10. Parallel Methods for Data Sorting

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

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

(hereinafter we will be discussing only the example of data sorting in the increasing order).

The possible methods of solving this problem are broadly discussed. The work by Knuth (1997) gives a complete survey of the data sorting algorithms. Among the latest editions we may recommend the work by Cormen et al. (2001).

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 with respect to the number of the data being sorted

$$T \sim n^2.$$

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

$$T \sim n \log_2 n .$$

This relation 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.

The extensive analysis of the data sorting problem should be a subject of further consideration. In this Section the main attention is devoted to the study of parallel methods of execution for a number of well known methods of *internal sorting*, when all the ordered data on each processor may be fully located in main memory.

This Section has been written based essentially on the teaching materials given in Kumar, at al. (1994) and Quinn (2004).

10.1. Parallelizing Principles

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 exchanging 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;
}
```

Example 10.1. Basic compare-exchange operation of many sorting procedures

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 consider the situation when the number of processors coincides with the number of values being sorted (i.e. $p=n$) and, as a result, there is only one value of the initial data on each processor. This consideration will be done for parallel generalization of the selected basic operation. Then the comparison of the values a_i and a_j, located correspondingly on processors P_i and P_i, may be organized in the following way (*a parallel generalization of the basic sorting operation*):

- Exchange the values available on processors P_i and P_i (the initial elements must be kept on the processors),
- Compare on each processor P_i and P_i the obtained identical pairs of values (a_i, a_j); the results of the comparison are used for data distribution among the processors: the smaller element remains on a processor (for instance, P_i), the other processor (i.e. P_j) stores the greater value of the pair for further processing

$$a_i' = \min(a_i, a_j) , \; a_j' = \max(a_i, a_j) .$$

10.2. Scaling Parallel Computations

Such parallel generalization of the basic sorting operation may be adequately adopted for the case when $p<n$, i.e. the number of processors is smaller than the number of the values being sorted. Each processor in this situation will already hold a part (*a block* of size n/p) of data being sorted.

Let us define the result of the parallel sorting algorithm execution as such the situation, when the data on the processors are sorted and the order of block distribution among the processors corresponds to linear numeration order (i.e. the value of the last element on the processor P_i is less or equal to the value of the first element on the processor P_{i+1}, where $0 \le i < p-1$).

Blocks are usually sorted at the very beginning of sorting on each processor separately by means of some fast algorithm (the initial stage of parallel sorting). Then in accordance with the described above scheme of a single value comparison, the interaction of the processors P_i and P_{i+1} for sorting the pair of blocks A_i and A_{i+1} can be implemented as follows:

- Execute the exchange of blocks between the processors P_i and P_{i+1},
- Unite the blocks A_i and A_{i+1} on each processor into a sorted block of double size (if blocks A_i and A_{i+1} have been initially sorted, the procedure of uniting is reduced to fast merging the sorted data),
- Subdivide the obtained double block into two equal parts and leave one of the parts (for instance, with smaller data values) on the processor P_i; then the other part (with the greater values correspondingly) must be located on the processor P_{i+1}

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

This procedure is usually called the *compare-split* operation. It should be noted that the blocks formed as a result of the procedure on the processors P_i and P_{i+1} are of the same size as the initial blocks A_i and A_{i+1} and all the values located on the processor P_i, do not exceed the values on the processor P_{i+1}.

The above mentioned compare-split operation may be defined as the basic computational subtask for organizing parallel computations. As it follows from its construction, the number of such subtasks parametrically depends on the number of the available processors. As a result, the problem of scaling the computations for parallel algorithms of data sorting became practically unnecessary. Alongside with this it should be noted that the data blocks of the subtasks change in the course of sorting. In simple cases the size of data blocks in the subtasks remains the same. In more complicated situations (as, for instance, in the quick sorting algorithms, - see Subsection 10.5) the amounts of data located on the processors may be different, which may lead to the violation of equal computational processor loading.

10.3. Bubble Sort

10.3.1. Sequential Algorithm

The sequential bubble sort algorithm (see, for instance, Knuth (1997), Cormen et al. (2001)) compares and exchanges the neighboring elements in a sequence to be sorted. For the sequence

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

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

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

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

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

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

```
// Algorithm 10.1.
// Sequential bubble sorting algorithm
BubbleSort(double A[], int n) {
  for (i=0; i<n-1; i++)
    for (j=0; j<n-i; j++)
      compare_exchange(A[j], A[j+1]);
}
```

Algorithm 10.1. The sequential bubble sort algorithm

10.3.2. Odd-Even Transposition Algorithm

The bubble sort algorithm is rather complicated for parallelizing. The comparison of the value pairs of the sorted data is strictly sequential. In this connection the modification of the algorithm, which is known as *the odd-even transposition*, is used in parallel application, - see, for instance, Kumar et al. (2003). The essence of modification may be described as follows: two different rules of executing the method iterations are introduced into the sort 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:

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

at even iterations the following elements are processed

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

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

```
//Algorithm 10.2
// 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]);
  }
}
```

Algorithm 10.2. The sequential odd-even transposition algorithm

10.3.3. Computation Decomposition and Analysis of Information Dependencies

Obtaining a parallel variant for 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 processor is less than the number of the values being sorted, the processors contain the data blocks of n/p size. The compare-split operation may be used as the basic computational subtask (see Subsection 10.2).

