Learning Lab 4: Parallel Methods of Data Sorting

Learning Lab 4: Parallel Methods of Data Sorting 1
Lab Objective 2
Exercise 1 – State the Data Sorting Problem 2
Exercise 2 – Code the Serial Sorting Program 3
Task 1 – Open the project SerialBubbleSort 3
Task 2 – Input the Sorted Data Size 4
Task 3 –Terminate the Program Execution 5
Task 4 – Implement the Bubble Sorting Algorithm 6
Task 5 – Carry out the Computational Experiments 7
Exercise 3 – Develope the Parallel Sorting Algorithm 9
Parallelizing Principles 9
Subtask Definition and Analysis of Information Dependencies 10
Scaling and Distributing the Subtasks among the Processors 11
Exercise 4 – Code the Parallel Sorting Program 11
Task 1 – Open the Project ParallelBubbleSort 11
Task 2 – Initialize and Terminate the Parallel Program 12
Task 3 – Input the Initial Data 13
Task 4 – Terminate the Parallel Program 15
Task 5 – Distribute the Data among the Processes 15
Task 6 – Implement the Local Data Sorting 17
Task 7 – Exchange the Sorted Data 18
Task 8 – Merge and Split the Data 19
Task 9 – Implement the Parallel Odd-Even Sorting Iteration 20
Task 10 – Collect the Sorted Data 21
Task 11 – Test the Parallel Program Correctness 22
Task 12 – Implement the Sorting for Any Data Amount 23
Task 13 – Carry out the Computational Experiments 26
Discussions 27
Exercises 28
Appendix 1. The Program Code of the Serial Application of Bubble Sorting 28
File SerialBubbleSort.cpp 28
File SerialBubbleSortTest.cpp 29
Application 2. The Program Code of the Parallel Application of Bubble Sorting 30
File ParallelBubbleSort.cpp 30
File ParallelBubbleSortTest.cpp 35

Sorting is a typical problem of data processing and is usually considered as the problem of disposing an unregulated set of values in the order of the monotonic increase or decrease.

The computational complexity of the sorting methods is considerably high. Thus, for a number of well known methods (bubble sort, insertion sorting etc.) the number of the necessary operations is determined by the square dependence on the number of the data being sorted:

$$T \sim n^2 . \tag{4.1}$$

For more efficient algorithms (merge sorting, Shell sorting, quick sorting) the complexity is determined by the following formula:

$$T \sim n \log_2 n . \tag{4.2}$$

This expression gives also the lower estimation of the necessary number of operations for sorting the set of *n* values. The algorithms of smaller complexity may be obtained only for particular variants of the problem.

Data sorting speedup may be provided by means of using several (*p>1*) processors. In this case the data is distributed among the processors. In the course of calculations the data are transmitted among the processors and one part of data is compared to another one. As a rule, the resulting (sorted) data are distributed among the processors. To regulate such distribution a scheme of consecutive numeration is introduced for the processors. It is usually required that after sorting termination the data located on the processors with smaller numbers should not exceed the values on the processors with greater numbers.

Lab Objective

The objective of this lab is to develop a parallel program for data sorting. The lab assignments include:

- Exercise 1 – Stating the data sorting problem.
- Exercise 2 – Coding the serial sorting program.
- Exercise 3 – Developing the parallel sorting algorithm.
- Exercise 4 – Coding the parallel sorting program.

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

The lab students are assumed to be familiar with the related sections of the training material: Section 4 "Parallel programming with MPI", Section 6 "Principles of parallel method development" and Section 10 "Parallel methods of data sorting". Besides, Lab 1 "Parallel algorithms of matrix-vector multiplication" is assumed to have been done.

Exercise 1 – State the Data Sorting Problem

In order to do this Exercise, you should study the serial bubble sorting algorithm. While doing the Exercise you should also study the basic principles used for data sorting and develop the suggested sorting algorithm in detail.

Data sorting is one of typical problems of data processing and is usually considered to be a problem of redistributing the elements of a given sequence of values

$$S = \{a_1, a_2, ..., a_n\}$$

in order of the monotonic increase or decrease

$$S \sim S' = \{(a_1', a_2', ..., a_n') : a_1' \le a_2' \le ... \le a_n'\}$$

Under closer consideration of data sorting operations applied in sorting algorithms, it becomes evident that many methods are based on the same basic *compare-exchange* operation. This operation consists in comparing a pair of values of the data set being sorted and transposition of the values, if their values do not correspond to the sorting conditions.

```
// Basic compare-exchange operation
if(A[i] > A[j]) {
  temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

The successive application of this operation makes possible to sort the data. In many cases just approaches for choosing the pairs of this operation determine the main difference between the sorting algorithms.

Let us choose one of the widely used methods of data sorting to be implemented in this lab: the bubble sort algorithm (see Section 10 "Parallel methods of data sorting" of the training material). This algorithm compares and exchanges the neighboring elements in a sequence to be sorted. For the sequence

$$(a_1,\ a_2, ...,\ a_n)$$

the algorithm first executes $n-1$ basic compare-exchange operations for sequential pairs of elements

$$(a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n).$$

As a result, the biggest element is moved to the end of the sequence after the first algorithm iteration. Then the last element in the transformed sequence may be omitted, and the above described procedure is applied to the remaining part of the sequence

$$(a'_1, a'_2, ..., a'_{n-1}).$$

As it can be seen, the sequence will be sorted out after $n-1$ iterations. The bubble sorting efficiency may be improved, if the algorithm is terminated when there are no changes of the data sequence being sorted in the course of some successive sorting iteration.

```
// Sequential bubble sorting algorithm
BubbleSort(double A[], int n) {
  for(i = 1; i < n; i++)
    for(j = 0; j < n - i; j++)
      compare_exchange(A[j], A[j+1]);
}
```

Exercise 2 – Code the Serial Sorting Program

In order to do the Exercise you should implement the serial algorithm of bubble sorting. The initial variant of the program to be developed is given in the project *SerialBubbleSort*, which contains a certain part of the initial code. While doing this Exercise it is necessary to add the input operations of initial data size, data initialization for sorting, the operations of data sorting and the operations of checking the program functioning to the given variant of the program.

Task 1 – Open the project SerialBubbleSort

Open the project *SerialBubbleSort* 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,**
- Make the double click on the file **SerialBubbleSort.sln** or execute the command **Open** after selecting the file**.**

After the project has been open in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *SerialBubbleSort.cpp*, as it is shown in Figure 4.1. After that, the code, which is to be enhanced, will be opened in the Visual Studio workspace.

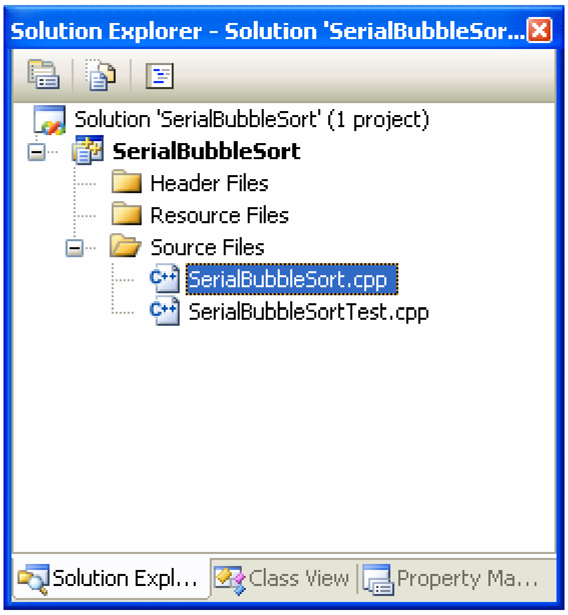


Figure 4.1. Opening the File SerialBubbleSort.cpp

The file *SerialBubbleSort.cpp* provides access to the necessary libraries and also contains the initial version of the head program function – the function *main*. It provides the possibility to declare the variables and to print out the initial program message. The file *SerialBubbleSortTest.cpp* contains the developed test functions, which will be necessary for checking the correctness of the functions to be developed.

Let us consider the variables, which are used in the main function of the program. The variable *pData* is the data to be sorted. After the sorting is completed this variable will point to the sorted data set. The variable *DataSize* defines the amount of data to be sorted.

```
  double *pData;    // Data to be sorted
  int DataSize;     // Size of data to be sorted
```

The output of the initial message is provided by means of the following program lines:

```
  printf("Serial bubble sort program\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 bubble sort program`".

Task 2 – Input the Sorted Data Size

In order to input the initial data, we will develop the function *ProcessInitialization*. This function is aimed at initializing all the variables, which are used in the program, in particularly, the ones used to input the amount of the sorted data, memory allocating for the sorted data and setting the data with the initial unsorted values. Thus, the function should have the following heading:

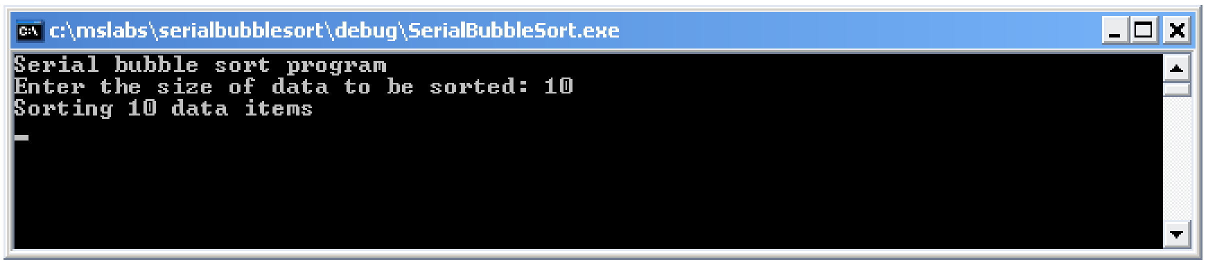
```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize);
```

Let us add the code, which allows to input the amount of the sorted data (to set the value of the variable *DataSize*) and check the correctness of the executed input. For this purpose add the code bold typed below to the function *ProcessInitialization* and add the call of the function to the main function of the program:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize) {
  do {
    printf("Enter the size of data to be sorted: ");
    scanf("%d", &DataSize);
    if(DataSize <= 0)
      printf("Data size should be greater than zero\n");
  }
  while(DataSize <= 0);

  printf("Sorting %d data items\n", DataSize);
}
```

The user can input the size of data to be sorted, which is read from the standard input flow *stdin* and stored in the integer variable *DataSize*. After that the value of the variable *DataSize* is checked (it must be greater than zero), in case of the invalid input the latter is repeated and, finally, the entered value is printed (see Figure 4.2).

Compile and run the application. Make sure that in case if a negative value of the variable *DataSize* is entered, the program produces the diagnostic message: "`Data size should be greater than zero`".



```
c:\mslabs\serialbubblesort\debug\SerialBubbleSort.exe
Serial bubble sort program
Enter the size of data to be sorted: 10
Sorting 10 data items
_
```

Figure 4.2. Setting the Sorted Data Size

Then it is necessary to enhance the function *ProcessInitialization* by memory allocation for the data to be sorted and initialize the data. For this purpose you should add the bold marked code to the function *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization (double *&pData, int& DataSize) {
  <…>
  printf("Sorting %d data items\n", DataSize);

  pData = new double[DataSize];
  DummyDataInitialization(pData, DataSize);
}
```

Data initialization will be executed by a special function. At the first stage it is possible to use a simple variant of implementation, which creates the data set, where the correctness of sorting can be easily checked. There is a version for the function (*DummyDataInitialization*) in the code. It is only necessary to enhance it with the following code:

```
// Function for simple setting the initial data
void DummyDataInitialization(double*& pData, int& DataSize) {
  for(int i = 0; i < DataSize; i++)
    pData[i] = DataSize - i;
}
```

The given code shows that the function fills the data with consecutive numbers beginning with the size of data and up to one. Thus, if the user has entered the amount of data to be sorted equal to 10, the data will be defined the following way:

$$(10,9,8,7,6,5,4,3,2,1)$$

(initializing the sorted data by means of a random number generator will be considered in Exercise 5).

It should be noted that the lab describes data sorting in the order of increase so that the given function fills the array with the values, which have been sorted in the reverse order.

Compile the application (execute the command **Rebuild Solution** of the menu option **Build**). In case if there are errors, check them comparing your code to the code given in this Exercise. After the errors have been corrected start the application.

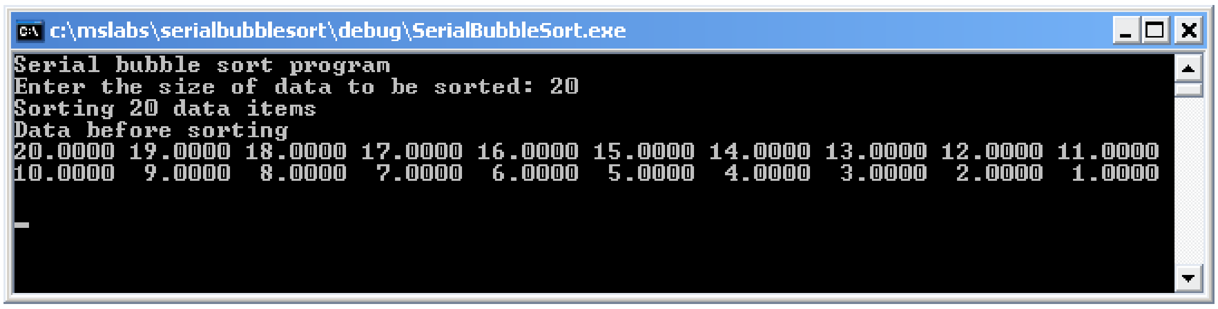
Let us develop one more function, which will further help to control the program execution. This is the function of the formatted data output *PrintData*. The draft version of the function may be found in the file *SerialBubbleSortTest.cpp*. It is possible to pass over to editing of the file on the analogy with choosing the file *SerialBubbleSort.cpp* for editing in Exercise 1. The array *pData,* where the data is stored and the number of the elements in the array *(Data Size),* are given as arguments to the function of the formatted data print *PrintData*:

```
// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
  printf("\n");
}
```

Let us add the call of the function *PrintData* to the function *main* of the application immediately after the call of the function *ProcessInitialization* in order to check the correctness of the initial data setting:

```
...
  // Process initialization
  ProcessInitialization(pData, DataSize);

  printf("Data before sorting\n");
  PrintData(pData, DataSize);
...
```

Compile and run the application. Make sure that the data input is executed according to the above-described rules (Figure 4.3). Run the application several times, using various data sizes.



```
c:\mslabs\serialbubblesort\debug\SerialBubbleSort.exe
Serial bubble sort program
Enter the size of data to be sorted: 20
Sorting 20 data items
Data before sorting
20.0000 19.0000 18.0000 17.0000 16.0000 15.0000 14.0000 13.0000 12.0000 11.0000
10.0000  9.0000  8.0000  7.0000  6.0000  5.0000  4.0000  3.0000  2.0000  1.0000
```

Figure 4.3. The Result of the Program Execution after Completion of Exercise 2

Task 3 –Terminate the Program Execution

In this Task before working out the code of data sorting we should develop the function for the correct computational process termination. For this purpose we will deallocate the memory allocated dynamically in the course of the program execution. We will develop the corresponding function *ProcessTermination*. The memory was allocated for storing the data to be sorted *pData*. Consequently, the pointer to the memory must be given to the function *ProcessTermination* as argument:

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

The call of the function *ProcessTermination* should be added to the program immediately before the termination of the main function *main*, when the sorted data is not needed anymore:

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

  return 0;
}
```

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

Task 4 – Implement the Bubble Sorting Algorithm

Let us develop the main computational part of the program now. To execute bubble sorting we will develop the function *SerialBubble*, which uses the initial data *pData* and the size of the data *DataSize* as input parameters.

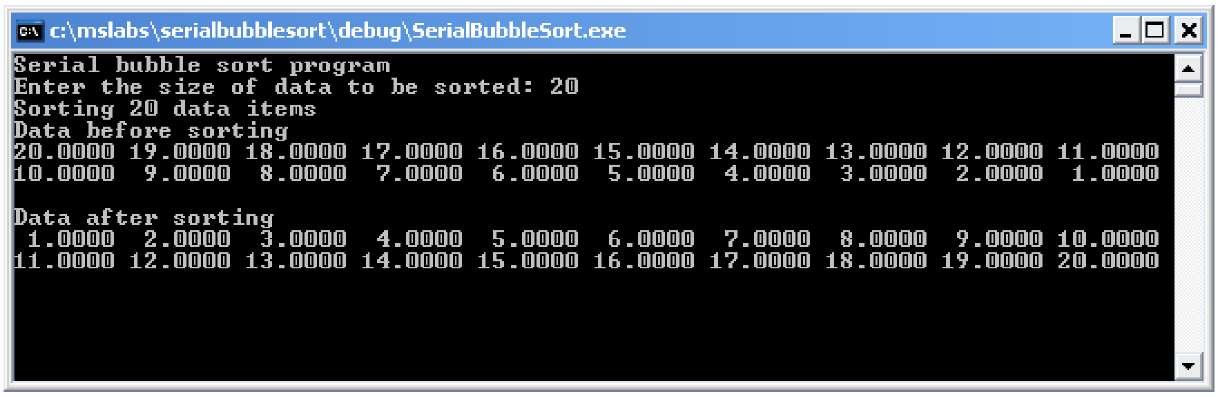
In accordance with the algorithm given in Exercise 1, the code of the function should be as follows:

```
// Function for the serial bubble sort algorithm
void SerialBubble(double *pData, int DataSize) {
  double Tmp;

  for(int i = 1; i < DataSize; i++)
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}
```

Let us call the function of sorting from the main program. In order to check the correctness of sorting we will print the obtained data:

```
..<…>

  printf("Data before sorting\n");
  PrintData(pData, DataSize);

  // Serial bubble sort
  SerialBubble(pData, DataSize);

  printf("Data after sorting\n");
  PrintData(pData, DataSize);
...
```

Compile and run the application. Analyze the result of the sorting algorithm operation. If the algorithm has been implemented correctly the result must present all the number from one and up to the size of the entered data in series. Carry out several computational experiments varying the size of the data set to be sorted.



```
Serial bubble sort program
Enter the size of data to be sorted: 20
Sorting 20 data items
Data before sorting
20.0000 19.0000 18.0000 17.0000 16.0000 15.0000 14.0000 13.0000 12.0000 11.0000
10.0000  9.0000  8.0000  7.0000  6.0000  5.0000  4.0000  3.0000  2.0000  1.0000

Data after sorting
 1.0000  2.0000  3.0000  4.0000  5.0000  6.0000  7.0000  8.0000  9.0000 10.0000
11.0000 12.0000 13.0000 14.0000 15.0000 16.0000 17.0000 18.0000 19.0000 20.0000
```

Figure 4.4. The Result of Sorting the Array of 20 Elements

Task 5 – Carry out the Computational Experiments

In order to test the speed up of the parallel program execution, it is necessary to carry out experiment on calculating the execution time for the serial program. It is reasonable to analyze the algorithm execution time for significantly large data sizes. The data will be given by means of a random data generator. For this purpose let us develop one more function for setting elements *RandomDataInitialization* (the random number generator is initialized by the clock):

```
// Function for initializing the data by the random generator
void RandomDataInitialization(double *&pData, int& DataSize) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}
```

Let us call this function instead of the previously developed function *DummyDataInitialization* in the function *ProcessInitialization*, which generated the predictable data, the use of which made possible to check the algorithm functioning easily:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize) {

  <…>
  pData = new double[DataSize];
  // Setting the data by the random generator
  RandomDataInitialization(pData, DataSize);
}
```

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 to find out the program execution time or the execution time of its part, to the obtained program. As previously, use the following function for this purpose:

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

In order to calculate the execution time, we will need three auxiliary variables, which must be additionally declared in the function *main*:

```
int main(int argc, char *argv[]) {
  double *pData = 0;
  int DataSize = 0;

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

Let us add the computation and the output of the sorting execution time to the program code. For this purpose we will clock in before and after the call of the function *SerialBubble*:

```
...
```

```
  // Serial bubble sort
  start = clock();
  SerialBubble(pData, DataSize);
  finish = clock();

  printf("Data after sorting\n");
  PrintData(pData, DataSize);

  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 with large sorted data set, switch off print before and after sorting (the corresponding code lines transform to the comment). Carry out the computational experiments and write the results in the third column of the table:

Table 4.1. The Results of the Computational Experiments of the Bubble Sorting Method

Test number	The Size of the Data to Be Sorted	The Execution Time of Bubble Sorting (sec.)	The Execution Time of Sorting from the Standard Library (sec.)
1	10		
2	100		
3	10,000		
4	20,000		
5	30,000		
6	40,000		
7	50,000		

Estimate the nuture of the dependence of the sorting time on the size of data to be sorted using the experimental results. Make sure that the dependence is square (if the amount of the data to be sorted is increased two times, the sorting time increases four times etc.).

