Step 4. Parallel Program…

Task 7 – Distribute the Data among the Processes

- Check the data distribution correctness



Step 4. Parallel Program…

Task 8 – Implement the Parallel Matrix-Vector Multiplication

- In order to calculate a block of the result vector, it is necessary to have access to the matrix stripe **`pProcRows`**, the vector **`pVector`** and the result vector block **`pProcResult`**
- Implement the **`ParallelResultCalculation()`** function

```
// Function for parallel matrix-vector multiplication
void ParallelResultCalculation(double* pProcRows,
  double* pVector, double* pProcResult, int Size,
  int RowNum);
```

Step 4. Parallel Program…

Task 8 – Implement the Parallel Matrix-Vector Multiplication

- Develop the function of testing the partial results, which were obtained by each of the processes, **`TestPartialResults()`**

```
void TestPartialResults (double* pProcResult, int RowNum);
```

- Call the functions **`ParallelResultCalculation()`** and **`TestPartialResults()`** in the main program

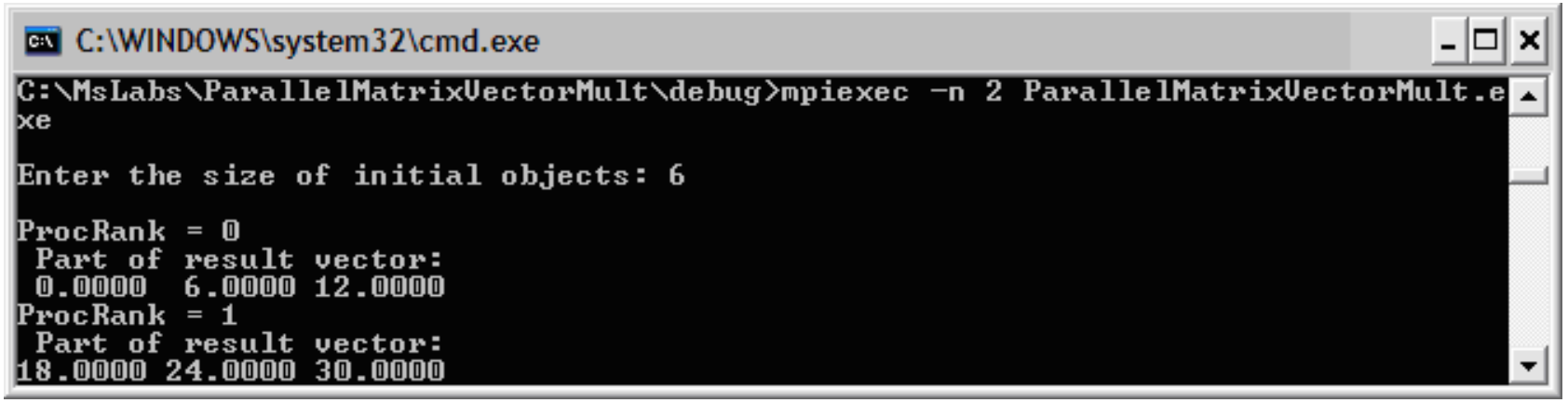
```
DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
// TestDistribution(pMatrix, pVector, pProcRows, Size, RowNum);

// Parallel matrix vector multiplication
ParallelResultCalculation(pProcRows, pVector, pProcResult, Size,
  RowNum);
TestPartialResults(pProcResult, RowNum);
```

Step 4. Parallel Program…

Task 8 – Implement the Parallel Matrix-Vector Multiplication

- The result vector block, which contains the values from *Size**(*i***RowNum*) to *Size**((*i*+1)**RowNum-1*), should be obtained on the process with the rank *i*



Step 4. Parallel Program…

Task 9 – Gather the Results

- To gather the parts of the result vector located on different processes let's use the corresponding function **`MPI_Allgather()`**

```
int MPI_AllGather(void *sbuf, int scount,
  MPI_Datatype stype, void *rbuf, int rcount,
  MPI_Datatype rtype, MPI_Comm comm);
```

- **`sbuf`**, **`scount`**, **`stype`** – the parameters of the transmitted message
- **`rbuf`**, **`rcount`**, **`rtype`** – the parameters of the received message
- **`comm`** – the communicator, within of which the operation is executed

Step 4. Parallel Program…

Task 9 – Gather the Results

- Develop The function **`ResultReplication()`** that will be responsible for gathering results

```
// Result vector replication
void ResultReplication(double* pProcResult,
  double* pResult, int Size, int RowNum) {
  MPI_Allgather(pProcResult, RowNum, MPI_DOUBLE, pResult,
    RowNum, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

- After the results are gathered, add the print of the result vector to the main function by means of the function **`PrintVector()`** on all the parallel processes
- Compile and run the application. Estimate the correctness of its operation

Step 4. Parallel Program…

Task 10 – Test the Parallel Program Correctness

- To test the correctness of the program execution develop the function **`TestResult()`**. It will compare the results of the serial and parallel programs

```
// Testing the parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector,
  double* pResult, int Size)
```

- To execute the serial algorithm use the function **`SerialResultCalculation()`**. The result of this function will be stored in the vector **`pSerialResult`**
- Compare the vector **`pSerialResult`** to the vector **`pResult`**, obtained by means of the parallel program element by element
- The result of the function is the print of the diagnostic message

Step 4. Parallel Program…

Task 10 – Test the Parallel Program Correctness

- Comment on the calls of the functions, using the debugging print, which have been previously used (the functions **`TestDistribution()`**, **`TestPartialResult()`**)
- 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 4. Parallel Program…

Task 11 – Implement the Computations for Any Matrix Sizes

- Consider the case, when the matrix and vector size **`Size`** is not divisible by the number of processes **`ProcNum`**
- Determine, how many rows should each process operate
- Use the function **`MPI_Scatterv()`** to distribute the data between processes
- Implement necessary changes in the functions **`ProcessInitialization()`**, **`DataDistribution()`**, **`ResultReplication()`**
- Check the result correctness with the function **`TestResult()`**

Step 4. Parallel Program

Task 12 – Carry out the Computational Experiments

- Add time measurement of matrix-vector multiplication execution
- Carry out the computational experiments with large objects
- Fill the table with results of experiments

Summary

- One parallel method of matrix-vector multiplication is considered
- Serial and parallel methods of matrix-vector multiplication are implemented
- Computational experiments are performed, comparison of serial and parallel algorithms is made

Exercises

- Study the parallel algorithm of matrix-vector multiplication based on column-wise block-striped matrix partitioning. Develop a program implementation of this algorithm

- Study the parallel algorithm of matrix-vector multiplication based on chessboard block matrix partitioning. Develop a program implementation of this algorithm

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. Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.
4. Foster, I. (1995). Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley.