Parallel Sorting with Dynamic Load-Balancing in Stochastic Environments

MOHSEN LESANI, AMIR MOGHIMI, ALI AKHAVAN BITAGHSIR, NASSER YAZDANI
Electrical and Computer Engineering Department, Faculty of Engineering,
University of Tehran
P.O BOX : 14395/515, Tehran
IRAN
{m.lesani, a.moghimi, a.akhavan}@ece.ut.ac.ir
n.yazdani@ut.ac.ir

Abstract: Parallel algorithms and specifically parallelism in sorting algorithms as a symbolic application have been intensively studied. Some algorithms are available for parallel sorting by dedicated or homogenous processors. Dedicated processors for sorting are rarely a practice of real world. This paper introduces an algorithm for parallel sorting on processors that are not only utilized for sorting but also may run a multitasking OS and perform other tasks concurrently. This multitasking leads to stochastic behaviors and virtual speeds of the processors felt by the processes including the sorter processes. An agent framework is used to simulate a multiprocessor environment. The results show that the proposed algorithm performs better load balancing between processors than previous algorithms in stochastic speed-variant environments.

Key-words: Parallel sorting, Dynamic load-balancing, Overpartitioning, Stochastic environments

1 Introduction

Sorting is known to be a fundamental representative application in computer science [1]. Several research-intensive parallel algorithms for sorting have been introduced since Batcher proposed the bitonic-sorting network [2]. Although theoretical methods have been designed for parallel sorting algorithms [5, 6, 8, 9, 10, 11, 12, 13], experiential efforts [14, 16, 17, 18, 19, 20, 21], started only in the late 1980's, when multiprocessor computers became widely available.

Usually in a regular parallel sorting algorithm, each processor holds a portion of the list of n elements for sorting. In a merge-based approach, at the first level, each processor sorts the portion dedicated to it by some sequential sorting algorithm, and finally all the sorted portions are exchanged among the processors and merged in a distributed manner. Instead, in a quicksort-based approach, the main unsorted list is partitioned into a number of smaller sublists, defined by some previously selected pivots, and then, the processors employ a sequential sorting algorithm to sort the sublists for which they are dedicated to. The act of merging the sorted portions from different processors in the merge-based method and the act of partitioning into sublists in the quicksort-based method can be done either by one step or several steps. Based on this parameter, the present parallel sorting algorithms can therefore be classified as *single-step* or *multi-step* (Table 1).

Table 1: Categorization of present Parallel Sorting Algorithms

$single - step$ merge-based	PSRS [22, 23], quickmerge [19]
$multiple - step$ merge-based	bitonic sort [2], parallel merge sort [24, 10], smooth sort [25], N&S sort [26], column sort [8], T& B sort [27], snakesort [28]
$single - step$ quicksort-based	parallel quicksort [29, 30] , parallel sample sort [31], Overpartitioning [7]
$multi - step$ quicksort-based	flashsort [9], B-flashsort [32], hyperquicksort [19, 18], K& K sort [33]

So far, the parallel sorting algorithms' performance was mainly considered to be dominated by two parameters : (I) communication cost & (II) degree of equal load balancing. Accordingly, single-step algorithms incur low communication cost, because each element is moved at most once in these algorithms; However, single-step algorithms usually have poor load balancing, because it is difficult to deduce nearly equal sized sublists in the absence of enough global information. Besides, single-step algorithms cannot rectify the sublists once they are arrived. On the other hand, multi-step algorithms usually incur higher communication cost because elements can be moved many

times between the processors.

The single-step algorithms can perform more efficiently if an impressive method to balance the load can be exploited. Efficient methods have been developed for this purpose. In 1992, *Parallel Sorting by Regular Sampling* (PSRS) has been proposed by Shi & Schaeffer [22, 23]. After a while, *Parallel Sorting by Overpartitioning* (PSOP) was introduced by Li & Sevcik [7], from which our method is inspired. The second method is proved experimentally to work better than the first one in equal load balancing between the processors. These single-step parallel algorithms, in turn, propose mathematical methods for load balancing between processors with high probability.