As it has already been noted there are more efficient sorting methods (in comparison to the bubble sorting algorithm). One of these quick algorithms is implemented in the standard library of the algorithmic language C++. Let us carry out the experiments in order to estimate the efficiency of this standard algorithm. For this purpose we should use the call of the function *sort* from the standard library (you should add the header file <*algorithm*> to the program) instead of the call of the bubble sorting function. Let us make use of the function *SerialStdSort*, which is located in the file *SerialBubbleSortTest.cpp*:

```
// Function for sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}
```

Transform the call *SerialBubble* into comment and add the call of the described function to the function *main*:

```
  start = clock();
  // Serial buble sort
  // SerialBubble(pData, DataSize);
  // Sorting by the standard library algorithm
  SerialStdSort(pData, DataSize);
  finish = clock();
```

Carry out the computational experiments on the analogy of the previous one and write the results in the last column of the table.

Let us estimate the computational comlexity of the bubble-sorting algorithm analytically (see Section 10 "Parallel Methods of Data Sorting" of the training material). Obtaining the sorted data implies the execution of *DataSize* – 1 identical operations of the sorting pass. Each sorting pass includes *DataSize* – *i* of basic operations "*compare-exchange*", where *i* is the number of the sorting pass. Thus, the algorithm execution time can be estimated as follows:

$$T_1 = (DataSize \cdot (DataSize - 1)/2) \cdot \tau \,, \qquad (4.3)$$

where τ is the execution time of the "compare-exchange" operation.

Let us show the comparison of the experiment execution time to the time, which may be obtained with the use of the formula (4.3) in the form of a table. In order to calculate the execution time of the "compare-exchange" operation, we will use the following method: let us choose one of the experiments as the pivot (for instance, the experiment on sorting 30,000 data elements) and divide the execution time of the experiment by the number of the executed operations. Thus, we will compute the execution time τ. After that we will use this value to compute the theoretical execution time for the rest of the experiments.

Write the results in the table below:

Table 4.2. The Comparison of the Experimental and the Theoretical Time for the Bubble Sorting Method

The Execution Time of the "Compare-Exchange" Operation τ (sec):			
Test Number	The Size of Data to Be Sorted	Execution Time (sec)	Theoretical Time (sec)
1	10		
2	100		
3	10,000		
4	20,000		
5	30,000		
6	40,000		
7	50,000		

It should be noted that the execution time of the "compare-exchange" operation depends on the amount of data to be sorted. This dependence can be explained by the properties of the computer architecture. If the amount of data is not large, the data may be located in the cache memory of the processor, and the access to the memory is very fast. If the amount of data in the algorithm makes possible to locate the data in RAM but not in the cache memory, then the operation execution time will be somewhat greater, as it requires more time to access to the RAM element than to access to cache. If there is so much data that it cannot be located in RAM, then the mechanism of swap file operation is activated. The data is stored on the external storage, and the read and write time for the external storage exceeds significantly the time of recording to the RAM location. Thus, when you select the pivot experiment (the experiment, for which you should calculate the execution time for the "compare-exchange" operation), you should keep within the range of an average situation. That is why we have chosen the experiment of sorting 30,000 data elements as the pivot one.

Exercise 3 – Develope the Parallel Sorting Algorithm

It order to do the Exercise you should study the principles of parallelizing bubble sorting algorithm. For this purpose you should decompose the problem, select the information interactions among the subtasks and distribute subtasks among the processors.

Parallelizing Principles

Bubble sorting algorithm is rather complicated for parallelizing. The comparison of the value pairs of the data set being sorted is strictly sequential. As a result the modification of the algorithm, which is known as odd-even transposition, is used in parallel calculation, - see, for instance, Section 10 "Parallel Methods of Data Sorting" of the training material. The main idea of modification may be described as follows: two different rules of executing the method iterations are introduced into the sorting algorithm. The elements with odd or even indices correspondingly are chosen for processing depending on the even or odd number of the sorting iteration. The selected values are compared to their right neighboring elements. Thus, at all odd iterations the following pairs are compared:

$(a_1, a_2), (a_3, a_4), ..., (a_{n-1}, a_n)$ (if n is even),

at even iterations the following elements are processed

$(a_2, a_3), (a_4, a_5), ..., (a_{n-2}, a_{n-1})$.

After n execution of sorting iterations the initial data set appears to be ordered.

```
// Sequential odd-even transposition algorithm
OddEvenSort ( double A[], int n ) {
  for ( i=1; i<n; i++ ) {
    if ( i%2==1 ) { // odd iteration
      for ( j=0; j<n/2-2; j++ )
```

```
        compare_exchange(A[2j+1],A[2j+2]);
      if ( n%2==1 ) // the comparison of the last pair, if n is odd
        compare_exchange(A[n-2],A[n-1]);
     }
     if ( i%2==0 )    // even iteration
       for ( j=1; j<n/2-1; j++ )
         compare_exchange(A[2j],A[2j+1]);
  }
}
```

More detailed information on the odd-even transposition algorithm is given in Section 10 "Parallel Methods of Data Sorting" of the training material.

Subtask Definition and Analysis of Information Dependencies

Parallelizing the odd-even transposition method does not cause any problems. The pairs of values at sorting iterations may be compared independently and in parallel. In case when $p<n$, i.e. the number of processors is less than the number of the values being sorted, the processors contain the data blocks of n/p size. The sorting algorithm in this case may be obtained as the generalization of the odd-even sorting procedure (see Subsection 10.2 of Section 10 "Parallel Methods of Data Sorting" of the training material).

Following the scheme of one-element comparison, the interaction of the processor pair P_i and P_{i+1} for conjoint sorting the blocks A_i and A_{i+1} may be execute the following way:

- Execute the exchange of blocks between the processors P_i and P_{i+1},
- Combine blocks A_i and A_{i+1} on each processor and create a block of sorted data of double size (if blocks A_i and A_{i+1} have been initially sorted, the procedure of uniting the blocks is reduced to the quick operation of merging two sets of sorted data),
- Divide the obtained double block into two equal parts and leave one of these parts (for instance, the one with smaller data values) on the processor P_i, and the other one (with greater data values, correspondingly) on the processor P_{i+1}

$$[A_i \cup A_{i+1}]_{sort} = A_i^{'} \cup A_{i+1}^{'} : \forall a_i^{'} \in A_i^{'}, \forall a_j^{'} \in A_{i+1}^{'} \Rightarrow a_i^{'} \le a_j^{'} .$$

The above described "compare- split" operation may be used as the basic computational subtask for parallel computation. The parallel algorithm obtained as a result may be presented the following way:

```
// Parallel odd-even transposition algorithm
ParallelOddEvenSort(double A[], int n) {
  int id = GetProcId();  // Process number
  int np = GetProcNum(); // Number of processes
  for ( int i=0; i<np; i++ ) {
    if ( i%2 == 1 ) { // Odd iteration
      if ( id%2 == 1 ) { // Odd process number
        // Compare-exchange with the right neighbor process
        if ( id < np -1 ) compare_split_min(id+1);
      }
      else
        // Compare-exchange with the left neighbor process
        if ( id > 0 ) compare_ split_max(id-1);
    }
    if ( i%2 == 0 ) { // Even iteration
      if( id%2 == 0 ) { // Even process number
        // Compare-exchange with the right neighbor process
        if ( id < np -1 ) compare_ split_min(id+1);
      }
      else
        // Compare-exchange with the left neighbor process
        compare_ split_max(id-1);
    }
  }
}
```

In order to illustrate this parallel method of sorting we give the following example of data sorting in case when $n=16, p=4$ (i.e. the block of values on each processor contains $n/p=4$ elements) in Table 4.3. In the first column of the table there is the number and the type of the method iteration and the pairs of processes, for which the "compare- split" operations are executed in parallel, are enumerated. The interacting pairs of processes are marked with the double-lined frame in the table. The state of the data set being sorted before and after the iteration is shown for each step of iteration.

Table 4.3. The Example of Data Sorting by the Parallel Method of the Odd-Even Transposition

Iteration Number and Type	Processors			
	1	2	3	4
Initial Data	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
1 odd (1,2),(3.4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
2 even (2,3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
3 odd (1,2),(3.4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
4 even (2,3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

Scaling and Distributing the Subtasks among the Processors

As it has been stated previously, the number of subtasks corresponds to the number of the available processors, and, as a result, there is no need scaling the computations. The initial distribution of the blocks of the data set to be sorted among the processors may be carried out arbitrarily. For efficient execution of the above described parallel sorting algorithm it is essential that the processors with the neighboring numbers should have direct communication links.

Exercise 4 – Code the Parallel Sorting Program

To do this Exercise you should develop the parallel sorting program. This Exercise is aimed at:

- Enhancing the practical knowledge on parallel program development,
- Developing the parallel program for data sorting.

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,
- Termination of MPI program.

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

Task 1 – Open the Project ParallelBubbleSort

Open the project *ParallelBubbleSort* using the following steps:

- Start the application **Microsoft Visual Studio 2005**, if it has not been started yet,
- Execute the command **Open→Project** in the menu **File,**
- Choose the folder **c:\MsLabs\ ParallelBubbleSort** in the dialog window **Open Project,**
- Make the double click on the file **ParallelBubbleSort.sln** or select it and execute the command **Open.**

After the project has been open in the window Solution Explorer (Ctrl+Alt+L), make the double click on the file of the initial code *ParallelBubbleSort.cpp*, as it is shown in Figure 4.5. After that, the code, which is to be enhanced, will be opened in the Visual Studio workspace.

