DESIGN AND EVALUATION OF TWO PARALLEL SORTING ALGORITHMS BASED ON MPI TECHNOLOGY

Ioan Z. MIHU, Horia V. CAPRITA

"Lucian Blaga" University of Sibiu, Computer Science Department, str. Emil Cioran, nr. 4, Sibiu, 550025, ROMANIA
E-mail: ioan.z.mihu@ulbsibiu.ro, horia.caprita@ulbsibiu.ro

Abstract: The message-passing architectures consist of multiple computers interconnected through an interconnection network, communicating one with other by send-receive message functions and synchronized by barrier functions. At the moment there exist many libraries that provide a set of standardized functions for parallel programming like Message Passing Interface (MPI). MPI library allow the implementation of parallel algorithms on message-passing architectures. In this paper we propose two parallel sorting algorithms designed for message-passing architectures and implemented using MPI library: parallel Insertionsort and parallel Quicksort algorithms. We evaluate the performance of the proposed algorithms on three types of message passing architectures: linear array, two-dimensional mesh and hypercube. We evaluate the sorting time, the interprocesses communication time and the total processing time for each topology and we analyze the efficiency of the two parallel sorting algorithms related to the parallel system topology.

Keywords: message passing architecture, parallel programming, parallel algorithms, sorting algorithms

1. INTRODUCTION

The key issue of parallel architectures is to exploit the potential of parallelism in real applications in order to minimize the processing time. Once the parallel systems have been developed, the scientists have to design efficient parallel algorithms usually derived from the correspondent sequential algorithms. A parallel program consists (generally) of more processes that synchronize and communicate between each other. A very important issue is to obtain an optimum matching degree between the parallel algorithm and the topology of the parallel machine (Culler *et* Singh 1999). Moreover, if the real application doesn't contains a sufficient potential of parallelism, then the parallel derived algorithm will be inefficient and very expensive at the execution time. There are two classes of architectures that exploit coarse-grain parallelism: Shared-Memory Architectures and Message Passing Architectures. The former category includes the multiprocessor systems in which the processes synchronize and communicate through the shared variables stored on the common memory (fig. 1). The second class includes the multicomputer systems based on static or dynamic interconnection networks and communicating by sending data packets from the source node to the destination node through the network (fig. 2). The multicomputer systems can be emulated on local area networks (LANs) using MPI library. The communication process is implemented with send-receive message functions. Using a LAN the MPI environment can emulate any type of interconnection network topology (linear array, meshes, hypercube etc.). Based on MPI library we implemented and evaluated the parallel Insertionsort and parallel Quicksort algorithms on three types of message-passing architectures: linear arrays, two-dimensional mesh, hypercube.

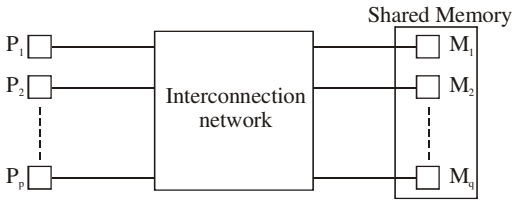

Fig. 1. Shared-Memory Architecture.

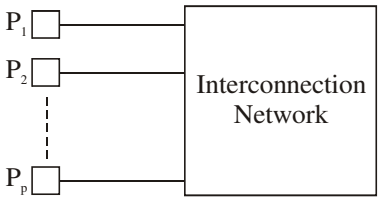

Fig. 2. Message-Passing architecture.

2. SIMULATED PARALLEL ARCHITECTURES

2.1. Linear array network

The linear array is a one-dimensional network in which N-nodes (processors) are connected by N-1 links in a line (fig. 3). Internal nodes have degree 2, and the terminal nodes have degree 1. The diameter is N-1, which is rather long for large N. The bisection is b=1. Linear arrays have the simplest connection topology, the structure is not symmetric and poses communication inefficiency when N becomes very large. This is because any data entering the network from one end must pass through all nodes in order to reach the other end of the network (Culler *et* Singh, 1999). However the linear array allows concurrent use of different sections (channels) of the structure by different source and destination pairs.

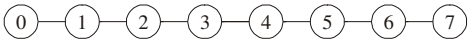

Fig. 3. Linear array network.

2.2. *Two-dimensional mesh network*

A two-dimensional mesh consists of $k_1 \times k_2$ nodes, where $k_i \geq 2$ denotes the number of nodes along dimension i. Figure 4 represents the two-dimensional mesh used in our experiments ($k_1=4$ and $k_2=2$). In this mesh network each node is connected to its north, south, east and west neighbors. In general, a node at row i and column j is connected to the nodes at locations (i-1, j), (i+1,j), (i, j-1) and (i, j+1). Therefore, the node degree is 4 (excepting the node on the edges and vertices) and the diameter of the mesh network is equal to the distance between nodes at opposite corners: $(k_1-1)+(k_2-1)$. The routing of data through a mesh network can be accomplished in a straightforward manner (Lee *et al*, 2001). The following simple routing algorithms routes a packet from source S to destination D in a n×m mesh network:

1. Compute the row distance R as: $R = \lfloor D/m \rfloor - \lfloor S/m \rfloor$
2. Compute the column distance C as: $C = D(\mathrm{mod}\, n) - S(\mathrm{mod}\, n)$
3. Add the values R and C to the packet header at the source nod
4. Starting from the source node, send a packet for R rows and then C columns

The values R and C determine the number of rows and columns that the packet needs to travel. The sign of the values R and C determines the direction of the message at each node. When R (C) is positive, the packet travels downward (right); otherwise, the packet travels upward (left). Each time that the packet travels from one node to the adjacent node downward, the value R is decremented by 1, and when it travels upward, R is incremented by 1. Once R becomes 0, the packet starts traveling in the horizontal direction. Each time that the packet travels from one node to the adjacent node in the right dimension, the value C is decremented by 1, and when it travels in the left direction, C is incremented by 1. When C becomes 0, the packet has arrived at the destination.

2.3. Hypercube network

A *n*-cube network, also called hypercube, consists of $N = 2^n$ nodes; *n* is called the dimension of the hypercube network. In a *n*-cube the nodes are considered the corners of an *n*-dimensional cube and the network connects each node to its *n* neighbors. The node degree of *n*-cube network is *n* and so does the network diameter. In fact, the node degree increases linearly with respect to the network dimension, making it difficult to consider the hypercube a scalable architecture. In a *n*-cube, individual nodes are uniquely identified by *n*-bit addresses ranging from 0 to N-1. Given a node with binary address *d*, this node is connected to all nodes whose binary addresses differ from d in exactly 1 bit. For example, in a 3-cube, in which there are 8 nodes, node 7 (address 111) is connected to node 6 (110), 5 (101) and 3 (011). Figure 5 demonstrates all the connections between the nodes. As can be seen in a 3-cube, two nodes are directly connected if their binary addresses differ by 1 bit. This method of connection is used to control the routing of data through the network in a simple manner. The following routing algorithm routes a packet from its source $S = (s_{n-1}...s_0)$ to destination $D = (d_{n-1}...d_0)$:

1. Tag $T = S \oplus D = t_{n-1}...t_0$ is added to the packet header at the source node ($\oplus$ denotes XOR function).
2. If $t_i \neq 0$ for some $0 \leq i \leq n-1$, then use

the i-th dimension link to send the packet to a new node with the same address as the current node except its i-th bit, and change t_i to 0 in the packet header.

3. Repeat step 2 until $t_i = 0$ for all $0 \le i \le n-1$

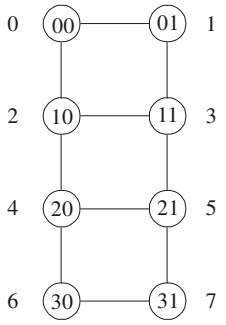

Fig. 4. Two-dimensional mesh network.