```
//Algorithm 10.3
// Parallel algorithm of odd-even transposition
ParallelOddEvenSort(double A[], int n) {
  int id = GetProcId();  // Process number
  int np = GetProcNum(); // Number of processors
  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);
    }
  }
}
```

Algorithm 10.3. The parallel odd-even transposition algorithm

To explain this parallel method of data sorting Figure 10.1 shows the example of data sorting when $n=16$, $p=4$ (i.e. the block of values on each processor holds $n/p=4$ elements). The number and type of the method iteration are given in the first column of the table. The same column shows the pairs of the processors, for which the compare-split operation are executed in parallel. The interacting pairs of processors are shown in the Table in double-lined frames. The Table shows the state of data being sorted for each sorting step before and after iteration execution.

Table 10.1. The example of data sorting by means of the parallel odd-even transposition method

№ and type of iteration	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

In the general case the execution of the parallel method may be terminated, if there are no changes in the state of the data being sorted during two sequential iterations of sorting. As a result, the total number of iterations may be reduced. To implement such modification a control processor should be introduced for fixing such situations. This processor should determine the state of the data after the execution of each sorting iteration. However, the complexity of this communication operation (gathering the messages from all the processors) may be so significant that the overhead of data communications will exceed the effect of the possible reduction of method iterations.

10.3.4. Scaling and Distributing Subtasks among Processors

As it has been previously mentioned, the number of subtasks corresponds to the number of the available processors. As a result, there is no need for computation scaling. The initial distribution of the blocks of the data being sorted among the processors may be randomly chosen. In order to execute the discussed parallel sorting algorithm efficiently, it is necessary that all the processors with the neighboring numbers should have direct communication lines.

10.3.5. Efficiency Analysis

Let us estimate the general complexity of the discussed parallel sort algorithm and then add the complexity characteristics of the performed communications to the obtained relations.

Let us first determine the complexity of the sequential computations. The bubble sort algorithm allows to demonstrate a very important aspect in consideration of this problem. As it has been already mentioned, the method of data sorting used for parallelizing is characterized by a square dependence of complexity with respect to the number of data being sorted, i.e. $T_1 \sim n^2$. However, application of this nonoptimal complexity estimation of the sequential algorithm will lead to the distortion of the quality criteria "sense" of parallel computations. In this case the efficiency characteristics would rather refer to the parallel execution of a given sort method than to the effectiveness of using parallelism for solving the problem of data sorting on the whole. The difference is that more efficient sequential algorithms may be used for sorting and complexity of these algorithms is the order:

$$T_1 = n \log_2 n . \qquad (10.1)$$

It is essential to use this very complexity estimation in order to compare, how faster the data may be sorted by means of parallel computations. As a result, we can formulate the following: *the efficiency of the best sequential algorithm should be used as the estimation of the complexity for the sequential method of solving the problem under consideration in determining the speedup and efficiency characteristics for parallel computations*. Parallel methods of solving problems should be compared to the most efficient fast sequential computational methods!

Let us determine now the complexity of the described parallel algorithm of data sorting. As it has been previously mentioned, each processor at the initial stage of the method operation sorts out its data blocks (the size of blocks in case of equal data distribution is equal to n/p). Let us assume that this initial sorting may be performed by means of the best sequential sort algorithms. The complexity of the initial computational stage may be determined in this case by the following relation:

$$T_p^1 = (n/p)\log_2(n/p) . \qquad (10.2)$$

Then at each iteration of parallel sorting the interacting pairs of processors exchange the blocks with each other. The block pairs formed on each processor are united using the merge procedure. The total number of iterations does not exceed the value p. As a result, the total number of operations in this part of parallel computations appears to be equal to the following:

$$T_p^2 = 2p(n/p) = 2n . \qquad (10.3)$$

With regard to the obtained relations the efficiency and speedup characteristics for the parallel method of data sorting look as follows:

$$S_p = \frac{n\log_2 n}{(n/p)\log_2(n/p)+2n} = \frac{p\log_2 n}{\log_2(n/p)+2p},$$
$$E_p = \frac{n\log_2 n}{p((n/p)\log_2(n/p)+2n)} = \frac{\log_2 n}{\log_2(n/p)+2p} \quad . \tag{10.4}$$

Let us enhance these expressions by taking into account the duration of the computational operations performed and estimate the complexity of the block exchange between the processors. In case when the Hockney model is used, the total execution time for all the block exchanges performed in the course of sorting may be estimated by means of the following relation:

$$T_p(comm) = p\cdot(\alpha + w\cdot(n/p)/\beta), \tag{10.5}$$

where α is the latency, β is the network bandwidth, and w is the size of the data element in bytes.

With regard to the complexity of the communication operations the total execution time of the parallel data sort algorithm is determined by the following expression:

$$T_p = ((n/p)\log_2(n/p)+2n)\tau + p\cdot(\alpha + w\cdot(n/p)/\beta), \tag{10.6}$$

where τ is the execution time of the basic sorting operation.

10.3.6. Computational Experiment Results

The computational experiments for estimating the efficiency of the parallel bubble sort algorithm were carried out under the conditions described in 7.6.5. In brief terms these conditions are the following.

The experiments were carried out on the computational cluster on the basis of the processor Intel XEON 4 EM64T 3000 Mhz and Gigabit Ethernet under OS Microsoft Windows Server 2003 Standart x64 Edition (see 1.2.3).

To estimate the duration τ of the basic sorting operation we solve the problem of ordering by means of a sequential algorithm. The time of computations obtained this way was further divided by the total number of operations. The value 10.41 nsec was obtained for τ as a result of the experiments. The experiments carried out in order to determine the network parameters showed the value of the latency α and the value of the network bandwidth β correspondingly 130 msec and 53.29 Mbyte/sec. All the computations were executed with the numerical values of double type, i.e. the value w is equal to 8 bytes.

The results of the computational experiments are given in Table 10.1. The experiments were carried out with the use of two and four processors.

Table 10.1. The results of the computational experiments for the parallel bubble sort algorithm

Number of elements	Sequential algorithm	Parallel algorithm			
		2 processors		4 processors	
		Time	Speedup	Time	Speedup
10,000	0.001422	0.002210	0.643439	0.003270	0.434862
20,000	0.002991	0.004428	0.675474	0.004596	0.650783
30,000	0.004612	0.006745	0.683766	0.006873	0.671032
40,000	0.006297	0.008033	0.783891	0.009107	0.691446
50,000	0.008014	0.009770	0.820266	0.010840	0.739299

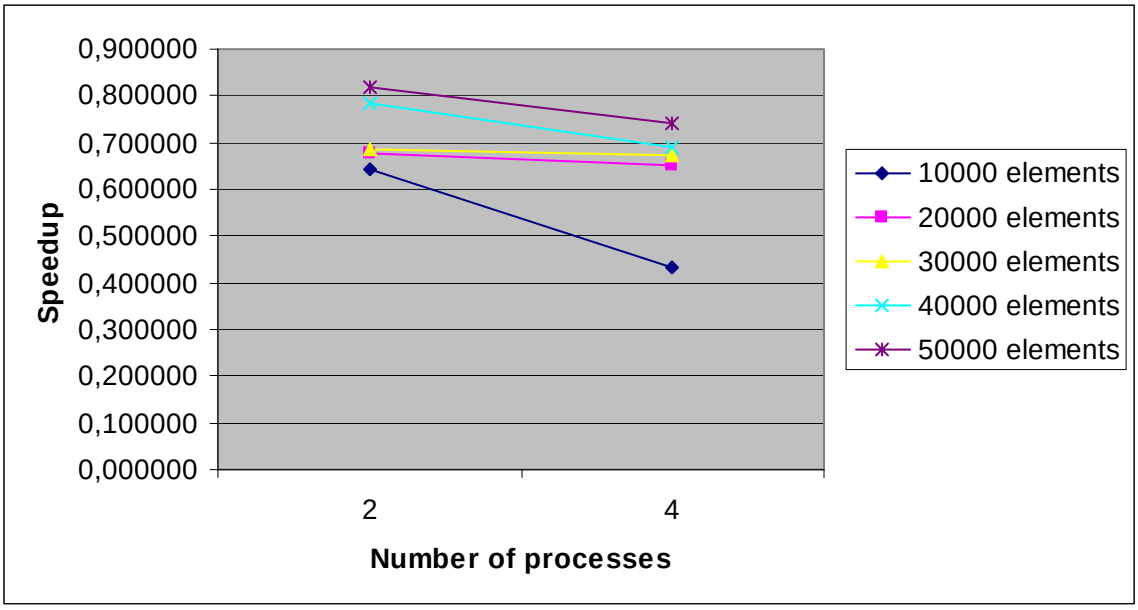

Figure 10.1. Speedup of the parallel bubble sort algorithm

According to the experimental results of the computational experiments the parallel bubble sort algorithm operates more slowly than the original sequential method of bubble sorting. The reason for it is that the volume of the data transmitted among the processors is rather large and is comparable to the number of the executed computational operations (this disbalance of the amount of computations and the complexity of data communications grows with the increase of the number of processors).

The comparison of the experimental execution time T_p^* and the theoretical estimation T_p from (10.5) is given in Table 10.2 and Figure 10.2.

Table 10.2. The comparison of the experimental and theoretical execution time for the parallel bubble sort algorithm

Data size	Parallel algorithm			
	2 processors		4 processors	
	T_2	T_2^*	T_4	T_4^*
10,000	0.002003	0.002210	0.002057	0.003270
20,000	0.003709	0.004428	0.003366	0.004596
30,000	0.005455	0.006745	0.004694	0.006873
40,000	0.007227	0.008033	0.006035	0.009107
50,000	0.009018	0.009770	0.007386	0.010840

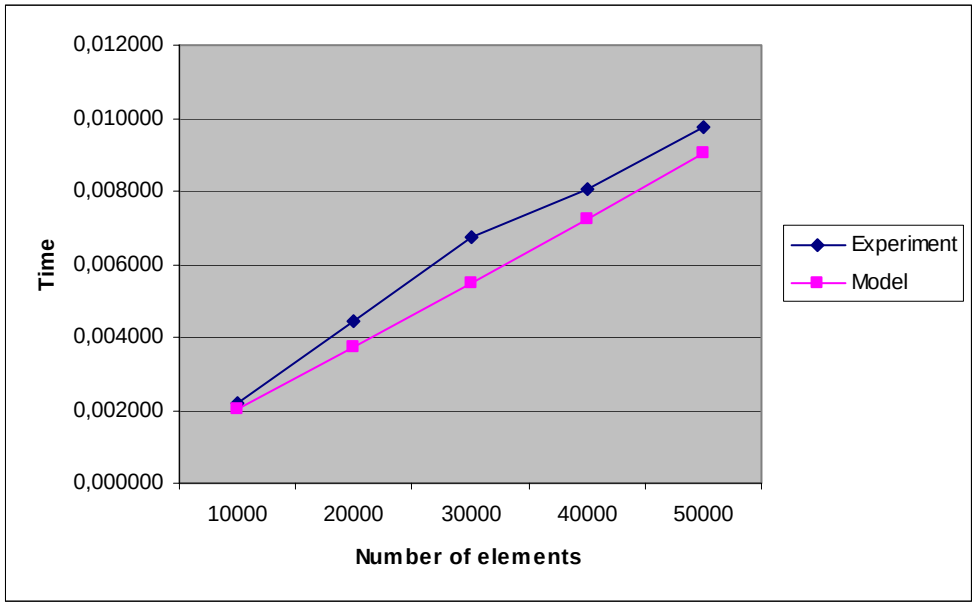

Figure 10.2. Experimental and theoretical execution time for 2 Processors

10.4. Shell Sort

10.4.1. Sequential Algorithm

In case of the Shell sort algorithm (see, for instance, например, Knuth (1997), Cormen et al. (2001)) from the very beginning the compared pairs of values are formed from elements that are located rather far from each other in the sorted data. This modification of the sort method makes possible to permute unsorted pairs of distant located values fast enough (sorting such pairs usually requires a greater number of permutation operations, if only neighboring elements are compared).

The general scheme of the method is described below. The elements of $n/2$ pairs $(a_i, a_{n/2+i})$ for $1 \le i \le n/2$.are sorted during the first step of the algorithm. The elements of n/4 groups of four elements each $(a_i, a_{n/4+1}, a_{n/2+1}, a_{3n/4+1})$ for $1 \le i \le n/4$ are sorted during the second step. During the third step the elements of $n/8$ groups of eight elements each are sorted etc. All the elements of the array $(a_1, a_2, \ldots, a_n)$ are sorted at the last step. The insertion sort method is used at each step for sorting elements in groups. As it can be noted the total number of iterations of the Shell algorithm is equal to $log_2 n$.

The Shell algorithm can be presented in a simpler way as it is shown below:

```
// Algorithm 10.4
// Sequential algorithm of Shell sorting
ShellSort(double A[], int n){
  int incr = n/2;
  while( incr > 0 ) {
    for ( int i=incr+1; i<n; i++  ) {
      j = i-incr;
      while ( j > 0 )
        if ( A[j] > A[j+incr] ){
          swap(A[j], A[j+incr]);
          j = j - incr;
        }
        else j = 0;
    }
    incr = incr/2;
  }
}
```

Algorithm 10.4. The sequential Shell sort algorithm

10.4.2. Parallel Algorithm

A parallel variant of the Shell sort method may be suggested (see, for instance, Kumar et al. (2003)), if the communication network topology may be presented as an *N*-dimensional hypercube (if the number of processors is

equal to $p=2^N$). In this case sorting may be subdivided into two sequential stages. The interaction of the processors neighboring in the hypercube structure takes place at the first stage (N iterations). These processors may appear to be rather far from each other in case of linear enumeration. The required mapping the hypercube topology into the linear array structure may be implemented using the Gray code (see Section 3). Forming the pairs of processors interacting with each other during the compare-split operation may be provided by means of the following simple rule: the processors whose bit codes of their numbers differ only in position N-i are paired at each iteration i, $0 \le i < N$.

At the second stage the usual iterations of the parallel odd-even transposition algorithm are performed. The iterations of this stage are executed up to the actual termination of changes of the data being sorted. Thus, the total number L of such iterations may vary from 2 to p.

Figure 10.3 shows the example of sorting the array, which consists of 16 elements by means of the discussed method. It should be noted that the data appears to be sorted after the completion of the first stage, and there is no need to execute the odd-even transposition iterations.

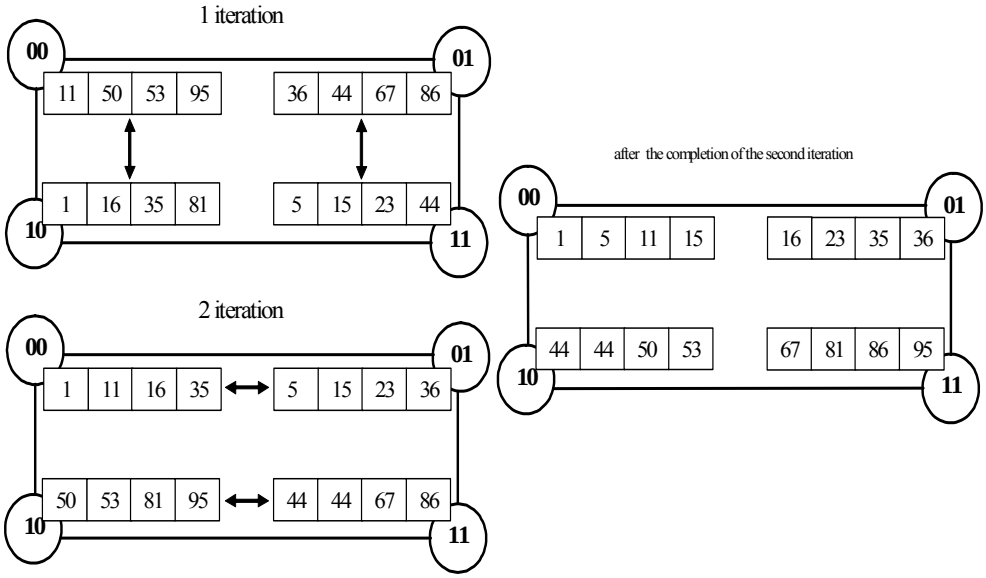

Figure 10.3. The example of the use of the parallel Shell algorithm for 4 processors (the processors are marked by circles, the processor numbers are given in their binary representation)

With regard to the given description the same decomposition approach can be applied and define the compare-split operation as the basic computational subtask. As a result, the number of subtasks will coincides with the number of the available processors (the size of the data blocks in the subtasks is equal to n/p). As a result, scaling the computations is not needed again. The distribution of the sorted data among the processors should be selected with regard to the efficient implementation of the compare-split operation in the hypercube network topology.

10.4.3. Efficiency Analysis

The relations obtained for parallel bubble sort method of (see Subsection 10.3.5) may be used for estimating the efficiency of the parallel variant of the Shell algorithm. It is only necessary to take into account the two stages of the Shell algorithm. With regard to this peculiarity the total execution time for the new parallel method may be determined by means of the following expression:

$$T_p = (n/p)\log_2(n/p)\tau + (\log_2 p + L)[(2n/p)\tau + (\alpha + w\cdot(n/p)/\beta)]. \qquad (10.7)$$

As it can be noted, the efficiency of the parallel variant of Shell sorting depends considerably on the value L. If the value L is small, the new parallel sorting method is executed more quickly than the previously described odd-even transposition algorithm.

10.4.4. Computational Experiment Results

The computational experiments for estimating the efficiency of the Shell sort parallel method were carried out under the same conditions as the experiments described previously (see 10.3.4).

The results of the computational experiments are given in Table 10.4. The experiments were carried out with the use of 2 and 4 processors. The time is given in seconds.

Table 10.4. The results of the computational experiments for the parallel Shell sort algorithm

Number of	Sequential	Parallel algorithm

elements	algorithm	2 processors		4 processors	
		Time	Speedup	Time	Speedup
10,000	0.001422	0.002959	0.480568	0.007509	0.189373
20,000	0.002991	0.004557	0.656353	0.009826	0.304396
30,000	0.004612	0.006118	0.753841	0.012431	0.371008
40,000	0.006297	0.008461	0.744238	0.017009	0.370216
50,000	0.008014	0.009920	0.807863	0.019419	0.412689

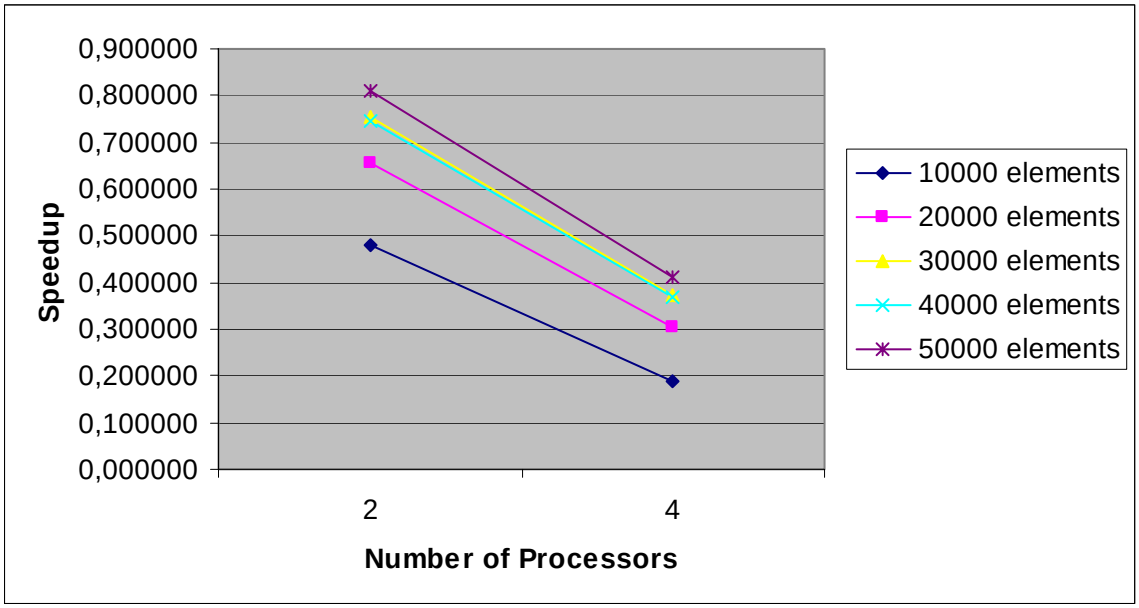

Figure 10.4. Speedup the parallel Shell sort algorithm

The comparison of the experimental execution time T_p^* and the theoretical estimation T_p from (10.7) is given in Table 10.5 and Figure 10.5.

Table 10.5. The comparison of the experimental and theoretical execution time for the Shell sort parallel algorithm

Number of elements	Parallel algorithm			
	2 processors		4 processors	
	T_2	T_2^*	T_4	T_4^*
10,000	0.002684	0.002959	0.002938	0.007509
20,000	0.004872	0.004557	0.004729	0.009826
30,000	0.007100	0.006118	0.006538	0.012431
40,000	0.009353	0.008461	0.008361	0.017009
50,000	0.011625	0.009920	0.010193	0.019419

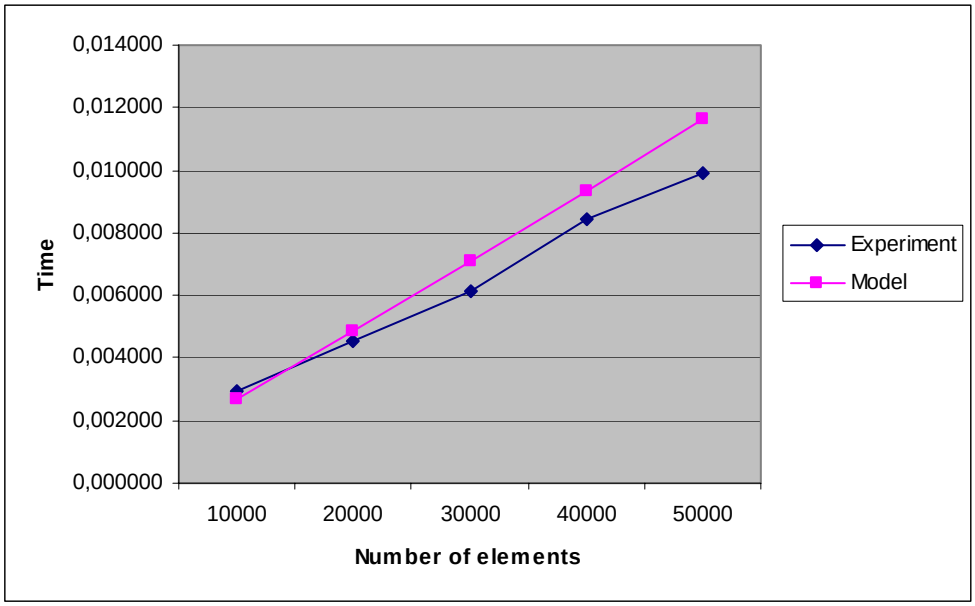

Figure 10.5. Experimental and theoretical execution time for 2 processors

10.5. Quick Sort

10.5.1. Sequential Algorithm

In the general consideration of the quick sort algorithm suggested by Hoare C.A.R., first of all it should be noted that the method is based on the sequential subdividing the sorted data into blocks of smaller sizes in such a way that the ordering relation is provided among the values of different blocks (for any pair of blocks all the values of one of the blocks do not exceed the values of the other one). The division of the original data into the first two parts is performed at the first iteration of the method. A certain *pivot element* is selected for providing this division, and all the values of the data, which are smaller that the pivot element, are transferred to the first block being formed. All the rest of the values form the second block of the sorted data. These rules are applied recursively for the two created blocks on the second iteration of the sorting etc. If the choice of the pivot elements is adequate, than the initial data array appears to be sorted after the execution of $log_2 n$ iterations. More detailed information concerning the method may be found in Knuth (1997), Cormen et al. (2001).

The quick sort method efficiency is determined to a great extent by the choice of the pivot elements during the data division into blocks. At worst case the complexity of the method is of the same order of complexity as the bubble sort method (i.e. $T_1 \sim n^2$). If the choice of the pivot elements is optimal, than each block is divided into equal sized parts and the complexity of the algorithm coincides with the complexity of the most efficient sort methods ($T_1 \sim n \log_2 n$). On average the number of the operations carried out by the quick sort algorithm is determined by the following expression (see, for instance, Knuth (1997), Cormen et al. (2001)):

$T_1 = 1.4 n \log_2 n$.

The general scheme of the quick sorting algorithm may be given in the following form (the pivot element is determined by the first element value of the sorted data):

```
// Algorithm 10.5
// The sequential Algorithm of Quick Sorting
QuickSort(double A[], int i1, int i2) {
  if ( i1 < i2 ){
    double pivot = A[i1];
    int is = i1;
    for ( int i = i1+1; i<i2; i++ )
      if ( A[i] ≤ pivot ) {
        is = is + 1;
        swap(A[is], A[i]);
      }
    swap(A[i1], A[is]);
    QuickSort (A, i1, is);
    QuickSort (A, is+1, i2);
```

```
  }
}
```

Algorithm 10.5. The sequential quick sort slgorithm

10.5.2. The Parallel Quick Sort Algorithm

10.5.2.1. Parallel Computational Scheme

The parallel generalization of the quick sorting algorithm (see, for instance, Quinn (2004)) may be obtained in the simplest way for a computer system, the topology of which is an N-dimensional hypercube (i.e. $p = 2^N$). Let the initial data, as previously, be distributed among the processors in blocks of the same size n/p. The resulting location of blocks must correspond to the enumeration of the hypercube processors. Under these conditions a possible method to execute the first iteration of the parallel method is the following:

- Select the pivot element and broadcast it to all the processors (for instance, the arithmetic mean of the elements of some pivot processor may be chosen as the pivot element);
- Subdivide the data block available on each processor into two parts using the pivot element;
- Form the pairs of processors, for which the bit presentation of the numbers differs only in N position. After that the exchange of the data among these processors should be executed. As a result of these data transmissions, the parts of the blocks with the values smaller than the pivot element must appear on the processors, for which the bit position N of the processor numbers are equal to 0. The processors with the numbers in which the bit N is equal to 1 must collect correspondingly all the data values exceeding the value of the pivot element.

As a result of executing this iteration, the initial data appear to be subdivided into two parts. One of them (with the values smaller than the pivot element value) is located on the processors, whose numbers hold 0 in the N-th bit. There are only $p/2$ such processors. Thus, the initial N-dimensional hypercube also is subdivided into two subhypercubes of $N-1$ dimension. The above described procedure may also be applied to these subhypercubes. After executing N such iterations, it is sufficient to sort the data blocks which have been formed on each separate processor to terminate the method.

To illustrate the parallel quick algorithm Figure 10.6 shows the example of sorting data when $n = 16$, $p = 4$ (i.e. each processor block holds four elements). The processors are shown as rectangles, the data blocks being sorted are shown inside the rectangles. The block values are given at the beginning and at the completion of each sorting iteration. The interacting pairs of processors are linked by double-headed arrows. The optimal values of the pivot elements were chosen for data partitioning. At the first iteration the value 0 was used for all the processors. At the second iteration for the pair of processors (0, 1) the pivot element was equal to 4, for the pair (2, 3) the value was chosen to be equal to -5.

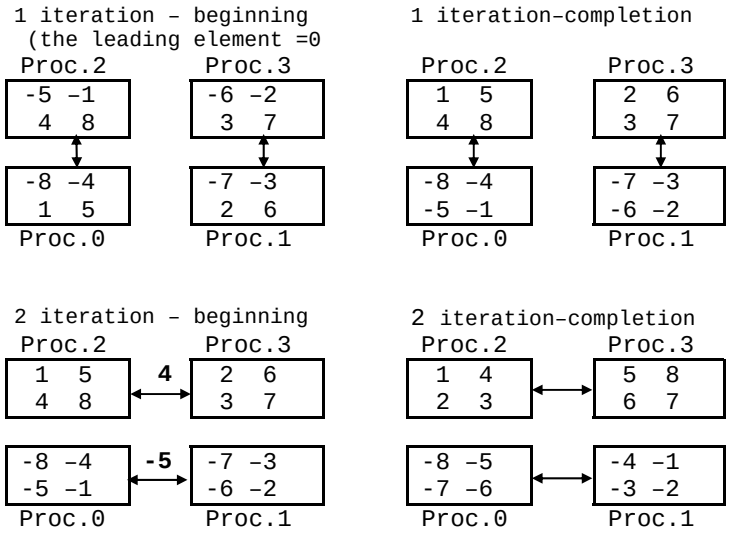

Figure 10.6. The example of sorting data by the parallel quick sort method of (the results of local block sorting are not included)

As previously, the basic computational subtask may be the compare-split operation. The number of the subtasks coincides with the number of the processors used. The distribution of the subtasks among the processors should be done with regard to the efficient algorithm execution for the hypercube network topology.

10.5.2.2. Efficiency Analysis

Let us estimate the complexity of the described parallel method. Let us assume that we have an *N*-dimensional hypercube (i.e. $p = 2^N$) and $p < n$.

The efficiency of the parallel quick sort method depends largely on the optimality of the pivot element choice, as it was in case of the sequential variant. It is rather complicated to work out the general rule for the selection of these values. But this choice can be implemented easier if at the beginning of the method execution the processor data blocks are sorted. It is also useful to provide the more uniform data distribution among the processors.

Let us determine the computational complexity of the sort algorithm. At each of $log_2 p$ sorting iterations each processor divides the data block with regard to the pivot element. The complexity of this stage is n/p operations (let us consider the best possible case that each block is divided into equally sized parts at each sorting iteration).

After the termination of the computations the processors carry out sorting the blocks. It may be done in $(n/p)log_2(n/p)$ operations by means of using the quick sort algorithm.

Thus, the total computational time for the parallel quick sort algorithm is the following:

$$T_p(calc) = [(n/p)\log_2 p + (n/p)\log_2(n/p)]\tau, \qquad (10.8)$$

where τ is the execution time of the basic sorting operation.

Let us consider the complexity of the communication operations. The total number of the processor communications to broadcast the pivot elements for the *N*-dimensional hypercube may be evaluated by the following estimation:

$$\sum_{i=1}^{N} i = N(N+1)/2 = \log_2 p\,(\log_2 p + 1)/2 \sim (\log_2 p)^2. \qquad (10.9)$$

With regard to the assumption we have made (the choice of the pivot elements is optimal), we define the number of the algorithm iterations as equal to $log_2 p$, and the amount of the transmitted data as always equal to a half of the block, i.e. $(n/p)/2$. Under these conditions, the communication complexity of the parallel algorithm for the quick sort method is determined by means of the following relation:

$$T_p(comm) = (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta), \qquad (10.10)$$