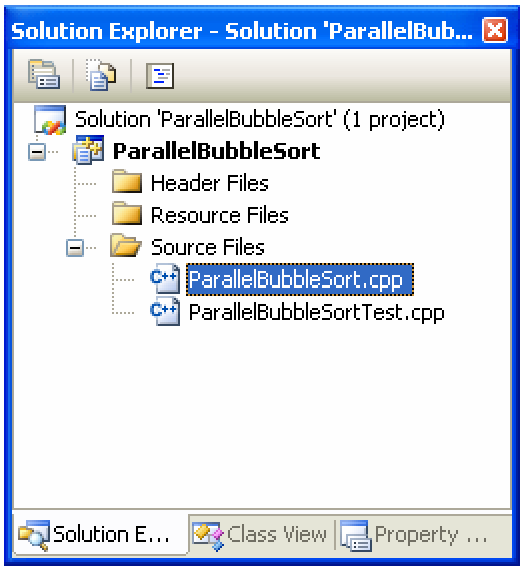


Figure 4.5. Opening File ParallelBubbleSort.cpp

The main function of the parallel program to be developed, which contains the declaration of the necessary variables, is located in the file *ParallelBubbleSort.cpp***.** The following functions copied from the serial project are also located in the file *ParallelBubbleSort.cpp***:** *DummyDataInitialization*, *RandomDataInitialization*. Besides, you can also see the function *PrintData* responsible for test data output in the file *ParallelBubbleSortTest.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 draft 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 bubble sort program"
```

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 define the number of processes available for MPI program, to define the process rank in communicator MPI_COMM_WORLD, and also set global variables for storing these values (*ProcNum* and *ProcRank* correspondingly). Add the following marked code:

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

void main(int argc, char* argv[]) {
  double *pData = 0;
  double *pProcData = 0;
  int DataSize = 0;
  int BlockSize = 0;

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

  printf("Parallel bubble sort program\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 execute the command **Run**,
2. Type the name of the program **cmd** in the dialog window, which appears on the screen (Figure 4.6)

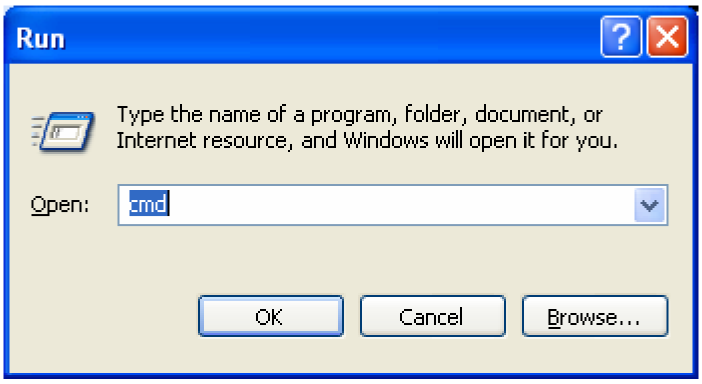
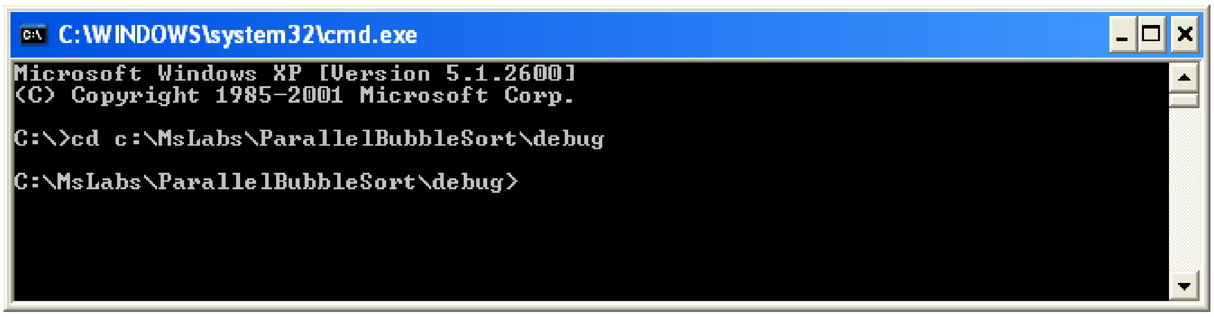

Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
cmd
OK
Cancel
Browse...

Figure 4.6. The Start of Command Prompt

In the command line go to the folder, which contains the developed program (Figure 4.7):

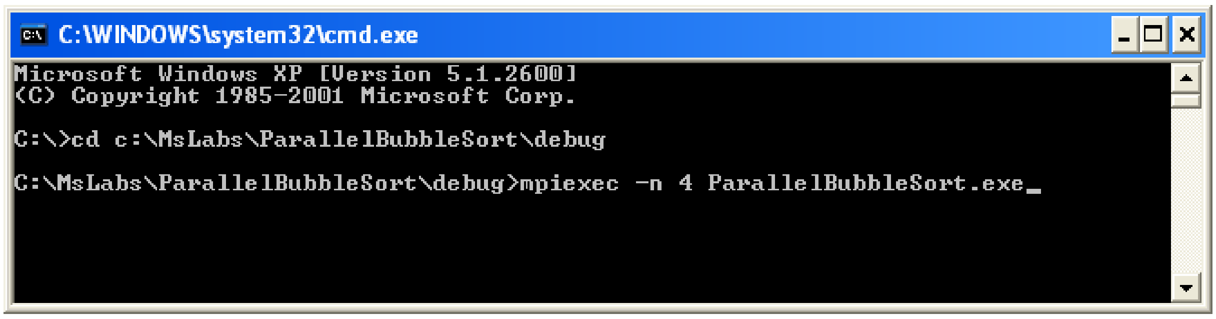


```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

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

C:\MsLabs\ParallelBubbleSort\debug>
```

Figure 4.7. Setting the Folder with Parallel Program

Then type the command (Figure 4.8) in order to execute the program using 4 processes:

```
mpiexec –n 4 ParallelBubbleSort.exe
```



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

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

C:\MsLabs\ParallelBubbleSort\debug>mpiexec -n 4 ParallelBubbleSort.exe_
```

Figure 4.8. Starting the Parallel Program

Make sure that initial message

```
"Parallel bubble sort program"
```

is output to the command console.

Task 3 – Input the Initial Data

At the following stage of the parallel application development, it is necessary to set the initial data. We should enhance our program with the code, which provides setting the amount of the data to be sorted and allocates memory for the sorted elements. 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 data to be sorted is distributed among all the processes so that each of them will operate with a part of sorted data (*block*). It should be noted that the first version of the program to be developed is implemented for the case when the number of sorted data is divisible by the number of processes without remainder, i.e. the data blocks on all the processes contain the same number of elements. This number of block elements will be stored in the variable *BlockSize*. The address of the memory buffer, where the data block is located on each of the processes, will be stored in the variable *pProcData*. As a result of sorting, each process obtains *BlockSize* of sorted elements of the result data. Then the data should be collected on the root process again (the process with the rank 0).

In order to implement all the activities described above, we will develop the function *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize,
  double *&pProcData, int& BlockSize);
```

Thus, first we should set the amount of the data to be sorted, i.e. set the value of the variable *DataSize*. In order to get the sorted data size 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 marked code to the function *ProcessInitialization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize,
  double *&pProcData, int& BlockSize) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    do {
      printf("Enter the size of data to be sorted: ");
      scanf("%d", &DataSize);
      if(DataSize < ProcNum) {
        printf("Data size should be greater than number of processes\n");
      }

      if(DataSize % ProcNum != 0) {
        printf("Data size should be divisible by number of processes\n");
      }
    } while((DataSize < ProcNum) || (DataSize % ProcNum != 0));

    printf("Sorting %d data items\n", DataSize);
  }
}
```

Now it is necessary to broadcast the size of data to be sorted 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(double *&pData, int& DataSize,
  setvbuf(stdout, 0, _IONBF, 0);
  double *&pProcData, int& BlockSize) {
  if(ProcRank == 0) {
    ...
    printf("Sorting %d data items\n", DataSize);
  }
  // Broadcasting the data size
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
  BlockSize = DataSize / ProcNum;
}
```

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

```
int main(int argc, char *argv[]) {
  <…>

  if(ProcRank == 0)
    printf("Parallel bubble sort program\n");

  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);
  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 various number of parallel processes (by means of the utility **mpiexec**) and various sizes of the data to be sorted.

After the amount of the data to be sorted has been obtained, it is possible to allocate the memory for these data and blocks assigned to the processes. Add the marked code to the function *ProcessInitialization*:

```
  // Broadcasting the data size
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
  BlockSize = DataSize / ProcNum;

  pProcData = new double[BlockSize];

  if(ProcRank == 0) {
    pData = new double[DataSize];

    // Data initalization
    DummyDataInitialization(pData, DataSize);
  }
}
```

It should be noted that in order to set the elements to be sorted on the root process, we have used the function of data generation *DummyDataInitialization*, which was developed for the serial sorting program. Pay attention to the fact that the function fills the array with the values, which have been sorted in the order of decrease.

Task 4 – Terminate the Parallel Program

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 all the sorted data *pData* was allocated on the root process; besides, memory was allocated on all the processes for storing the data block *pProcData*. This two variables must be given to the function *ProcessTermination* as arguments:

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

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

```
...
  // Process termination
  ProcessTermination(pData, pProcData);

  MPI_Finalize();
}
```

The commands for printing the sorted data on the root process should be added to the code of the main function (use the function *PrintData*, which was implemented in the course of serial program 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 data to be sorted must be distributed among the processes in equal-sized blocks.