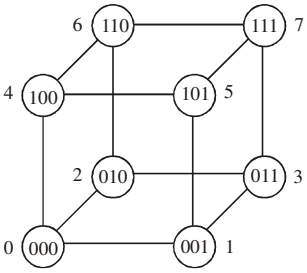

Fig. 5. 3-cube network.

3. THE PARALLEL INSERTION SORTING ALGORITHM

The parallel version of the algorithm consists of the following steps:

1. splitting the unsorted string in substrings of N/p elements (N – string dimension; p – number of processors);
2. sending the substrings from the master processor to others. Each processor will receive a substring to be sorted;
3. sorting the substrings by the insertion sorting method in all the processors. This will be done in parallel by all the processors
4. collecting the sorted substrings. This will be done according to the tree presented in figure 6.
5. repeat steps 3 and 4 until all the substrings will be collected in P0.
6. final sorting in P0

The sequential sorting algorithm complexity is $O(N^2)$. According to the implementation model presented in figure 7, the complexity of the parallel algorithm is:

$$O\left(\frac{N^2}{64}\right)+O\left(\frac{N^2}{64}\right)+O\left(\frac{N^2}{16}\right)+O\left(\frac{N^2}{4}\right)=O\left(\frac{11N^2}{32}\right)$$

$Level0$ $Level1$ $Level2$ $Level3$

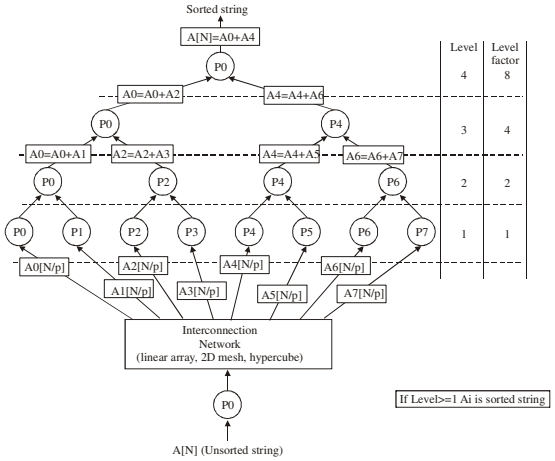

Fig. 6. Parallel sorting algorithm

4. THE PARALLEL QUICKSORT ALGORITHM

The sequential Quicksort algorithm is based on *divide and conquer* method. This method consists of two steps:

1. **Divide** the problem into smaller independent subproblems and solve these subproblems individually;
2. **Conquer**: solve the main problem by combining the solutions of the individual subproblems.

For the Quicksort algorithm the two steps consist in:

1. **Divide**: Partition function splits the string into two nonempty substrings (the elements positioned in the lower substring are all smaller than the elements of the upper substring);
2. **Conquer**: substrings are sorted by recursive calls to Quicksort function.

Our parallel version of the algorithm follows two phases:

Phase 1 – Divide and Conquer (figure 8a):

1. **Divide**: Find the pivot *m* (partition function partitions the string into two nonempty substrings). Starting from the root of the tree, on each level, the nodes implied in sorting will send the upper substring to the node placed on the next level. That is, on the last level, each processor will have a substring to be sorted;
2. **Conquer**: The substrings are sorted in parallel by recursive calls to sequential Quicksort algorithm. In this step all the processors runs the sequential algorithm.