where α is the latency, β is the network bandwidth, and w is the size of the set element in bytes.

Finally we may determine the algorithm time complexity by the following expression:

$$T_p = [(n/p)\log_2 p + (n/p)\log_2(n/p)]\tau + (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta). \qquad (10.11)$$

10.5.2.3. Computational Experiments Results

The computational experiments for estimating the efficiency of the parallel quick sort method were carried out under the same conditions as the experiments described previously (see 10.3.4).

The results of the computational experiments are given in Table 10.6. The experiments were carried out with the use of 2 and 4 processors. The time is given in seconds.

Table 10.6. The results of the computational experiments for the parallel quick sort algorithm

Number of elements	Sequential algorithm	Parallel algorithm			
		2 processors		4 processors	
		Time	Speedup	Time	Speedup
10,000	0.001422	0.001521	0.934911	0.003434	0.414094
20,000	0.002991	0.002234	1.338854	0.004094	0.730581
30,000	0.004612	0.003080	1.497403	0.005088	0.906447
40,000	0.006297	0.004363	1.443273	0.005906	1.066204
50,000	0.008014	0.005486	1.460809	0.006635	1.207837

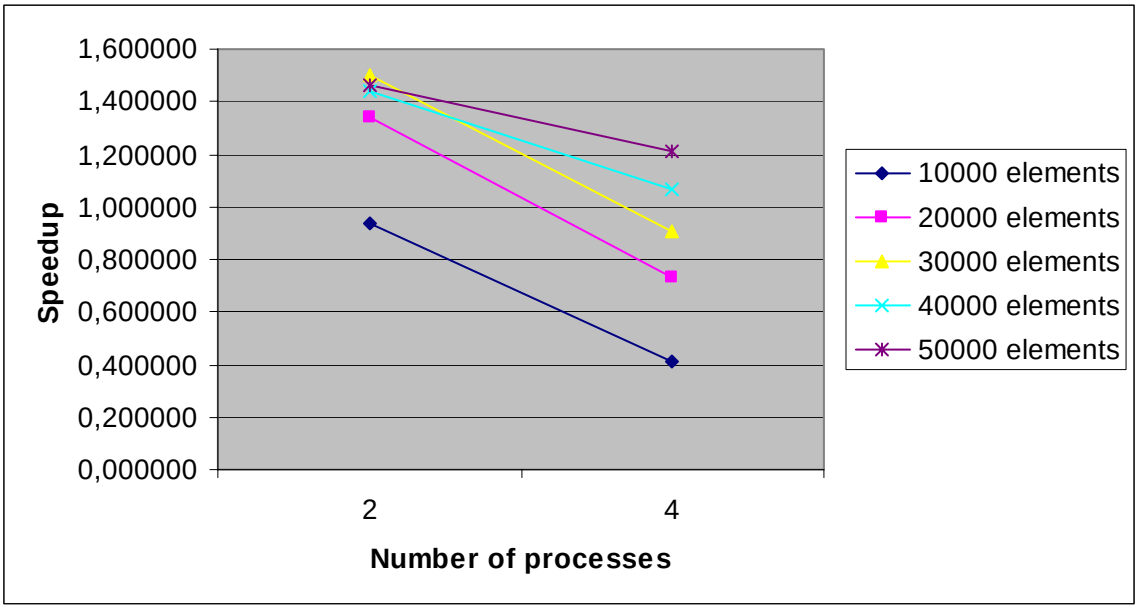

Figure 10.7. Speedup of the parallel quick sort algorithm

According to the results of the computational experiments, the parallel quick sort algorithm allows to speed up solving the problem of data sorting.

The comparison of the experimental execution time T_p^* and the theoretical estimation T_p from (10.11) is given in Table 10.7 and Figure 10.8.

Table 10.7. The comparison of the experimental and theoretical execution time for the parallel quick sort algorithm

Data size	Parallel algorithm			
	2 processors		4 processors	
	T_2	T_2^*	T_4	T_4^*
10,000	0.001280	0.001521	0.001735	0.003434
20,000	0.002265	0.002234	0.002321	0.004094
30,000	0.003289	0.003080	0.002928	0.005088
40,000	0.004338	0.004363	0.003547	0.005906
50,000	0.005407	0.005486	0.004175	0.006635

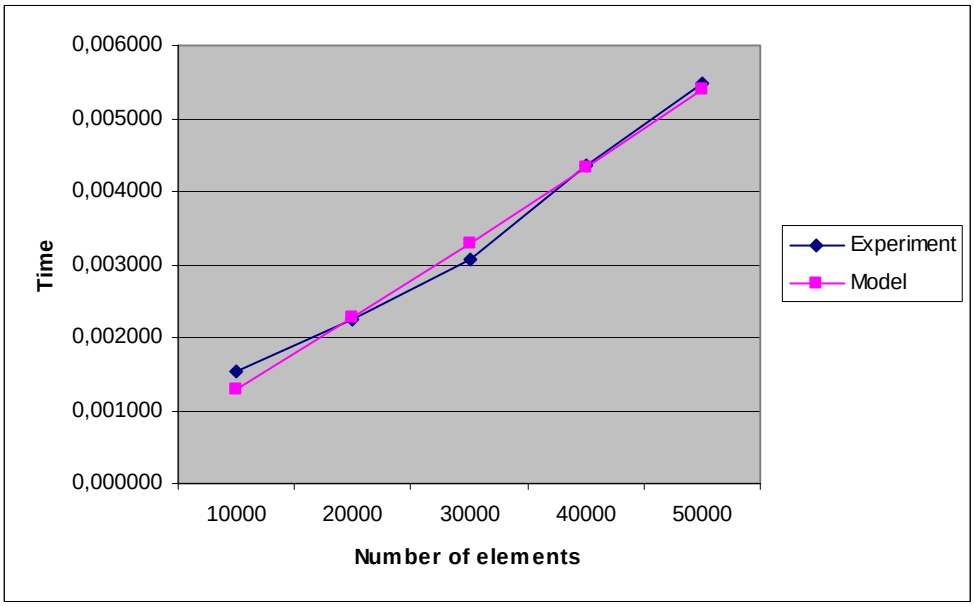

Figure 10.8. Experimental and theoretical execution time for 2 processors

10.5.3. The Parallel HyperQuickSort Algorithm

In addition to the above described quick sort method, there is a generalized technique called *the HyperQuickSort algorithm* which suggests a specific scheme for choosing the pivot elements. In accordance with this scheme the data blocks located on the processors should be sorted at the very beginning of the computations. Besides, the processors should merge the parts of blocks obtained after their partitioning so as to maintain data ordering in the course of computations. As a result, due to the regularity of blocks, it is reasonable to choose the average element of some block (for instance, the block on the first processor) as the pivot element at each iteration of the quick sort algorithm. In some cases the pivot elements selected in such a way may appear to be very closer to real arithmetic mean of the data being sorted than any other randomly chosen value.

All the other operations in the algorithm being described are executed according to the original quick sort method. In detail the HyperQuickSort algorithm is described, for instance, in Quinn (2004).

It is possible to use the relation (10.11) for analyzing the efficiency of the HyperQuickSort algorithm. It should be noted that the operation of merging block parts is carried out at each method iteration (as previously we assume that the size of the block parts is the same and is equal to *n/p)/2*). Besides, the procedure of partitioning may be modified due to the block regularity. It is sufficient to carry out the binary search for the pivot element position in a block instead of exhaustive linear search through all the block elements. With regard to this, the complexity of the HyperQuickSort algorithm may be determined by means of the following expression:

$$T_p = [(n/p)\log_2(n/p) + (\log_2(n/p) + (n/p))\log_2 p]\tau + (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta). \quad (10.12)$$

10.5.3.1. Software Implementation

Let us discuss a possible variant of software implementation of the HyperQuickSort algorithm. It should be noted that program code of several modules is not given as its absence does not influence the understanding of the general scheme of parallel computations.

1. The main function. The main function implements the computational method scheme by sequential calling out the necessary subprograms.

```
// Program 10.1.
// The HyperQuickSort Method
  int ProcRank;       // Rank of current process
  int ProcNum;        // Number of processes
int main(int argc, char *argv[]) {
  double *pProcData;     // Data block for the process
  int     ProcDataSize;  // Data block size

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

  // Data Initialization and their distribution among the processors
  ProcessInitialization ( &pProcData, &ProcDataSize);

  // Parallel sorting
  ParallelHyperQuickSort ( pProcData, ProcDataSize );

  // The termination of process computations
  ProcessTermination ( pProcData, ProcDataSize );

  MPI_Finalize();
}
```

The function *ProcessInitialization* determines the initial data for the problem being solved (the size of the data being sorted). It also allocates memory for data storage and generates the data being sorted (for instance, by means of random number generator). The function also distributes the data among the processes.

The function *ProcessTermination* performs the necessary output of the sorted data and releases all the previously allocated memory for storing the data.

The implementation of all the above mentioned functions may be performed on the analogy with the examples, which have been discussed earlier and is given to the reader a training exercise.

2. The function ParallelHyperQuickSort. It performs parallel quick sorting according to the algorithm, which has been described above.