The function *DataDistribution* is responsible for data distribution. The sorted data *pData*, the data size *DataSize* and the process block *pProcData*, and also the size of the block *BlockSize* must be given to the function as arguments:

```
// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData,
  int BlockSize);
```

To implement the fuction it is necessary to use the generalized operation of data transmission from one process to all processes (data distribution) and to distribute the data to be sorted among the processes:

```
// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData,
  int BlockSize) {

  MPI_Scatter(pData, BlockSize, MPI_DOUBLE, pProcData, BlockSize,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

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

```
  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  // Distributing the initial data among processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);
```

Now let us check the correctness of the data distribution among the processes up. For this purpose we should print the data to be sorted and the data blocks located on each of the processes after the execution of the function *DataDistribution.* Let us add to the program one more function, which helps to check the correctness of the data distribution stage. This function will be referred to as *TestDistribution*.

In order to provide the formatted data input we will make use of the previously developed function *PrintData*:

```
// Function for testing the data distribution
void TestDistribution(double *pData, int DataSize, double *pProcData,
  int BlockSize) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial data:\n");
    PrintData(pData, DataSize);
  }

  MPI_Barrier(MPI_COMM_WORLD);

  for (int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Block:\n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

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

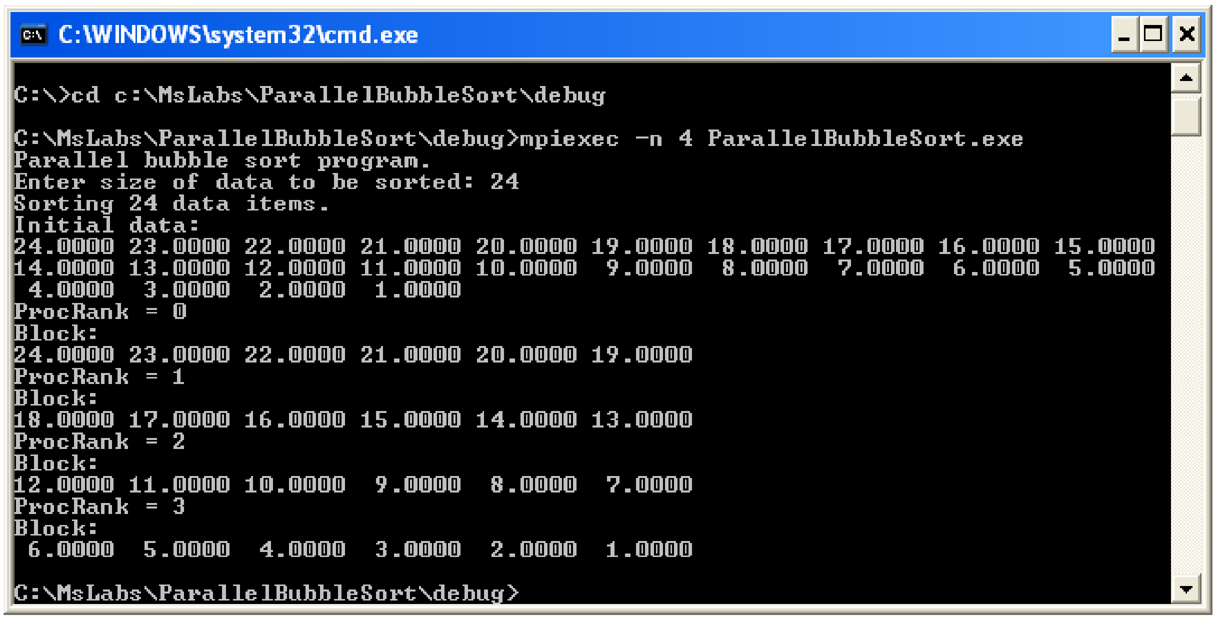
Add the call of the test distribution function immediately after the function *DataDistribution*:

```
...
  // Distributing the initial data among the processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);
  // Testing the data distribution
  TestDistribution(pData, DataSize, pProcData, BlockSize);
```

```
...
```

It should be noted that the function of the initial data generation *DummyDataInitialization* is implemented so that it generates the initial data sorted in the order of decrease. It means that after data partitioning the data in the interval from *DataSize–i*BlockSize* to *DataSize*–(*i*+1)**BlockSize*+1 must appear on the process with the rank *i*.

Compile the application. In case any errors are identified, correct them comparing your code to the code given here. Start the application, which uses four processes and set the data size equal to 24. Make sure that data distribution is performed correctly (Figure 4.9).



```
C:\>cd c:\MsLabs\ParallelBubbleSort\debug

C:\MsLabs\ParallelBubbleSort\debug>mpiexec -n 4 ParallelBubbleSort.exe
Parallel bubble sort program.
Enter size of data to be sorted: 24
Sorting 24 data items.
Initial data:
24.0000 23.0000 22.0000 21.0000 20.0000 19.0000 18.0000 17.0000 16.0000 15.0000
14.0000 13.0000 12.0000 11.0000 10.0000  9.0000  8.0000  7.0000  6.0000  5.0000
 4.0000  3.0000  2.0000  1.0000
ProcRank = 0
Block:
24.0000 23.0000 22.0000 21.0000 20.0000 19.0000
ProcRank = 1
Block:
18.0000 17.0000 16.0000 15.0000 14.0000 13.0000
ProcRank = 2
Block:
12.0000 11.0000 10.0000  9.0000  8.0000  7.0000
ProcRank = 3
Block:
 6.0000  5.0000  4.0000  3.0000  2.0000  1.0000

C:\MsLabs\ParallelBubbleSort\debug>
```

Figure 4.9. Data Distribution for the Program Using Four Processes

Task 6 – Implement the Local Data Sorting

Let us implement the algorithm of the parallel odd-even sorting 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 parallel data sorting function. It is necessary to have the process block *pProcData*, where the data is located and the block size *BlockSize* in order to sort the data block, which belongs to a process. As a result, the developed sorting function will have the following heading:

```
// Parallel bubble sort algorithm
void ParallelBubble(double *pProcData, int BlockSize);
```

In accordance with the general scheme of the parallel odd-even sorting it is first of all necessary to sort the process block data. For this purpose we may use the function *SerialBubbleSort* (which may be found in the file *ParallelBubbleSortTest.cpp*). This function was developed in Exercise 2. In order to execute the local sorting you should carry out the first implementation of the function *ParallelBubble*:

```
// Parallel bubble sort algorithm
void ParallelBubble(double *pProcData, int BlockSize) {

  // Local sorting the process data
  SerialBubbleSort(pProcData, BlockSize);

  // Print the sorted data
  ParallelPrintData(pProcData, BlockSize);
}
```

As you may notice, the call of the function printing blocks of all the processes has been added to check the correctness of local sorting execution. A possible version of the debugging print function *ParallelPrintData* consists in the following:

```
// Function for parallel data output
void ParallelPrintData(double *pProcData, int BlockSize) {
  // Print the sorted data
  for(int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Proc sorted data:\n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

Add the call of the function *ParallelBubble* to the function *main*. Besides, in order to decrease the amount of the debugging output, comment on the call of the function *TestDistribution*. Compile and run the application. Make sure that the local data is being sorted correctly.

Task 7 – Exchange the Sorted Data

The next stage in the development of the parallel bubble-sorting algorithm is the pairwise exchange of the copies of the sorted process data. Let us develop the function *ExchangeData*, the heading of which is given below:

```
// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData);
```

This function exchanges the process blocks with the process whose rank is *DualRank*. The sent block is *pProcData* and the received block data is recorded in the array *pDualData*. The data block exchange will be carried out with the help of the function *MPI_Sendrecv,* which is familiar to those who have done the previous labs.

Add the function *ExchangeData* to the source code of the parallel application, which is being developed:

```
// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData) {

  MPI_Status status;
  MPI_Sendrecv(pProcData, BlockSize, MPI_DOUBLE, DualRank, 0,
    pDualData, BlockSize, MPI_DOUBLE, DualRank, 0,
    MPI_COMM_WORLD, &status);
}
```

Now it is necessary to call the developed function from the function *ParallelBubble*. In order to perform the call, it is necessary to allocate memory for the data to be received. In accordance with the general scheme of odd-even sorting the exchange will be carried out with the neighboring in number process to the left or to the right of the current one depending on the iteration number and the process rank. To indicate the neighboring process for carrying out the exchange, we will introduce the variable *Offset*, which will accept the values either +1 (if it is necessary to exchange blocks with the following process) or alternatively –1.

Add the following code to the function *ParallelBubble* after the call of *SerialBubbleSort*:

```
...
  double *pDualData = new double[BlockSize];

  int Offset;

  if(ProcRank != 0) {
    Offset = -1;
    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData);
  }

  delete []pDualData;
...
```

As it can be noted, the exchange in this simplified variant is carried out with the process to the left of the current one (if there is any). It should be noted that if an unavailable process rank has been pointed to the function *MPI_Sendrecv* does not perform any action and terminates its operation.

In order to check the correctness of the exchange procedure move the call of the function *ParallelPrintData* to the line, which follows the call of the function *ExchangeData*, and print the data obtained by the process. Compile and run the application. Make sure that the processes exchange the data blocks correctly. Test the exchange with the process to the right.

Task 8 – Merge and Split the Data

In accordance with the algorithm of odd-even sorting the processes should combine the available data (the initial block *pProcData* and the new data set *pDualData* obtained from the neighboring process) into one double-sized block.

This operation may be reduced to the procedure of merging as both data sets to be combined remain sorted. To merge the data you may use the function *merge* of the standard library of the algorithmic language C++. In order to store the merged data it is necessary to allocate memory for the combined block *pMergedData* and return this memory to the system, when it is not necessary any longer. Add the following lines to the function *ParallelBubble*:

```
...
  double *pDualData   = new double[BlockSize];
  double *pMergedData = new double[2 * BlockSize];
...
    // Data merging
    merge(pProcData, pProcData + BlockSize, pDualData,
      pDualData + BlockSize, pMergedData);
...
  delete []pDualData;
  delete []pMergedData;
...
```

As it can be seen, in accordance with the rules of using the function *merge* both arrays being merged have to be given by two parameters: by the pointer to the first element and by the pointer to the element, which is the first one after the last element. However the merged array is only given by the pointer to the first element.

In order to check the correctness of the merging procedure, you should move the call of the function *ParallelPrintData* to the line, which follows the call of the function *merge*, and choose for printing the combined data set. Compile and run the application. Make sure that data merging is carried out correctly.

Then each process should keep only a part of the combined data set. Let us declare the variable *SplitMode* for this purpose, which will show, which part of the combined array a process should keep. Let us define the enumerated type for setting possible values of the variable *SplitMode*:

```
enum split_mode { KeepFirstHalf, KeepSecondHalf };
```

If *SplitMode* is equal to *KeepFirstHalf*, the process should keep the first half of the data, and if *SplitMode* is equal to *KeepSecondHalf* it should keep the second half. The data to be kept should be copied to the memory according to the pointer *pProcData*.

Add the following bold marked changes to the code of the function *ParallelBubble*:

```
...
  int Offset;
  split_mode SplitMode = KeepFirstHalf;

  if(ProcRank != 0) {
    Offset = -1;
    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData);

    // Data merging
    merge(pProcData, pProcData + BlockSize, pDualData,
      pDualData + BlockSize, pMergedData);

    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData + BlockSize, pMergedData + 2*BlockSize, pProcData);
  }
```

```
...
```

As you can see, the indicator, showing which part of the result array the process must keep, is defined in the test mode immediately in the source code. The correct method for setting the value of the variable *SplitMode* will be added later in the course of the development of the final *ParallelBubble*function version.

Move the call of the function *ParallelPrintData* to the end of the function *ParallelBubble*, compile and run the application. Make sure that the processes perform data merging and splitting correctly. Test the case, when the processes keep the second half of the combined array.

Task 9 – Implement the Parallel Odd-Even Sorting Iteration

In accordance with the general scheme of odd-even sorting, the operations of block exchange, data merging and splitting implemented in Tasks 7-8, must be repeated *p* times. It should be noted that the exchange must be performed with the neighboring in number process to the right or to the left of the current one at each iteration (to indicate the necessary neighboring process we make use of the variable *Offset*). At odd iterations odd number processes must exchange with the neighbors to the right, and even number processes must exchange with the neighbors to the left. At even iterations everything happens vice versa (odd number processes exchange with the neighbors to the left, and even number processes – with the neighbors to the right). The part of the combined data kept on the processes is determined on the analogy.

The above described rules may be implemented the following way (add the necessary overpatching to the function *ParallelBubble):*

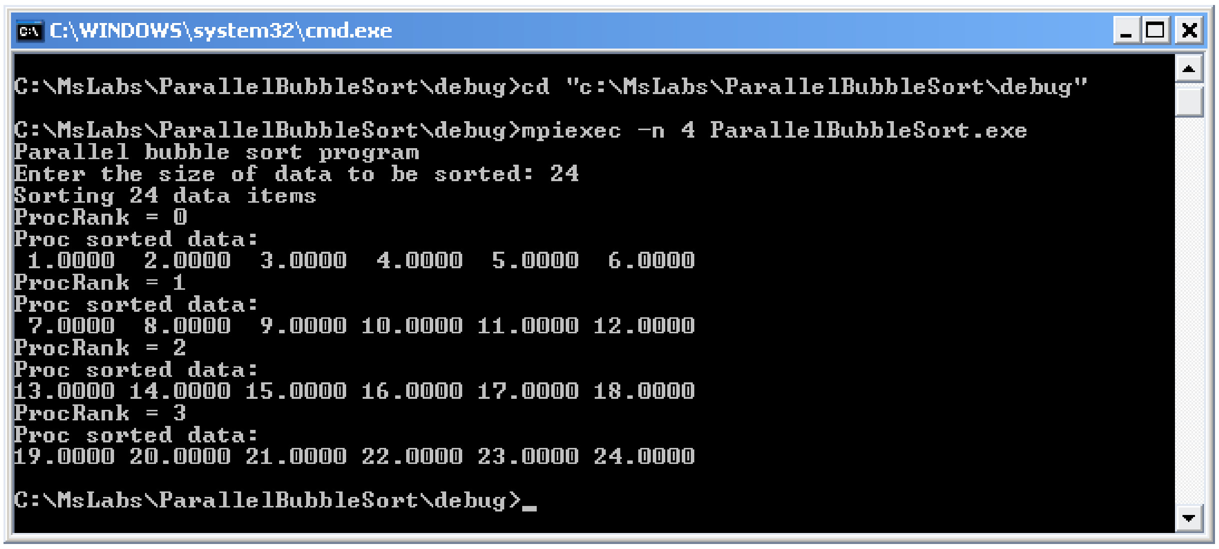
```
...
  for(int i = 0; i < 2 * ProcNum; i++) {
    if((i % 2) == 1) {
      if((ProcRank % 2) == 1) {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
      else {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
    }
    else {
      if((ProcRank % 2) == 1) {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
      else {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
    }
    // Check the first and last processes
    if((ProcRank == ProcNum - 1) && (Offset ==  1)) continue;
    if((ProcRank == 0          ) && (Offset == -1)) continue;

    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData);

    // Data merging
    merge(pProcData, pProcData + BlockSize, pDualData,
      pDualData + BlockSize, pMergedData);

    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData + BlockSize, pMergedData + 2*BlockSize, pProcData);
  }
...
```

It should be noted that if there are no necessary neighbors available (to the left for the processor with rank 0 and to the right for the process with the rank *ProcNum*–1) all the operations executed at the iteration for the process are omitted (two conditional statements before the call of the function *ExchangeData*).

This stage, as well as the previous ones, should be checked. Let us again make use of the debugging print with the help of the function *ParallelPrintData*. Transform all the previous calls of the function to comments and add a new call of this function immediately after the call of the sorting function *ParallelBubble*:

```
// Distributing the initial data among the processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);
// TestDistribution(pData, DataSize, pProcData, BlockSize);

  // Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);
  ParallelPrintData(pProcData, BlockSize);
```

For the data set by means of the function *DummyDataInitialization*, the result of sorting is known beforehand. The block of the result vector, which contains the element in the range from *i***BlockSize* + 1 to (*i* + 1)**BlockSize i* is obtained on the process with the rank *i.* Thus, for instance, if a parallel program is executed using four processes and the amount of the data to be sorted is equal to 24, then the block, which contains the numbers from 1 to 6 must be obtained on the first process, and the block containing the numbers from 7 to 12 – on the second process etc (Figure 4.10).



```
C:\MsLabs\ParallelBubbleSort\debug>cd "c:\MsLabs\ParallelBubbleSort\debug"

C:\MsLabs\ParallelBubbleSort\debug>mpiexec -n 4 ParallelBubbleSort.exe
Parallel bubble sort program
Enter the size of data to be sorted: 24
Sorting 24 data items
ProcRank = 0
Proc sorted data:
 1.0000  2.0000  3.0000  4.0000  5.0000  6.0000
ProcRank = 1
Proc sorted data:
 7.0000  8.0000  9.0000 10.0000 11.0000 12.0000
ProcRank = 2
Proc sorted data:
13.0000 14.0000 15.0000 16.0000 17.0000 18.0000
ProcRank = 3
Proc sorted data:
19.0000 20.0000 21.0000 22.0000 23.0000 24.0000

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

Figure 4.10. The Result of Testing the Sorted Blocks if the Program is Executed using Four Processes and the Amount of Data to Be Sorted Equal to 24

Compile and run the application. Check the correctness of obtaining partial results using the above given explanations while setting different number of processes and different amounts of data to be sorted.

Task 10 – Collect the Sorted Data

The final stage of completing all the operations is collecting the sorted data on the root process (the process with the rank 0). It should be noted that this stage is not mandatory in case of parallel sorting, as the amount of data to be sorted 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 example of a training Exercise and also for the purpose of final comparison of the serial and parallel sorting results.

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

```
// Function for data collection
void DataCollection(double *pData, int DataSize, double *pProcData,
  int BlockSize) {

  MPI_Gather(pProcData, BlockSize, MPI_DOUBLE, pData,
    BlockSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

The call of the function from the main program:

```
// Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);
  ParallelPrintData(pProcData, BlockSize);

  // Execution of data collection
  DataCollection(pData, DataSize, pProcData, BlockSize);
```

After the execution of the data collection, add print of the sorted data by means of the function *PrintData* 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 11 – Test the Parallel Program Correctness

After the program has been developed, 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 algorithm to the results of the parallel one. In order to execute the serial algorithm you may use the previously developed function *SerialBubbleSort*, which is copied in advance in the file *ParallelBubbleSortTest.cpp*.

To make the sorting function *SerialBubbleSort* operate the same data as the function *ParallelBubble*, it is necessary to create a copy of the data using the function *CopyData* (which is also located in the file *ParallelBubbleSortTest.cpp*):

```
// Function for copying the sorted data
void CopyData(double *pData, int DataSize, double *pDataCopy) {
  copy(pData, pData + DataSize, pDataCopy);
}
```

In order to check the correctness, let us compare the results of the serial sorting to the result of the parallel program element by element with the help of the function *CompareData*, which is also located in the file *ParallelBubbleSortTest.cpp*:

```
// Function for comparing the data
bool CompareData(double *pData1, double *pData2, int DataSize) {
  return equal(pData1, pData1 + DataSize, pData2);
}
```

Let us add the calls of these functions to the source code. It is necessary to declare the variable *pSerialData* for storing the copies of the data in the function *main*. It is also necessary to make ready this copy:

```
...
  int DataSize = 0;
  int BlockSize = 0;

  double *pSerialData = 0;

  // Process Initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  if (ProcRank == 0) {
    // Data copying
    pSerialData = new double[DataSize];
    CopyData(pData, DataSize, pSerialData);
  }
...
```

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

```
...
  ProcessTermination(pData, pProcData);
  if (ProcRank == 0)
    delete []pSerialData;

  MPI_Finalize();

  return 0;
}
```

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

```
// Function for testing the result of parallel bubble sort
void TestResult(double *pData, double *pSerialData, int DataSize) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialBubbleSort(pSerialData, DataSize);
    if(!CompareData(pData, pSerialData, DataSize)) {
      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 algorithm operation regardless of the size of the array to be sorted and with any initial data values with the help of this function.

Convert to the comment the call of the functions using the debugging print, which have been previously used for testing the correctness of the parallel program (the function *TestDistribution*, *TestPartialResult*). 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 experiments for different data amounts. Make sure that the program operates correctly.

Task 12 – Implement the Sorting for Any Data Amount

The parallel program, which was developed in the course of executing the previous Tasks, was implemented for the case when the number of the data to be sorted *DataSize* is divisible by the number of processors *ProcNum*. In this case the data is divided among the processes equally, and the sizes of blocks processed by each of the processes are equal.