Phase 2 – Collecting the sorted substrings:

In this phase every node store his sorted substring. Using a similar tree the P0 node will collect the sorted substrings from the others nodes in four steps. Therefore, through the concatenation of all these substrings, the node P0 obtains the sorted string on the last level. This will be done according to the tree presented in figure 8b.

```
Parallel_Quicksort (begin, limit, level)
{
if (level>1){
//The processors implied in sorting find the pivot m
        m=Seq_Quicksort_Partition(begin,limit);
//Sending the upper substring to the others processors
        m=Send_Upper_SubString(m,limit,level);
//Every processor will find, in function of current level, the tree //branch on which it works
        left=Find_Branch(level);
 if(left==1){
        Parallel_Quicksort(begin,m,level/2);
 }
 else if(left==0){
        Parallel_Quicksort(m+1,limit,level/2);
 }}
else{  //level=1, call Sequential_Quicksort
        Sequential_Quicksort(begin,limit);
}}
```

Fig. 7. Parallel Quicksort algorithm.

The complexity of the sequential Quicksort algorithm is O(n×log(n)). For the parallel quicksort algorithm, the covering of the sorting tree nodes can be made in parallel unlike in the sequential case, when the single processor covers only one node at a time. For the parallel version the complexity becomes O(n×log(n)/log(p)), where *p* is number of processors.

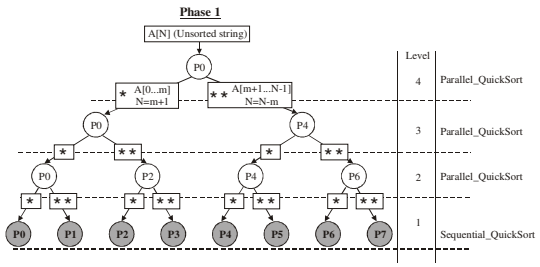

Fig. 8a. Parallel Quicksort algorithm – Phase 1.

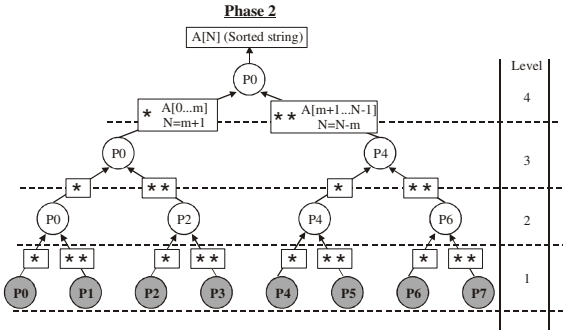

Fig. 8b. Parallel Quicksort algorithm – Phase 2.

5. EXPERIMENTAL RESULTS

In this section we evaluate the performances of the parallel Insertion sorting and parallel Quicksort algorithms on three parallel topologies: linear array, two-dimensional mesh and hypercube (3-cube). These topologies were emulated using MPI library on a LAN with 8 Intel Pentium IV (2,4GHz) / 256 MB RAM computers connected by a 100Mbs switch. The input data (the unsorted string) were obtained from an input file consisting of random values. Therefore, we had the same input data in all simulated architectures. Table 1 presents data input characteristics according to the simulated parallel topologies for the two algorithms.

Table 1. The input parameters and the simulated configurations

Interconnection network	Number of nodes	Number of string elements (N)	String dimension (N×4 Bytes)
Linear array	8	524,288	2MB
2-D mesh	8	1,048,576	4MB
Hypercube	8	2,097,152	8MB
		4,194,304	16MB

5.1. The parallel insertion sorting algorithm

This section presents the results for the parallel Insertion sorting algorithm in MPI. The execution/ sorting/communication times corresponding to all the three types of message-passing architectures are presented in tables 2, 3, 4 and 5. The "Other operations" field in these tables represents the time required for MPI environment initialization, data input reading and variables settings.