```
// The Parallel HyperQuickSort Method
void ParallelHyperQuickSort ( double *pProcData, int ProcDataSize) {
  MPI_Status status;
  int CommProcRank;   // Rank of the processor involved in communications
  double *pMergeData, // Block obtained after merging the block parts
   *pData,     // Block part, which remains on the processor
   *pSendData, // Block part, which is sent to the processor CommProcRank
   *pRecvData; // Block part, which is received from the proc CommProcRank
  int     DataSize, SendDataSize, RecvDataSize, MergeDataSize;
  int HypercubeDim = (int)ceil(log(ProcNum)/log(2)); // Hypercube dimension
  int Mask = ProcNum;
  double Pivot;

  // Local data sorting
  LocalDataSort ( pProcData, ProcDataSize );

  // Hyperquicksort iterations
  for ( int i = HypercubeDim; i > 0; i-- ) {

    // Determination of the pivot value and broadcast it to processors
    PivotDistribution (pProcData,ProcDataSize,HypercubeDim,Mask,i,&Pivot);
    Mask = Mask >> 1;

    // Determination of the data division position
    int pos = GetProcDataDivisionPos (pProcData, ProcDataSize, Pivot);

    // Block division
    if ( ( (rank&Mask) >> (i-1) ) == 0 ) { // high order bit= 0
      pSendData = & pProcData[pos+1];
      SendDataSize = ProcDataSize - pos – 1;
      if ( SendDataSize < 0 ) SendDataSize = 0;
      CommProcRank = ProcRank + Mask
      pData = & pProcData[0];
      DataSize = pos + 1;
    }
    else { // high order bit = 1
      pSendData = & pProcData[0];
      SendDataSize = pos + 1;
      if ( SendDataSize > ProcDataSize ) SendDataSize = pos;
      CommProcRank = ProcRank – Mask
      pData = & pProcData[pos+1];
      DataSize = ProcDataSize - pos - 1;
      if ( DataSize < 0 ) DataSize = 0;
    }
    // Sending the sizes of the data block parts
    MPI_Sendrecv(&SendDataSize, 1, MPI_INT, CommProcRank, 0,
      &RecvDataSize, 1, MPI_INT, CommProcRank, 0, MPI_COMM_WORLD, &status);

    // Sending the data block parts
    pRecvData = new double[RecvDataSize];
    MPI_Sendrecv(pSendData, SendDataSize, MPI_DOUBLE,
      CommProcRank, 0, pRecvData, RecvDataSize, MPI_DOUBLE,
      CommProcRank, 0, MPI_COMM_WORLD, &status);

    // Data merge
    MergeDataSize = DataSize + RecvDataSize;
    pMergeData = new double[MergeDataSize];
    DataMerge(pMergeData, pMergeData, pData, DataSize,
              pRecvData, RecvDataSize);
    delete [] pProcData;
    delete [] pRecvData;
```

```
    pProcData = pMergeData;
    ProcDataSize = MergeDataSize;
  }
}
```

The function *LocalDataSort* sorts the data block on each processor using the sequential quick sort algorithm.

The function *PivotDistribution* determines the pivot element and sends its value to all the processors.

The function *GetProcDataDivisionPos* calculates the position of the data block partition with respect to the pivot element. The result of the function is the integer number, which determines the position of the element on the border of two blocks.

The function *DataMerge* merges the data parts into the sorted data block.

3. The function PivotDistribution. This function selects the pivot element and sends it to all the hypercube processors. As the data located on the processors have already been sorted, the pivot element is selected as the middle element of the data block.

```
// Determination of the pivot value and broadcast it to all the processors
void PivotDistribution (double *pProcData, int ProcDataSize, int Dim,
  int Mask, int Iter, double *pPivot) {
  MPI_Group WorldGroup;
  MPI_Group SubcubeGroup; // a group of processors – a subhypercube
  MPI_Comm  SubcubeComm;  // subhypercube communcator
  int j = 0;

  int  GroupNum = ProcNum /(int)pow(2, Dim-Iter);
  int *ProcRanks = new int [GroupNum];

  // Forming the list of ranks for the hypercube processes
  int StartProc = ProcRank – GroupNum;
  if (StartProc < 0 ) StartProc = 0;
  int EndProc = (ProcRank + GroupNum;
  if (EndProc > ProcNum ) EndProc = ProcNum;
  for (int proc = StartProc; proc < EndProc; proc++) {
    if ((ProcRank & Mask)>>(Iter) == (proc & Mask)>>(Iter)) {
      ProcRanks[j++] = proc;
    }
  }
  // Creating the communicator for the subhypercube processes
  MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);
  MPI_Group_incl(WorldGroup, GroupNum, ProcRanks, &SubcubeGroup);
  MPI_Comm_create(MPI_COMM_WORLD, SubcubeGroup, &SubcubeComm);

  // Selecting the pivot element and seding it to the subhypercube processes
  if (ProcRank == ProcRanks[0])
    *pPivot = pProcData[(ProcDataSize)/2];