Typically, in the mentioned algorithms and majority of other similar algorithms, the speed of each processor is considered to be the same over the sorting time period. In other words, it is assumed that each processor, is only dedicated to its associated sorting task, which is not necessarily true in real world. The intrinsic speed of different processors may not be the same, which implies different task assignments during sorting. Besides, the process resources (processors), are often dedicated to different tasks and processes, using multi-tasking mechanisms, leading to virtual variant speeds felt by processes. In another case, a processor's speed may decrease suddenly during the operation, because of some fault occurred in its structure (e.g., High temperature, etc). Accordingly, we assume that the behavior of a specific processor is changing over time, stochastically; rising or falling by an unpredictable speed function. Therefore, the overall performance of the algorithm is not governed by equal load balancing between processors. Rather, it is more efficient to assign longer tasks to faster processors. In this paper, we propose a new approach to single-step parallel sorting in such environments. The main idea behind the algorithm, is to set the length of the submitted task to each processor, based on its current speed and also its history, in comparison to the other processors.

In the first Section, the parallel sorting algorithm by overpartitioning is described. In section 3, some issues about efficient load balancing in stochastic environments are described. Afterwards, we introduce our approach in Section 4, showing that our algorithm performs logically better in stochastic environments. For the implementation phase, a framework (JADE) [4] has been used for developing FIPA-compliant [3] sorter agents, simulating the whole environment that is presented in Section 5; Also the simulation results are presented in this section. Finally the conclusions will come in section 6.

2 The Parallel Sorting by Overpartitioing

There are four main phases for a canonical single-step parallel sort on p processors. The **P**arallel **S**orting by **O**ver**P**artitioning (**PSOP**) approach follows the four phase canonical form of single-step sort. These four steps are as follows [7] :

1. Initially processor i has l_i, a portion of size n/p of the unsorted list l.
2. *Selecting Pivots*. A sample of pks candidates are randomly picked from the list, where s is the oversampling ratio which improves our sampling, and k is the overpartitioning ratio which defines the number of partitions the list will be divided to. Each processor picks sk candidates and passes them to a designated processor. These candidates are sorted, and then $pk-1$ pivots are selected by taking $s^{th}, 2s^{th}, \ldots, (pk-1)^{th}$ candidates from the sample. The selected pivots, $d_1, d_2, \ldots, d_{pk-1}$ are made available to all the processors.
3. *Partitioning*. Since the pivots have been sorted, each processor performs binary partitioning on its local portion. Processor i first used the middle pivot, $d_{(pk-1)/2}$, to partition l_i into l_i^{left} and l_i^{right}, then recursively uses the left pivots to partition l_i^{left} and the right pivots to partition l_i^{right} until all l_{ij} are identified where l_{ij} is the j^{th} part of the i^{th} processor's portion. A sublist S_j is the union of l_{ij} with i ranging over all processors :
$$S_j = \cup_{i=1}^{p} l_{ij} \tag{1}$$
4. *Building a task queue and sorting sublists*. Let $T(S_j)$ denote the task of sorting S_j. The size of each sublist can be computed :
$$|S_j| = \sum_{i=1}^{p} |l_{ij}| \tag{2}$$
Also, the starting position of sublist S_j in the final sorted array can be calculated :
$$\sigma_j = 1 + \sum_{h=1}^{j-1} |S_h| \tag{3}$$
A task queue is built with the tasks ordered from the largest sublists size to smallest. Each processor repeatedly takes one task, $T(S_j)$, at a time from the queue. It processes the task by (a) copying the p parts of the sublist, $< l_{1j}, \ldots, l_{pj} >$, into the final array at positions σ_j to $\sigma_j + |S_j| - 1$, and (b) applying a sequential sort to the elements in that range. This process continues, until the task queue is empty.