Let us consider the case when the amount of data to be sorted *DataSize* is not multiple of the number of processes *ProcNum*. In this case the value *BlockSize* of the data to be sorted on each process may be different: some processes will get $\lfloor DataSize/ProcNum \rfloor$, and the rest of them - $\lceil Data\,Size/ProcNum \rceil$ elements of the sorted set (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 delete the processing of the situation in the function *ProcessInitialization* when the amount of data to be sorted is not divisible by the number of processes. Let us use the following algorithm of data distribution in order to determine the amount of process data. We will be allocating the data to the processes sequentially: first we will define how many values the process with the rank 0 will operate; then we will define the same for the process with the rank 1 etc. The process with the rank 0 will be assigned $\lfloor Data\,Size/ProcNum \rfloor$ values (the result of the operation $\lfloor\ \rfloor$ coincides with the result of the integer division of the variable *DataSize* by the variable *ProcNum*). After assigning the data for the process with the rank 0, it is necessary to distribute $DataSize - \lfloor DataSize/ProcNum \rfloor$ data elements among *ProcNum*-1 processes. The following process will be assigned the number of values, which is equal to the result of the integer division of the remaining number of values by the number of remaining processes etc. In the general case the process with the rank *i* should be assigned $\lfloor RestData/(ProcNum - i) \rfloor$ values (*RestData* is the remaining – not yet distributed–number of values).

Introduce the necessary changes in the code of the function *ProcessInitilization*:

```
// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize,
  double *&pProcData, int& BlockSize) {
...
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  int RestData = DataSize;
  for(int i = 0; i < ProcRank; i++)
    RestData -= RestData / (ProcNum - i);
  BlockSize = RestData / (ProcNum - ProcRank);
...
}
```

The necessary changes must be also performed for the function *DataDistribution*, as in case of different size blocks the general function *MPI_Scatterv* must be used for data distribution. As it has been demonstrated in the previous labs, in order to call the function *MPI_Scatterv*, it is necessary to define two auxiliary arrays. The size of the arrays coincides with the number of the available processes. Let us make the necessary changes in the code of the function *DataDistribution*:

```
// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData, int
BlockSize) {

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

  int RestData = DataSize;

  int CurrentSize = DataSize / ProcNum;
  pSendNum[0] = CurrentSize;
  pSendInd[0] = 0;
  for(int i = 1; i < ProcNum; i++) {
    RestData    -= CurrentSize;
    CurrentSize  = RestData / (ProcNum - i);
    pSendNum[i]  = CurrentSize;
    pSendInd[i]  = pSendInd[i - 1] + pSendNum[i - 1];
  }

  MPI_Scatterv(pData, pSendNum, pSendInd, MPI_DOUBLE, pProcData,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

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

On the analogy, for data collection it is necessary to use the more general function *MPI_Gatherv* instead of the function *MPI_Gather*, which is oriented at gathering data of the same size from all the communicator processes:

```
// Function for data collection
void DataCollection(double *pData, int DataSize, double *pProcData,
  int BlockSize) {
  // Allocate memory for temporary objects
  int *pReceiveNum = new int[ProcNum];
  int *pReceiveInd = new int[ProcNum];

  int RestData = DataSize;

  pReceiveInd[0] = 0;
  pReceiveNum[0] = DataSize / ProcNum;
  for(int i = 1; i < ProcNum; i++) {
    RestData -= pReceiveNum[i - 1];
    pReceiveNum[i] = RestData / (ProcNum - i);
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  MPI_Gatherv(pProcData, BlockSize, MPI_DOUBLE, pData,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);

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

Besides, we will have to make several changes in the function *ExchangeData*, as the block sizes of the neighboring processes may be different. Add one more parameter *DualBlockSize* to the heading of the function and change the function implementation with regard to the fact:

```
// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData, int DualBlockSize) {

  MPI_Status status;

  MPI_Sendrecv(pProcData, BlockSize, MPI_DOUBLE, DualRank, 0,
    pDualData, DualBlockSize, MPI_DOUBLE, DualRank, 0,
    MPI_COMM_WORLD, &status);
}
```

Then it is necessary to change the function *ParallelBubble*. Before the execution of the sorted block exchange it is necessary to find out the neighbors block size. It is possible to do it using the familiar function *MPI_Sendrecv*:

```
...
    MPI_Status status;

    int DualBlockSize;

    MPI_Sendrecv(&BlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      &DualBlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      MPI_COMM_WORLD, &status);

    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData,
      DualBlockSize);
...
```

Then it is necessary to take into account the fact that the variable *DualBlockSize* stores the amount of data to be sorted by the neighboring process, and to modify correspondingly the rest of the function *ParallelBubble*. First of all the changes will concern memory allocation/deallocation for the arrays *pDualData* and *pMergedData*. The calls of the functions *ExchangeData*, *merge* and *copy* will also be changed.

Make the necessary modifications in accordance with the code given below:

```
  for(int i = 0; i < ProcNum; i++) {
...
    MPI_Sendrecv(&BlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      &DualBlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      MPI_COMM_WORLD, &status);

    double *pDualData   = new double[DualBlockSize];
    double *pMergedData = new double[BlockSize + DualBlockSize];

    // Data exchange
    ExchangeData(pProcData, BlockSize, ProcRank + Offset, pDualData,
      DualBlockSize);

    //Data merging
    merge(pProcData, pProcData + BlockSize,
      pDualData, pDualData + DualBlockSize, pMergedData);

    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData + DualBlockSize, pMergedData + BlockSize +
        DualBlockSize, pProcData);

    delete []pDualData;
    delete []pMergedData;
  }
...
```

Compile and run the application. Test the correctness of sorting by means of the function *TestResult*.

After completing this Task it is possible to make the conclusion that the program for the parallel algorithm of bubble development is ready. If you have difficulty doing this or that Exercise, you should compare your program code with the the program code given in Appendix 2 of this lab.

Task 13 – Carry out the Computational Experiments

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

Let us determine the parallel algorithm execution time. For this purpose we will add clocking to the program code. As the parallel algorithm includes the stages of data distribution, data sorting on each process and collecting the sorted values, clocking should start immediately before the call of the function *DataDistribution*, and stop right after the execution of the function *DataCollection*:

```
...
  double start, finish;

  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  start = MPI_Wtime();
  // Distributing the initial data between the processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);

  // Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);

  // Process data collection
  DataCollection(pData, DataSize, pProcData, BlockSize);
  finish = MPI_Wtime();;

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

  // Process termination
  ProcessTermination(pData, pProcData);

  MPI_Finalize();
...
```

It is obvious that this way we will print the execution time spent 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 marked code to the main function. Compile and run the application. Carry out the computational experiments and register the results in the Table 4.4:

Table 4.4. The Results of the Computational Experiments for Parallel Method of Bubble Sorting

Test Number	Data Amount	Serial Bubble Sorting	Serial Standard Sorting	Parallel Algorithm. The Number of Processors		
				2	4	8
1	10					
2	100					
3	10,000					
4	20,000					

5	30,000					
6	40,000					
7	50,000					

The columns "Serial Bubble Sorting" and "Serial Standard Sorting" are assigned for writing the execution times of the serial algorithms of bubble sorting and the sorting offered by the C++ standard library measured in the course of testing the serial program in Exercise 2. Calculate the obtained computation speed up as the ration of the seqrial program time and the parallel program time and give the results in the Table 4.5 (in the column "Speed up 1" write the speed up of the parallel program in relation to the serial bubble sorting, and in the column "Speed up 2" - the same in relation to srial sorting from the standard library).

Table 4.5. Computation Speed Up Obtained for the Parallel Method of Bubble Sorting

Test Number	Parallel Algorithm					
	2 Processors		4 Processors		8 Processors	
	Speed Up 1	Speed Up 2	Speed Up 1	Speed Up 2	Speed Up 1	Speed Up 2
1						
2						
3						
4						
5						
6						
7						

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

$$T_p = ((n/p)\log_2(n/p) + 2n)\tau + p\cdot\left(\alpha + w\cdot\left(n/p\right)/\beta\right) \qquad (4.4)$$

(the derivation of this expression is considered in detail in subsection 10.3.5 of Section 10 "Parallel Methods of Data Sorting" of the training materials). Here *n* is the amount of the data to be sorted, *p* is the number of processes, τ is the execution time of the basic sorting operation (the 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 execution 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 program execution using formula (4.4). Tabulate the results in Table 4.6.

Table 4.6. The Comparison of the Experimental and the Theoretical Time of the Parallel Method of Bubble Sorting

Test Number	Data Size	Parallel Algorithm Execution Time					
		2 Processors		4 Processors		8 Processors	
		Model	Experiment	Model	Experiment	Model	Experiment
1	10						
2	100						
3	10,000						
4	20,000						
5	30,000						
6	40,000						
7	50,000						

Discussions

- The execution time of the first process was chosen as the parallel program execution time. What modifications should be made in the code in order to select the maximum time value among the time values obtained on all processes?

- How great is the difference between the execution time of the serial algorithm of bubble sorting and the parallel algorithm? Why?
- Compare the execution time of the parallel algorithm to the execution time of the serial algorithm and analyze the results. What conclusions can be made concerning the perspective of improving the obtained parallel algorithm? Could you have predicted such results?
- Was any speed up achieved in case of sorting 10 data elements? Why?
- Were the theoretical and the experiment execution time congruent? What might be the reason for incongruity?

Exercises

- Modify the developed application in such a way that the sorting from the standard library is used as the local sorting (for this purpose use the function *SerialStdSort*, located in the file *ParallelBubbleSortTest.cpp*). Carry out computational experiments according to the above-described scheme and compare the results of the new program variant to the previously obtained ones.
- Study other parallel sorting algorithms (Shell sorting, different types of quick sorting – see Section 10 "Parallel Methods of Data Sorting" of the training materials). Develop the program implementing these algorithms.