  MPI_Bcast ( pPivot, 1, MPI_DOUBLE, 0, SubcubeComm );
  MPI_Group_free(&SubcubeGroup);
  MPI_Comm_free(&SubcubeComm);
  delete [] ProcRanks;
}
```

10.5.3.2. Computational Experiments Results

The computational experiments for estimating the efficiency of the parallel variant of the HyperQuickSort method were carried out under the same conditions as the experiments described previously (see 10.3.4).

The results of the computational experiments are given in Table 10.8. The experiments were carried out with the use of 2 and 4 processors. The time is given in seconds.

Table 10.8. The results of the computational experiments for the parallel HyperQuickSort algorithm

Number of elements	Sequential algorithm	Parallel algorithm			
		2 processors		4 processors	
		Time	Speedup	Time	Speedup

10,000	0.001422	0.001485	0.957576	0.002898	0.490683
20,000	0.002991	0.002180	1.372018	0.003770	0.793369
30,000	0.004612	0.003077	1.498863	0.004451	1.036172
40,000	0.006297	0.003859	1.631770	0.004721	1.333828
50,000	0.008014	0.005041	1.589764	0.005242	1.528806

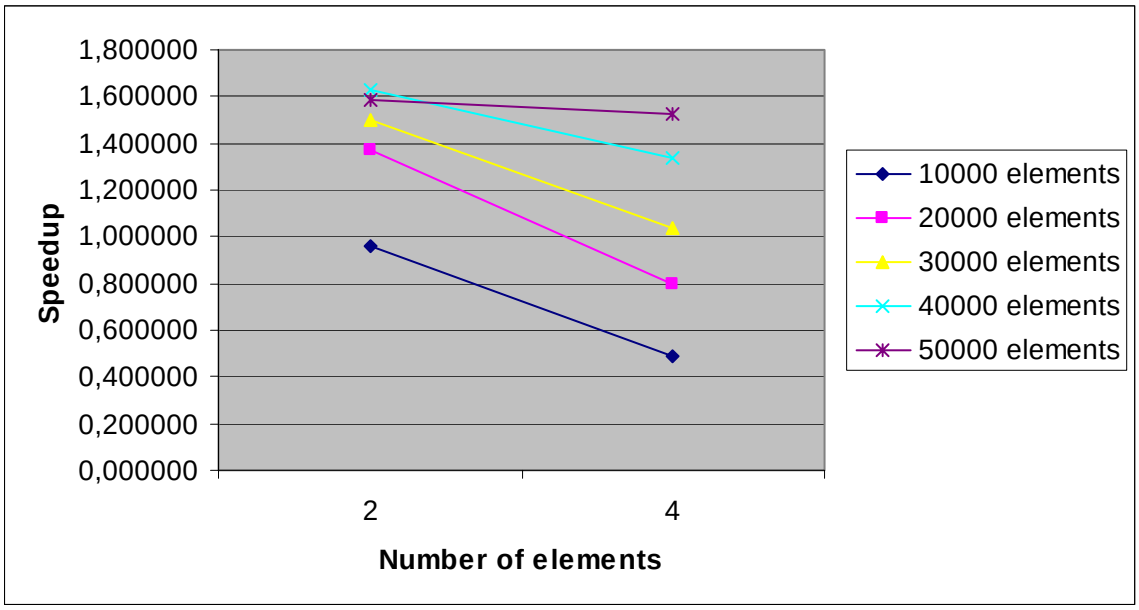

Figure 10.9. Speedup of the parallel HyperQuickSort algorithm

The comparison of the experimental execution time T_p^* and the theoretical estimation T_p from (10.12) is given in Table 10.9 and Figure 10.10.

Table 10.9. The comparison of the experimental and theoretical execution time for the parallel HyperQuickSort algorithm

Data size	Parallel algorithm			
	2 processors		4 processors	
	T_2	T_2^*	T_4	T_4^*
10,000	0.001281	0.001485	0.001735	0.002898
20,000	0.002265	0.002180	0.002322	0.003770
30,000	0.003289	0.003077	0.002928	0.004451
40,000	0.004338	0.003859	0.003547	0.004721
50,000	0.005407	0.005041	0.004176	0.005242

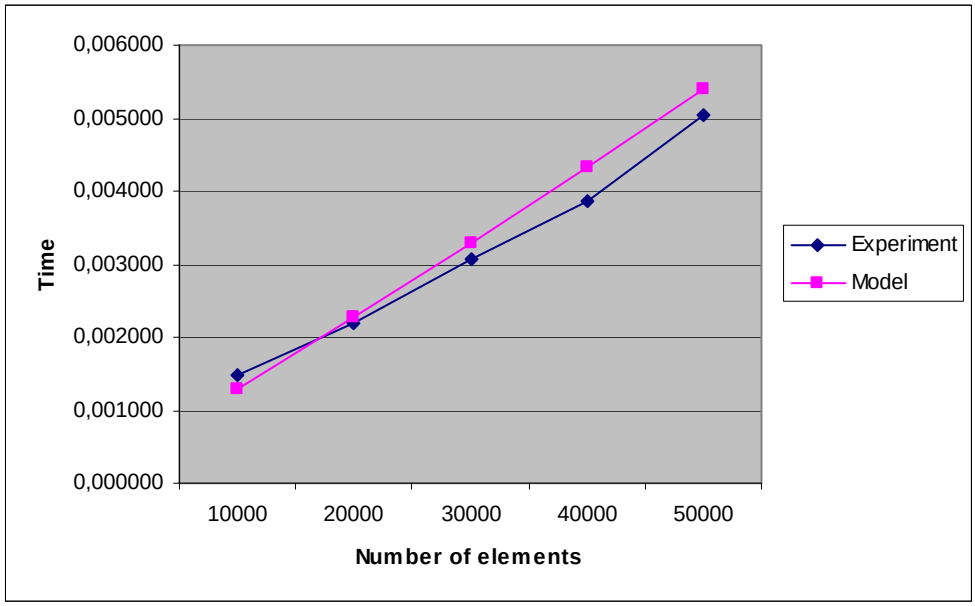

Figure 10.10. Experimental and theoretical execution time for 4 processors

10.5.4. The Parallel Sorting by Regular Sampling

10.5.4.1. Parallel Computational Scheme

The algorithm of *the Parallel Sorting by regular sampling* is also a generalization of the quick sort method (see, for instance, Quinn (2004)).

To sort data in accordance with this new variant of the quick sort algorithm the following four stages should be implemented:

- In *the first stage* of the algorithm the blocks located on the processors are sorted. This operation may be executed by each processor independently of the other processors by means of the original quick algorithm. Each processor then forms a set of elements of its blocks with the indices $0,\ m,\ 2m, \ldots, (p\text{-}1)m$, where $m = n/p^2$ (this set can be considered as *regular samples* of the processor data block);
- In *the second stage* of the algorithm all the data sets, which have been formed on the processors, are accumulated on a processor and are merged in a single sorted set. Then the values of this set with the indices

$$p + \lfloor p/2 \rfloor - 1,\quad 2p + \lfloor p/2 \rfloor - 1, \ldots, (p-1)p + \lfloor p/2 \rfloor$$

form a new set of the pivot elements, then this set is transmitted to all the processors being used. At the end of the stage each processor partitions its block into p parts using the obtained set of the pivot values;

- In *the third stage* of the algorithm each processor sends the selected parts of its block to all the other processors. It is done in accordance with the enumeration order – the part $j,\ 0 \le j < p$, of each block is transmitted to the processor j;
- In *the fourth stage* of the algorithm each processor merges p the obtained parts in a single sorted block.

After the termination of the stage 4 the initial data become sorted.

Figure 10.11 shows an example of data sorting by means of the algorithm, which is described above. It should be noted that the number of processors for the given algorithm may be arbitrary. In this example it is equal to 3.

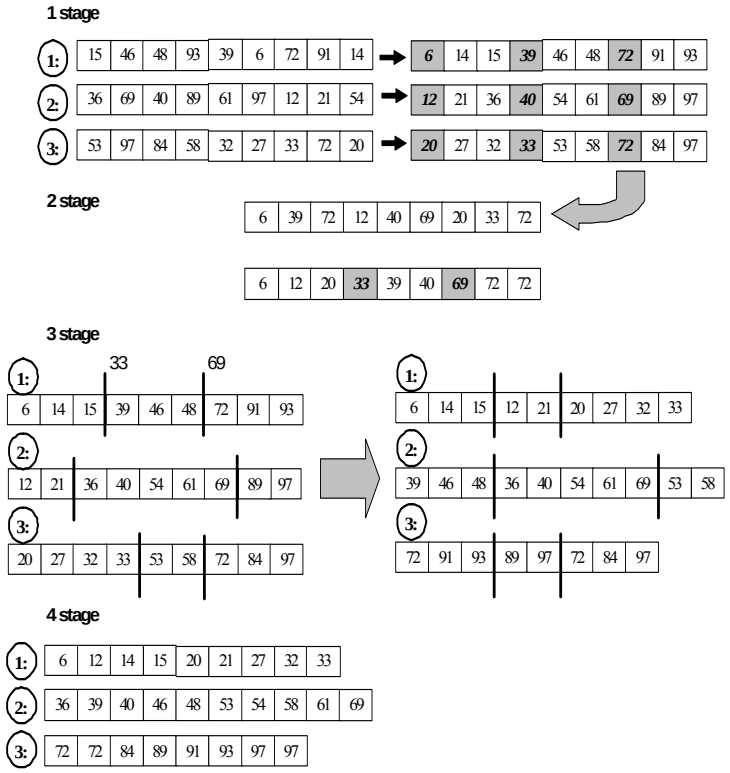

Figure 10.11. The example of executing the quick sorting algorithm by regular sampling for 3 processors

10.5.4.2. Efficiency Analysis

Let us estimate the complexity of this parallel method. Let n be the amount of the sorted data, p, $p<n$, denotes the number of the processors being used, and correspondingly n/p is the size of the data blocks on the processors.

During the first stage of the algorithm each processor sorts its data block by means of the quick sort method. Thus, the duration of the operations performed is equal to the following:

$$T_p^1 = (n/p)\log_2(n/p)\,\tau,$$

where τ is the execution time of the basic sorting operation.

During the second stage of the algorithm one of the processors accumulates the sets of p elements from all the other processors and merges the obtained data (the total number of the elements is equal to p^2), and forms the set of $p-1$ pivot elements. Then the processor transmits the set of pivot elements to the other processors. Taking into account all the above mentioned operations we may determine the duration of the second stage as follows:

$$T_p^2 = [\alpha \log_2 p + wp(p-1)/\beta] + [p^2 \log_2 p\,\tau] + [p\tau] + [\log_2 p\,(\alpha + wp/\beta)]$$

(the subexpressions in square brackets correspond to the four above mentioned operations); in this case, as previously, α is the latency, β is the network bandwidth, and w is the size of the set element in bytes.

During the third stage of the algorithm each processor divides its block with regards to the pivot elements into p parts (the total number of the operations for this purpose may be limited by the value n/p). Then all the processors transmit the formed parts of blocks to each other. The complexity estimation of this communications in case of the hypercube network topology was considered in Section 3. It was shown that the execution of this operation might be carried out in $log_2 p$ steps. Each processor at each step transmits and receives a message of $(n/p)/2$ elements. As a result, the complexity of the third stage may be estimated as follows:

$$T_p^3 = (n/p)\tau + \log_2 p(\alpha + w(n/2p)/\beta).$$

During the fourth stage each processors merges p sorted parts in a single sorted block. The estimation of complexity for the operations was carried out in the course of the consideration of the second stage. Thus, the duration of the merge procedure execution is as follows:

$$T_p^4 = (n/p)\log_2 p\,\tau\ .$$

With regard to all the obtained relations the total execution time for the parallel sorting by regular sampling may be estimated as follows:

$$T_p = (n/p)\log_2(n/p)\,\tau + (\alpha\log_2 p + wp(p-1)/\beta) + p^2\log_2 p\,\tau + (n/p)\tau + \\ + \log_2 p(\alpha + wp/\beta) + p\tau + \log_2 p(\alpha + w(n/2p)/\beta) + (n/p)\log_2 p\,\tau \quad . \qquad (10.13)$$

10.5.4.3. Computational Experiment Results

The computational experiments for estimating the efficiency of the parallel sorting by regular sampling were carried out under the same conditions as the experiments described previously (see 10.3.4).

The results of the computational experiments are given in Table 10.10. The experiments were carried out with the use of 2 and 4 processors. The time is given in seconds.

Table 10.10. The results of the computational experiments for the parallel sorting by regular sampling

Number of elements	Sequential algorithm	Parallel algorithm			
		2 processors		4 processors	
		Time	Speedup	Time	Speedup
10,000	0.001422	0.001513	0.939855	0.001166	1.219554