These four steps define a general framework for parallel sorting algorithms with overpartitioning, where any

sequential sorting algorithm can be used by each processor in the last phase. The oversampling ratio governs the variance of sublists sizes. A larger oversampling ratio produces sublists with a smaller variability in size.

3 Load Balancing in Stochastic Environments

As mentioned before, load balancing is highly effective in single-step parallel sorting algorithms. Now suppose that the processors' speed are not the same. Besides, suppose that each processor's speed is varying over time, due to several parameters. For example the processor, may receive several tasks simultaneously from different sources, resulting a lower speed for processes, at least for a period of time. The other case happens when a processor faces a failure in its structure, which will decrease its speed significantly. In such cases, equal load balancing is definitely not the best way of task assignment to processors. It is trivial to show that in the best case, the faster processors should receive more tasks in a specific time period. In other words, the utilization of the system resources will be maximized, only when processors receive tasks relative to their actual speed.

The other important concern is the variation of each processor's speed over its operation time period. Suppose a processor is assigned a list l to partition, with length $Size(l)$; The processor starts to process the list, but suddenly the processor's speed significantly decreases, due to different parameters mentioned before. Now the processor will process the rest of list l in a longer time period, resulting into a delay in the system. The reverse case may also occur. namely, when the processor's speed significantly increases during the dedicated process. In this case, the system must be able to assign the processor further tasks, to utilize its increased speed well or its computation ability will be wasted.

In the PSOP algorithm, these realistic cases are not handled. We propose another approach to parallel sorting, in which the varying behavior of processors (System Resources), are handled wisely. The main idea behind our modification to PSOP algorithm is to assign the tasks to processors, based on the system's *cognition* over each specific processor. The system will evaluate this insight over the processor, by means of the processor's "momentary speed" and also its "speed history". The algorithm is described in detail, in the next section.

4 The Parallel Sorting algorithm in Stochastic Environments

In this section, we present the PSSE approach for parallel sorting. As mentioned before the PSSE approach tries to handle time-variant behaviors of processors in dynamic environments.

The PSSE approach follows the phases in overpartitioning method, respectively. The algorithm phases are described in detail as follows :

1. *Selecting pivots.* Initially, a designated processor picks sq equally spaced candidates starting from the beginning of the list l where s is oversampling ratio, and $q = pk$ where k is the overpartitioning ratio and p is the number of sorter processors. These candidates are sorted, and then $pk - 1$ pivots are selected by taking $s^{th}, 2s^{th}, \ldots, (pk-1)^{th}$ candidates from the candidates. The selected pivots are made available to all the processors.

2. *Task Assignment and Partitioning.* In this phase the designated processor in the previous phase, starts sending portions of l sequentially to other processors for partitioning. Starting sequentially from the beginning of list l, suppose that the designated processor has sent n numbers to the processors for partitioning so far (at first $n = 0$). At this time, suppose that processor P is idle. Also assume that the designated processor gets the speed of P, V_P, by some mechanism. This speed is calculated iteratively as follows:

$$V_P(0) = V_{P_{Current}} \quad (4)$$

$$V_P(n) = \alpha * V_P(n-1) + (1-\alpha) * V_{P_{Current}} \quad (5)$$

where $V_{P_{Current}}$ is the current transient speed of the processor P and $0 \leq \alpha \leq 1$ is the trust factor in speed history of processor P while $(1-\alpha)$ expresses the importance associated to the processor's momentary speed. $V_P(i)$, $1 \leq i$ is calculated iteratively at fixed time intervals.

Now, having the calculated processor speed, the designated processor will assign the next portion of elements to processor P with length L_P for partitioning where :

$$L_P = \frac{V_P}{V_{Max}} * R \quad (6)$$

V_{Max} is the maximum momentary speed of the whole processors (whether idle or not), and R is a constant which impose a restriction on the total number of elements passed to a processor each time $(R << Size(l))$. The portion starts from the $(n+1)^{th}$ element in l.