Appendix 1. The Program Code of the Serial Application of Bubble Sorting

File SerialBubbleSort.cpp

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <ctime>
#include "SerialBubbleSort.h"
#include "SerialBubbleSortTest.h"

using namespace std;

const double RandomDataMultiplier = 1000.0;

int main(int argc, char *argv[]) {
  double *pData = 0;
  int DataSize = 0;

  time_t start, finish;
  double duration = 0.0;

  printf("Serial bubble sort program\n");

  // Process initialization
  ProcessInitialization(pData, DataSize);

  printf("Data before sorting\n");
  PrintData(pData, DataSize);

  // Serial bubble sort
  start = clock();
  SerialBubble(pData, DataSize);
  finish = clock();

  printf("Data after sorting\n");
  PrintData(pData, DataSize);

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

  // Process termination
  ProcessTermination(pData);
  return 0;
}

// Function for allocating the memory and setting the initial values
void ProcessInitialization(double *&pData, int& DataSize) {
  do {
    printf("Enter the size of data to be sorted: ");
    scanf("%d", &DataSize);
    if(DataSize <= 0)
      printf("Data size should be greater than zero\n");
  }
  while(DataSize <= 0);

  printf("Sorting %d data items\n", DataSize);

  pData = new double[DataSize];

  // Simple setting the data
  DummyDataInitialization(pData, DataSize);

  // Setting the data by the random generator
  //RandomDataInitialization(pData, DataSize);
}

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

// Function for simple setting the initial data
void DummyDataInitialization(double*& pData, int& DataSize) {
  for(int i = 0; i < DataSize; i++)
    pData[i] = DataSize - i;
}

// Function for initializing the data by the random generator
void RandomDataInitialization(double *&pData, int& DataSize) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}

// Function for the serial bubble sort algorithm
void SerialBubble(double *pData, int DataSize) {
  double Tmp;

  for(int i = 1; i < DataSize; i++)
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}
```

File SerialBubbleSortTest.cpp

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

```
// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
  printf("\n");
}

// Sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}
```

Application 2. The Program Code of the Parallel Application of Bubble Sorting

File ParallelBubbleSort.cpp

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

#include "ParallelBubbleSort.h"
#include "ParallelBubbleSortTest.h"

using namespace std;

const double RandomDataMultiplier = 1000.0;

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

int main(int argc, char *argv[]) {
  double *pData = 0;
  double *pProcData = 0;
  int DataSize = 0;
  int BlockSize = 0;

  double *pSerialData = 0;

  double start, finish;
  double duration = 0.0;

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

  if(ProcRank == 0)
    printf("Parallel bubble sort program\n");

  // Process initialization
  ProcessInitialization(pData, DataSize, pProcData, BlockSize);

  if (ProcRank == 0) {
    pSerialData = new double[DataSize];
    CopyData(pData, DataSize, pSerialData);
  }

  start = MPI_Wtime();
  // Distributing the initial data between processes
  DataDistribution(pData, DataSize, pProcData, BlockSize);

  // Testing the data distribution
  TestDistribution(pData, DataSize, pProcData, BlockSize);

  // Parallel bubble sort
  ParallelBubble(pProcData, BlockSize);
  // Print the sorted data
  ParallelPrintData(pProcData, BlockSize);

  // Execution of data collection
  DataCollection(pData, DataSize, pProcData, BlockSize);
  TestResult(pData, pSerialData, DataSize);
  finish = MPI_Wtime();

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

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

  // Process termination
  ProcessTermination(pData, pProcData);

  MPI_Finalize();

  return 0;
}

// Function for allocating the memory and setting the initial values
void   ProcessInitialization(double   *&pData,   int&   DataSize,   double
*&pProcData, int& BlockSize) {
  setvbuf(stdout, 0, _IONBF, 0);
  if(ProcRank == 0) {
    do {
      printf("Enter the size of data to be sorted: ");
      scanf("%d", &DataSize);
      if(DataSize < ProcNum)
        printf("Data size should be greater than number of processes\n");

    } while(DataSize < ProcNum);

    printf("Sorting %d data items\n", DataSize);
  }

  // Broadcasting the data size
  MPI_Bcast(&DataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  int RestData = DataSize;
  for(int i = 0; i < ProcRank; i++)
    RestData -= RestData / (ProcNum - i);
  BlockSize = RestData / (ProcNum - ProcRank);

  pProcData = new double[BlockSize];

  if(ProcRank == 0) {
    pData = new double[DataSize];

    // Data initalization
    //RandomDataInitialization(pData, DataSize);
    DummyDataInitialization(pData, DataSize);
  }
```

```
}

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

  delete []pProcData;
}

// Function for simple setting the data to be sorted
void DummyDataInitialization(double*& pData, int& DataSize) {
  for(int i = 0; i < DataSize; i++)
    pData[i] = DataSize - i;
}

// Function for initializing the data by the random generator
void RandomDataInitialization(double *&pData, int& DataSize) {
  srand( (unsigned)time(0) );

  for(int i = 0; i < DataSize; i++)
    pData[i] = double(rand()) / RAND_MAX * RandomDataMultiplier;
}

// Data distribution among the processes
void DataDistribution(double *pData, int DataSize, double *pProcData, int
BlockSize) {

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

  int RestData = DataSize;

  int CurrentSize  = DataSize / ProcNum;
  pSendNum[0] = CurrentSize ;
  pSendInd[0] = 0;
  for(int i = 1; i < ProcNum; i++) {
    RestData    -= CurrentSize;
    CurrentSize  = RestData / (ProcNum - i);
    pSendNum[i]  = CurrentSize;
    pSendInd[i]  = pSendInd[i - 1] + pSendNum[i - 1];
  }

  MPI_Scatterv(pData, pSendNum, pSendInd, MPI_DOUBLE, pProcData,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

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

// Function for data collection
void DataCollection(double *pData, int DataSize, double *pProcData, int
BlockSize) {

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

  int RestData = DataSize;

  pReceiveInd[0] = 0;
  pReceiveNum[0] = DataSize / ProcNum;
  for(int i = 1; i < ProcNum; i++) {
    RestData -= pReceiveNum[i - 1];
    pReceiveNum[i] = RestData / (ProcNum - i);
    pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];
  }

  MPI_Gatherv(pProcData, BlockSize, MPI_DOUBLE, pData,
    pReceiveNum, pReceiveInd, MPI_DOUBLE, 0, MPI_COMM_WORLD);

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

// Parallel bubble sort algorithm
void ParallelBubble(double *pProcData, int BlockSize) {

  // Local sorting the process data
  SerialBubbleSort(pProcData, BlockSize);

  int Offset;
  split_mode SplitMode;

  for(int i = 0; i < ProcNum; i++) {
    if((i % 2) == 1) {
      if((ProcRank % 2) == 1) {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
      else {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
    }
    else {
      if((ProcRank % 2) == 1) {
        Offset    = -1;
        SplitMode = KeepSecondHalf;
      }
      else {
        Offset    = 1;
        SplitMode = KeepFirstHalf;
      }
    }

    // Check the first and last processes
    if((ProcRank == ProcNum - 1) && (Offset ==  1)) continue;
    if((ProcRank == 0          ) && (Offset == -1)) continue;

    MPI_Status status;

      int DualBlockSize;

    MPI_Sendrecv(&BlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      &DualBlockSize, 1, MPI_INT, ProcRank + Offset, 0,
      MPI_COMM_WORLD, &status);

    double *pDualData   = new double[DualBlockSize];
    double *pMergedData = new double[BlockSize + DualBlockSize];

    // Data exchange
    ExchangeData(pProcData,  BlockSize,  ProcRank  +  Offset,  pDualData,
DualBlockSize);

    // Data merging
    merge(pProcData,  pProcData  +  BlockSize,  pDualData,  pDualData  +
DualBlockSize, pMergedData);

    // Data splitting
    if(SplitMode == KeepFirstHalf)
      copy(pMergedData, pMergedData + BlockSize, pProcData);
    else
      copy(pMergedData  +  BlockSize,  pMergedData  +  BlockSize  +
DualBlockSize, pProcData);

    delete []pDualData;
    delete []pMergedData;
  }
}

// Function for data exchange between the neighboring processes
void ExchangeData(double *pProcData, int BlockSize, int DualRank,
  double *pDualData, int DualBlockSize) {

  MPI_Status status;
  MPI_Sendrecv(pProcData, BlockSize, MPI_DOUBLE, DualRank, 0,
    pDualData, DualBlockSize, MPI_DOUBLE, DualRank, 0,
    MPI_COMM_WORLD, &status);
}

// Function for testing the data distribution
void TestDistribution(double *pData, int DataSize, double *pProcData, int
BlockSize) {
  MPI_Barrier(MPI_COMM_WORLD);
  if (ProcRank == 0) {
    printf("Initial data:\n");
    PrintData(pData, DataSize);
  }

  MPI_Barrier(MPI_COMM_WORLD);

  for (int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Block:\n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}

// Function for parallel data output
void ParallelPrintData(double *pProcData, int BlockSize) {
  // Print the sorted data
  for(int i = 0; i < ProcNum; i++) {
    if (ProcRank == i) {
      printf("ProcRank = %d\n", ProcRank);
      printf("Proc sorted data: \n");
      PrintData(pProcData, BlockSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
}
```

```
// Function for testing the result of parallel bubble sort
void TestResult(double *pData, double *pSerialData, int DataSize) {
  MPI_Barrier(MPI_COMM_WORLD);

  if(ProcRank == 0) {
    SerialBubbleSort(pSerialData, DataSize);
    //SerialStdSort(pSerialData, DataSize);
    if(!CompareData(pData, pSerialData, DataSize))
      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");
  }
}
```

File ParallelBubbleSortTest.cpp

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

#include "ParallelBubbleSortTest.h"

using namespace std;

// Function for copying the sorted data
void CopyData(double *pData, int DataSize, double *pDataCopy) {
  copy(pData, pData + DataSize, pDataCopy);
}

// Function for comparing the data
bool CompareData(double *pData1, double *pData2, int DataSize) {
  return equal(pData1, pData1 + DataSize, pData2);
}

// Serial bubble sort algorithm
void SerialBubbleSort(double *pData, int DataSize) {
  double Tmp;

  for(int i = 1; i < DataSize; i++)
    for(int j = 0; j < DataSize - i; j++)
      if(pData[j] > pData[j + 1]) {
        Tmp          = pData[j];
        pData[j]     = pData[j + 1];
        pData[j + 1] = Tmp;
      }
}

// Sorting by the standard library algorithm
void SerialStdSort(double *pData, int DataSize) {
  sort(pData, pData + DataSize);
}

// Function for formatted data output
void PrintData(double *pData, int DataSize) {
  for(int i = 0; i < DataSize; i++)
    printf("%7.4f ", pData[i]);
  printf("\n");
}
```