20,000	0.002991	0.002307	1.296489	0.002081	1.437290
30,000	0.004612	0.003168	1.455808	0.003099	1.488222
40,000	0.006297	0.004542	1.386394	0.003819	1.648861
50,000	0.008014	0.005503	1.456297	0.004370	1.833867

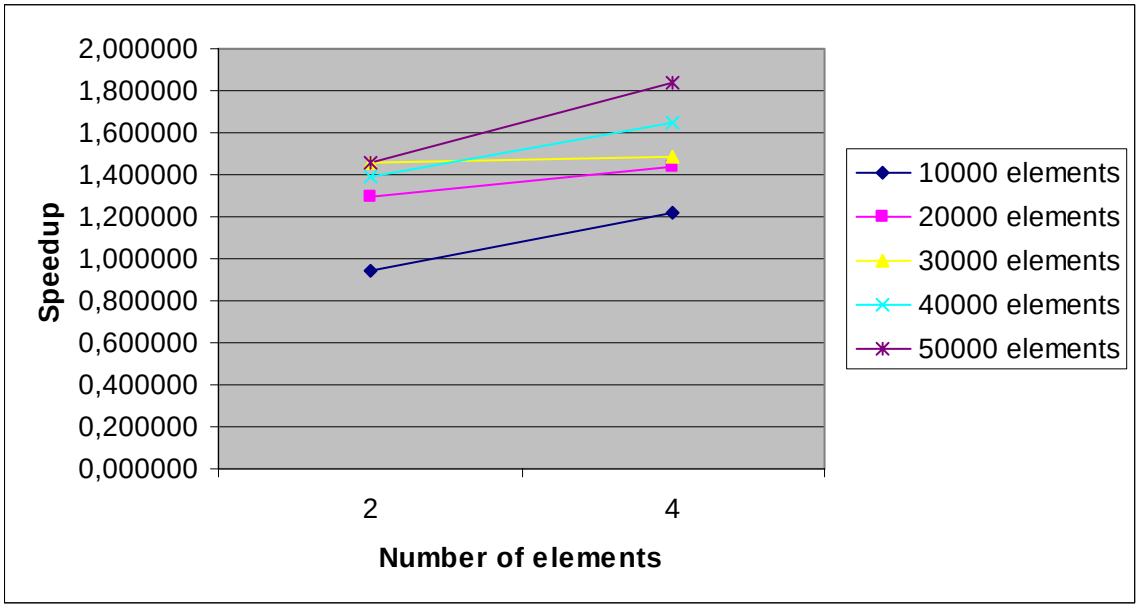

Figure 10.12. Speedup of the parallel sorting by regular sampling

The comparison of the experimental execution time T_p^* and the theoretical estimation T_p from (10.13) is given in Table 10.11 and Figure 10.13.

Table 10.11. The comparison of the experimental and theoretical execution time for the parallel sorting by regular sampling

Data size	Parallel algorithm			
	2 processors		4 processors	
	T_2	T_2^*	T_4	T_4^*
10,000	0.001533	0.001513	0.001762	0.001166
20,000	0.002569	0.002307	0.002375	0.002081
30,000	0.003645	0.003168	0.003007	0.003099
40,000	0.004747	0.004542	0.003652	0.003819

50,000	0.005867	0.005503	0.004307	0.004370

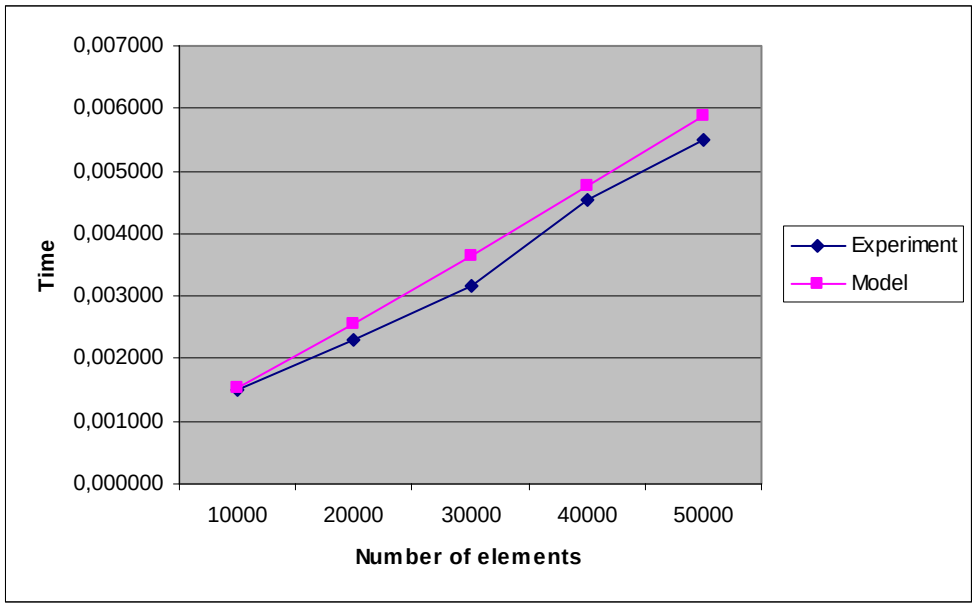

Figure 10.13. Experimental and theoretical execution time for 4 processors

10.6. Summary

The Section discusses the problem of data sorting, which is widespread in applications. For solving this problem three well-known sort methods are chosen in this Section. This is the bubble sort algorithm, the Shell sort algorithm and the quick sort algorithm. The Section focuses on the possible method of parallelizing the algorithms, the efficiency analysis and the comparison of the theoretical estimations to the results of the computational experiments performed.

In its original form *the bubble sort algorithm* (Subsection 10.3) is very hard to parallelize due to the sequential execution of the main iterations of the method. *The odd-even transposition method* is considered as a means of introducing parallelism. The essence of the generalized algorithm is that two different rules of executing the method iterations depending on the evenness of the sorting iteration number are introduced into the sort algorithm. The comparison of the pairs of values of the data being sorted on the iterations of the odd-even transposition method is independent and may be executed in parallel.

The parallel scheme for *the Shell algorithm* in case of the hypercube network topology is discussed in Subsection 10.4. The hypercube network topology makes possible to carry out data communications between the processors, which are located far from each other in case of linear enumeration. As a rule, such scheme of computations allows the decrease of the number of the executed sorting iterations.

Three schemes of parallelizing are given for *the quick sort algorithm* (Subsection 10.5). The first two schemes are also based on the hypercube network topology. At the beginning of the quick sort method the pivot processor choose in some way the pivot element and send it to all the hypercube processors. After obtaining the pivot value the processors subdivide their blocks. The parts of blocks are transmitted between the pairwise linked processors. After the execution of this iteration the initial hypercube appears to be subdivided into two hypercubes of smaller dimension. As a result, the above described computational scheme can be applied to these subhypercubes recursively.

Analyzing the computation efficiency in Subsection the special attention addresses on one of the main aspects in the execution of the quick sort algorithm. That is the adequate choice of the pivot element. The optimal case is when the value of the pivot element is chosen so that the data on the processors are partitioned into the parts of equal sizes. Generally, if the initial data is randomly generated, it is rather difficult to select such a value. The first scheme suggests choosing the pivot element as, for instance, the arithmetic mean of the elements on the pivot processor. The data on each processor in the second scheme is sorted preliminary so that the middle element of the data block can be chosen as the pivot value.

The third parallel scheme of the quick sorting algorithm is based on a multilevel scheme of forming a set of pivot elements. This approach may be applied for an arbitrary number of processors. In this case it is possible to avoid numerous data communications and to obtain a better balance of data distribution among the processors.

10.7. References

The methods of solving the problem of data sorting are widely discussed at present. A complete survey of sorting algorithms may be found in Knuth (1997), the work by Cormen et al. (2001) may be recommended as one of the latest editions.

The parallel variants of the bubble sort algorithms and the Shell algorithms are discussed in Kumar, et al. (1994).

The parallel schemes of the quick sort algorithm in case of the hypercube network topology are described in Kumar, et al. (1994) and Quinn (2004). The parallel sorting by regular sampling is discussed in Quinn (2004).

The work by Akl (1985) may be useful in the consideration of the parallel computation issues for data sorting.

10.8. Discussions

1. What is the statement of the data sorting problem?
2. Give several examples of data sorting algorithms. What is the computational complexity for each of the algorithms?
3. What is the basic operation for the problem of data sorting?
4. What is the essence of the parallel generalization of this basic operation?
5. What is the essence of the odd-even transposition algorithm?
6. Describe the parallel variant of the Shell algorithm. In what way does it differ from the odd-even transposition method?
7. What is essence of the parallel variant of the quick sort method?
8. What depends on the adequate choice of the pivot element for the parallel quick sort algorithm influence?
9. What methods may be suggested for choosing the pivot element?
10. To what topologies can the described algorithms be applied?
11. What is the essence of the parallel sorting by regular sampling?

10.9. Exercises

1. Develop the implementation of the parallel bubble sort algorithm. Carry out the necessary experiments. Formulate the theoretical estimations of parallel computation efficiency. Compare the obtained theoretical estimations with the results of the experiments.

2. Develop the implementation of the parallel quick sort algorithm using one of the described schemes. Determine the values of the latency, the network bandwidth and the execution time of the basic sorting operation. Determine the estimations of the speedup and efficiency characteristics for the method. Compare the obtained theoretical estimations with the results of the experiments.

3. Design a parallel computation scheme for the well-known merge sort algorithm (a detailed description of the method may be found, for instance, in Knuth (1997) or Cormen et al. (2001)). Develop the implementation of the developed algorithm and formulate the necessary theoretical estimations of the method complexity.

References

Akl, S. G. (1985). Parallel Sorting Algorithms. – Orlando, FL: Academic Press

Cormen, T.H., Leiserson, C. E. , Rivest, R. L. , Stein C. (2001). Introduction to Algorithms, 2nd Edition. - The MIT Press.

Knuth, D. E. (1997). The Art of Computer Programming. Volume 3: Sorting and Searching, second edition. – Reading, MA: Addison-Wesley.

Kumar V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc. (2nd edn., 2003)

Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.