By applying this algorithm, because the portions sizes assigned to processors are computed according to processor speeds it is expected that processors finish their tasks simultaneously and another portions be assigned to them according to the newly computed processor speeds. So a processor may receive several portions for partitioning during this phase. But If a processor finishes its assigned task sooner than the expected time as a result of its speed increase then the designated processor would give it another portion. It is because the designated processor does not distribute all elements of the list l at first in contrast to PSOP. In PSOP the list l is partitioned into P equal parts and distributed among the processors at first; So if a processor finishes its task sooner, the designated processor would not have any task to assign to it. Thus, The computation ability of a fast processor is wasted this way. This inelegance is recovered in PSSE algorithm since it sends the list gradually in portions.

According to the fact that the pivots have been sorted previously, each processor performs binary partitioning on its portion. Processor i first uses the middle pivot, $d_{(pk-1)/2}$, to partition l_i into l_i^{left} and l_i^{right}, then recursively used the left pivots to partition l_i^{left} and the right pivots to partition l_i^{right} until all l_{ij} are identified. Each processor, will then send back l_{ij}s to the designated processor. A sublist S_j is the union of l_{kij} with i ranging over all processors and k ranging over all the portions assigned to processor i.

$$S_j = \cup_{i=1,k}^{p} l_{kij} \tag{7}$$

The designated processor collects each sublist gradually until the last portion of the list is returned partitioned.

3. *Building a task queue and sorting sublists.* Similar to the fourth phase of PSOP, in this phase Let $T(S_j)$ denote the task of sorting S_j. The size of each sublist can be computed :

$$|S_j| = \sum_{i=1,k}^{p} |l_{kij}| \tag{8}$$

Where k is ranging over all portions assigned to processor i. Also, the starting position of sublist S_j in the final sorted array can be calculated :

$$\sigma_j = 1 + \sum_{h=1}^{j-1} |S_h| \tag{9}$$

A task queue is built with the tasks ordered from the largest sublists size to smallest. The processors are ordered by their speeds. The tasks are assigned to the corresponding processors meaning that fastest processor is assigned the longest task $T(S_j)$ in the queue and so on. Each processor processes the task by (a) copying all parts of the sublist S_j, into the final array at positions σ_j to $\sigma_j + |S_j| - 1$, and (b) applying a sequential sort to the elements in that range. When a processor finished its task, another task is given to it from the task queue. This process continues, until the task queue is empty. The list is sorted now.

The pivots are selected centrally in the designated processor in this algorithm. Other processors would not contribute in this phase. The reason behind it is that the PSSE algorithm main idea is to distribute the portions gradually in the partitioning phase. Selecting the pivots before the processors can partition the portions is necessary and selecting the pivots should be done from all the list elements. This means that the other processors can only contribute in selecting the sample pivots only on the first portion before they start to partition and selecting sample pivots from the remaining part of the list should be done by the designated processor. This little parallelism is ignored and the whole job of selecting the pivots is done by the designated processor itself.

5 Implementation

For the implementation phase, we exploited a software framework for simulating a multiprocessor system. For this purpose, we've used **J**ava **A**gent **DE**velopment Framework (**JADE**) [4]. JADE (Java Agent DEvelopment Framework) is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middle-ware that claims to comply with the FIPA [3] specifications and a set of tools that support the debugging and deployment phase. All agent communication is performed through message passing, where FIPA ACL (Agent Communication Language) is the language for message representation. The **F**oundation for **I**ntelligent **P**hysical **A**gents (FIPA) was

formed in 1996 to produce software standards for heterogeneous and interacting agents and agent-based systems. FIPA specifications represent a collection of standards which are intended to promote the interoperation of heterogeneous agents and the services that they can represent.