Table 2. Execution, sorting and communication time for 2MB string size (in seconds)

2MB string	**Total execution time**	**Sorting Time**	**Comm. time**	**Other operations**
Sequential	425.841	425.831	0.000	0.010
Linear Array	289.924	288.987	0.930	0.007
2-D Mesh	287.624	287.102	0.517	0.004
Hypercube	287.834	287.448	0.379	0.005

Table 3. Execution, sorting and communication time for 4MB string size (in seconds)

4MB string	**Total execution time**	**Sorting Time**	**Comm. time**	**Other operations**
Sequential	1714.627	1714.587	0.000	0.039
Linear Array	1172.237	1169.986	2.194	0.056
2-D Mesh	1170.226	1168.199	1.994	0.032
Hypercube	1171.065	1169.182	1.856	0.026

Table 4. Execution, sorting and communication time for 8MB string size (in seconds)

8MB string	Total execution time	Sorting Time	Comm. time	Other operations
Sequential	6889.789	6889.710	0.000	0.079
Linear Array	4884.873	4870.190	13.689	0.993
2-D Mesh	4964.767	4951.410	12.360	0.996
Hypercube	4712.252	4701.073	11.174	0.004

Table 5. Execution, sorting and communication time for 16MB string size (in seconds)

16MB string	Total execution time	Sorting Time	Comm. time	Other opera-tions
Sequential	27454.589	27454.439	0.000	0.150
Linear Array	19090.490	18995.740	94.591	0.155
2-D Mesh	18890.350	18889.740	91.602	0.605
Hypercube	18889.910	18801.250	88.649	0.004

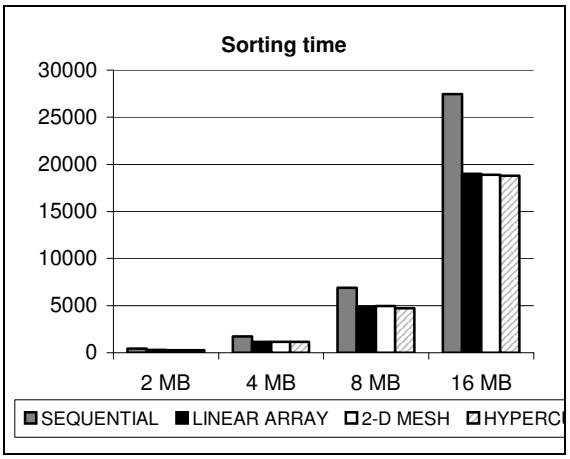

Fig. 9. Parallel versus sequential sorting time. (Insertion sorting algorithm)

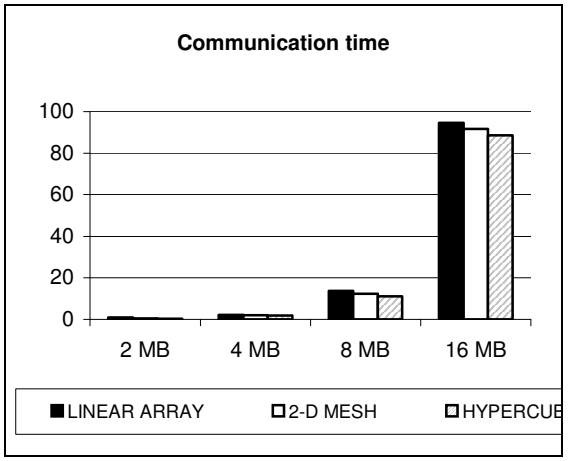

Fig. 10. Communication time for the three parallel architectures emulated on the 8 Pentium IV LAN. (Insertion sorting algorithm)

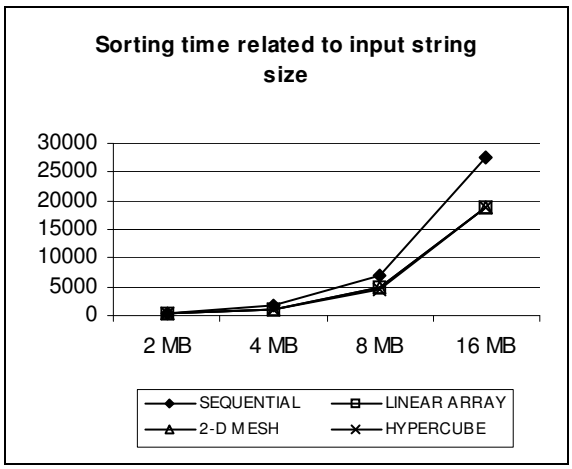

Fig. 11. Parallel versus sequential sorting time for 2MB, 4 MB, 8 MB and 16 MB input string size. (Insertion sorting algorithm)

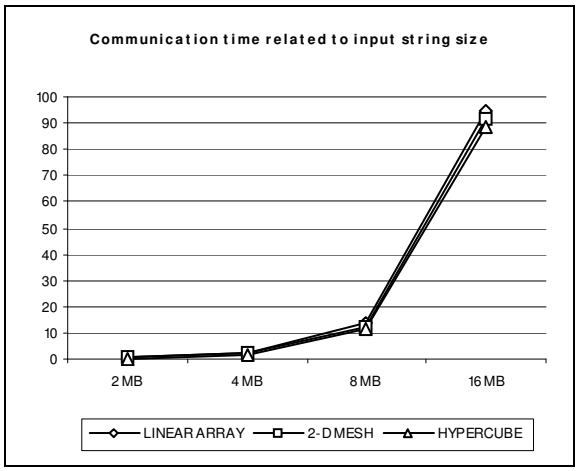

Fig. 12. Parallel communication time for 2MB, 4 MB, 8 MB and 16 MB input string size. (Insertion sorting algorithm)

5.2. The parallel quicksort algorithm

In this section we present the results for the parallel Quicksort algorithm. The execution, sorting and communication times corresponding to all the three types of message-passing architectures are presented in tables 6, 7, 8 and 9. The "Other operations" field in these tables represents the time required for MPI environment initialization, data input reading and variables settings.

Table 6. Execution, sorting and communication time for 2MB string size (in seconds)

2MB string	Total execution time	Sorting Time	Comm. time	Other operations
Sequential	0.301	0.283	0.000	0.018
Linear Array	2.682	0.100	2.573	0.008
2-D Mesh	1.779	0.101	1.567	0.011
Hypercube	1.408	0.101	1.009	0.013

Table 7. Execution, sorting and communication time for 4MB string size (in seconds)

4MB string	Total execution time	Sorting Time	Comm. time	Other operations
Sequential	0.570	0.570	0.000	0.036
Linear Array	6.578	0.238	6.181	0.159
2-D Mesh	4.487	0.246	3.743	0.162
Hypercube	3.675	0.241	2.401	0.162

Table 8. Execution, sorting and communication time for 8MB string size (in seconds)

8MB string	Total execution time	Sorting Time	Comm. time	Other operations
Sequential	1.342	1.270	0.000	0.071
Linear Array	11.853	0.575	11.230	0.047
2-D Mesh	8.705	0.582	6.185	0.048
Hypercube	7.221	0.606	4.233	0.047

Table 9. Execution, sorting and communication time for 16MB string size (in seconds)

16MB string	Total execution time	Sorting Time	Comm. time	Other operations
Sequential	2.403	2.403	0.000	0.150
Linear Array	27.001	0.799	26.070	0.132
2-D Mesh	17.686	0.789	16.412	0.134
Hypercube	13.808	0.780	12.056	0.138

The sequential sorting time and the parallel sorting time (for the three parallel emulated topologies) are represented in figure 13 and 15. The sequential sorting time corresponds to the sequential version of the Quicksort algorithm implemented on a single Pentium IV processor system. The parallel sorting time corresponds to the parallel version of the Quicksort algorithm implemented on the three parallel architectures (linear array, two-dimensional mesh and hypercube) emulated on the 8 Pentium IV LAN.

How we expected, the parallel sorting time is practically independent of the parallel system topology and is 2.80 (for 2 MB string size) / 2.36 (4 MB) / 2.16 (8 MB) / 3.04 (16 MB) times smaller than sequential sorting time. There are many difficulties for sorting time evaluation in a multitasking environment. Despite of the mentioned difficulties, the average speed-up is 2.59 which is not very far from theoretical speed-up of log(p) = log(8) = 3; this confirms the efficiency of our parallel Quicksort algorithm.

On the contrary, the communication time is strongly dependent on the parallel system topology (fig. 14 and 16). The communication time is rather big because of the fact that the three parallel topologies were emulated on a LAN. Relevant are not the absolute values of the communication time but the comparative values of this time. The average communication time for 2-D mesh topology (hypercube) is only 59.88% (40.51%) of average communication time for linear array.

As the node degree increases and the parallel architecture topology matches better the communication pattern implemented in the parallel algorithm, as the communication time decreases and the parallel application becomes more efficient. This is confirmed by the communication time measured on each parallel system topology (fig. 14 and 16).

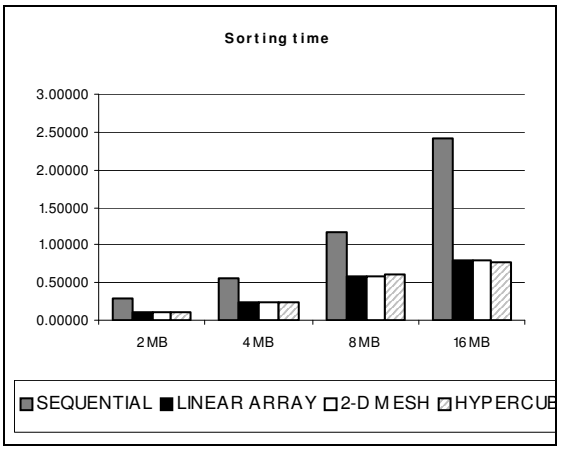

Fig. 13. Parallel versus sequential sorting time. (Quicksort algorithm)

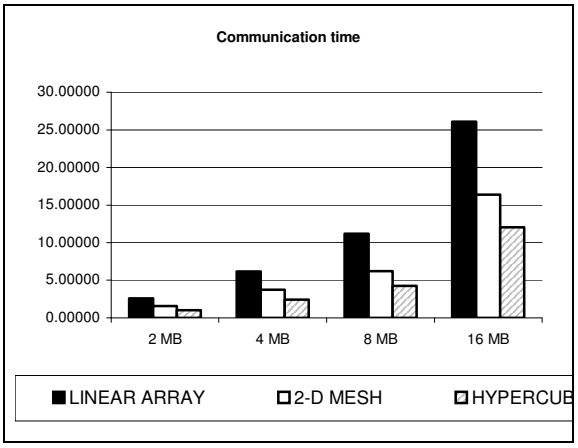

Fig. 14. Communication time for the three parallel architectures emulated on the 8 Pentium IV LAN. (Quicksort algorithm)

6. CONCLUSIONS

From our experiments the following conclusions can be drawn:

- The efficiency of parallel application depends on both parallel algorithm efficiency (see parallel Quicksort related to parallel Insertion sorting) and parallel machine topology;
- In order to obtain a global optimization (minimal execution time) a perfect match must be reached

between the parallel algorithm and the parallel machine topology;

- The efficiency of the parallel application increase as the amount of data processed growth;
- The communication time is strongly related to the architecture topology. Even if in absolute values the communication times in our experiments are very high because of the physical support used, by referring to the relative values the conclusions become confident and not dependent on physical platform.

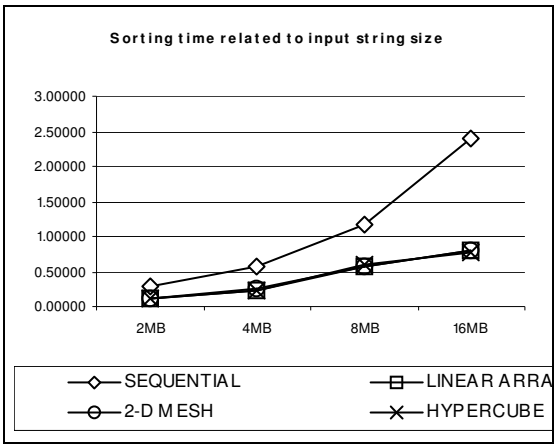

Fig. 15. Parallel versus sequential sorting time for 2MB, 4 MB, 8 MB and 16 MB input string size. (Quicksort algorithm)

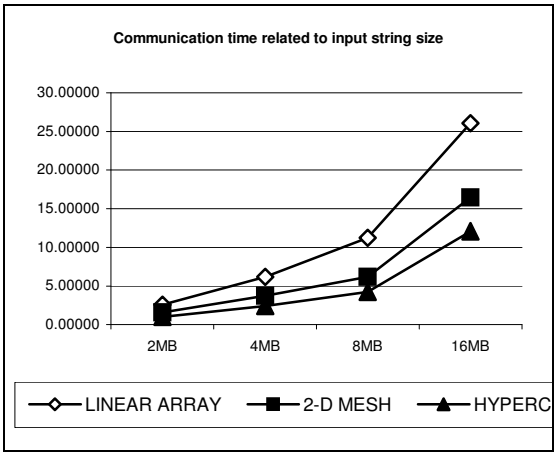

Fig. 16. Parallel communication time for 2MB, 4 MB, 8 MB and 16 MB input string size. (Quicksort algorithm)

Let K be the ratio between the sorting times for parallel Insertion sorting algorithm and parallel Quicksort algorithm respectively. In fig. 17 we present this K ratio related to the input string size. As can be seen the efficiency of parallel Quicksort algorithm versus parallel Insertion sorting algorithm increase as the amount of input data increase.

Designing a parallel application is a complex task; an optimal parallel algorithm doesn't always produce the best results. In all the parallel applications the algorithm, the amount of data to be processed and the parallel machine characteristics (number of processors, the interconnection network topology, the intercommunication functions and protocols) have to be analyzed and correlated.

Therefore, the building process of the parallel applications is a difficult task: there are many successive steps and the decisions taken in every step will affect the following steps and, at the end, the global performances of the parallel application.

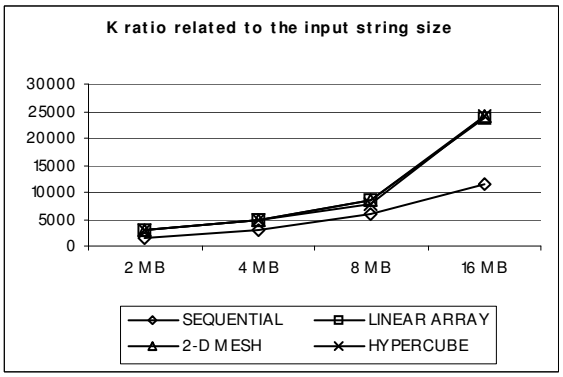

Fig. 17. K ratio for 2 MB, 4 MB, 8 MB and 16 MB input string size.

REFERENCES

G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus, *Supercomputing 2001*. IEEE, Denver, 2001.

F. Bernman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, L. T. D. Reed, and R. Wolski. The GrADS project: Software support for high-level grid application development, 27. August 2001.

D.E. Culler, J.P. Singh, Parallel computer architecture. A hardware/software approach, Morgan-Kaufmann Publishers, San Francisco, 1999.

E. Gabriel, M. Resch, and R. Ruhle. Implementing MPI with optimized algorithms for metacomputing, *Message Passing Interface Developer's and Users Conference (MPIDC'99)*, pages 31–41, Atlanta, March 10-12 1999.

C. Lee, S. Matsuoka, D. Talia, A. Sussmann, M. Muller, G. Allen, and J. Saltz. A Grid programming primer. Global Grid Forum, August 2001.

S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Performance modelling for self-adapting collective communications for MPI, *LACSI Symphosium*. Springer, Eldorado Hotel, Santa Fe, NM, Oct. 15-18 2001.