An agent is an autonomous entity that has some defined behaviors. Agents can communicate with each other through messages. An Agent lives for a period of time and can terminate or die after a while. A processor can be viewed as an agent that has its own local memory, processing ability and tasks to run. An agent is run as a thread on a platform. The speed of this thread can be considered as the speed of the processor that is modelled as an agent. As the agent runs more behaviors, its speed will decrease because the behaviors are procedures that are interleaved in a software thread. This decrease of speed is the model for the decrease of speed that the processor causes to processes when many processes are run concurrently by a multitasking strategy on it. Thus the agent thread can be viewed as the processor and the agent behaviors can be viewed as processes on the processor. The agent communication messages can be considered as data that is travelling between processors in the system bus and memory. As the agents are run on a single platform in this implementation the communication cost is negligible.

A sorter agent is implemented to play the role of a processor. It can act both as a sorting manager and sorting client at the same time. There are two main behaviors implemented for this purpose: *manager behavior* and *client behavior*. The manager behavior first selects pivots and then sends them to the sorting clients through a message with a *Request* performative act [3]. Then it waits for the acknowledge message from the clients. For every acknowledge message received from clients, it sends the partition share of the agent, its length computed in the PSSE algorithm. Afterwards, again it waits for the agents to send acknowledge message of receiving the pivots or to send the partitioned numbers given to them. After all of the numbers have been partitioned by the client agents, the manager distributes partitions among clients to sort them according to their speeds. There is also another behavior inside manager responsible for receiving updated client speeds from the directory facilitator *(DF)* [3].

The client behavior waits for the pivots message from the manager behavior from the beginning and sends an acknowledge message as soon as receiving them. The client behavior sends messages with the *Inform* performative act [3]. This behavior also waits for receiving partition shares and sort shares and replies the results with an *Inform* message. There is also another behavior in the agent with the responsibility to update the speed of the agent in the directory facilitator *(DF)* regularly. The speed is evaluated according to the method proposed in the PSSE algorithm.

For the first step of simulation, a non-stochastic environment is considered and the two algorithms of PSOP and PSSE are compared with each other. In this environment, agents are dedicated to sorting task and also their speeds where nearly invariant during the simulation. As it is shown in Fig. 1, PSOP works better than PSSE for every value of R ranging from 5000 to 25000. This happens because of PSOP is well-designed for this kind of environment, while PSSE is designed to handle more dynamic environments by means of more communication cost. This communication cost decreases the relative performance of PSSE, compared to PSOP, in non-stochastic environments. As R increases PSSE becomes more similar to PSOP, whereas for lower values of R the communication cost significantly decreases PSSE's performance.

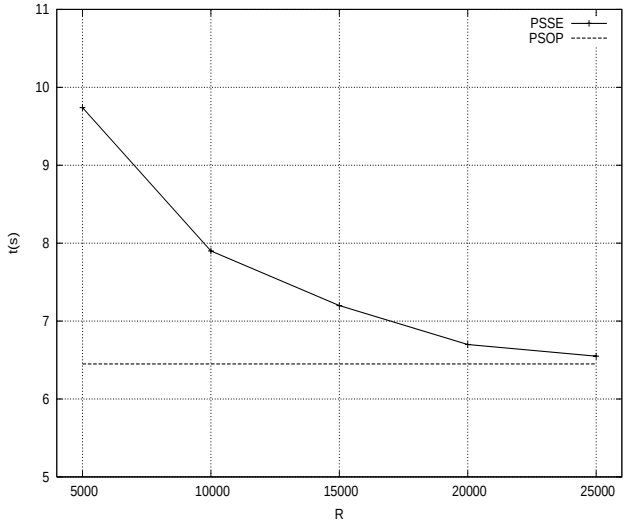

Figure 1: Comparison between PSOP and PSSE in a Non-Stochastic Environment

In the second step of simulation, a stochastic environment has been generated by deliberately injecting some tasks into specific agents during the simulation. In the comparison of PSOP and PSSE (Fig. 2), we observe that PSSE's performance dominates PSOP's performance significantly specially as R decreases. This is because for high variations of agents' speeds, the sorter manager can gradually re-balance the partition load between agents due to their new speeds. There is an exception for very low values of R for which the performance of PSSE decreases rather than increasing, because the communication cost increases considerably. This introduces a tradeoff between R and the communication load in the system.

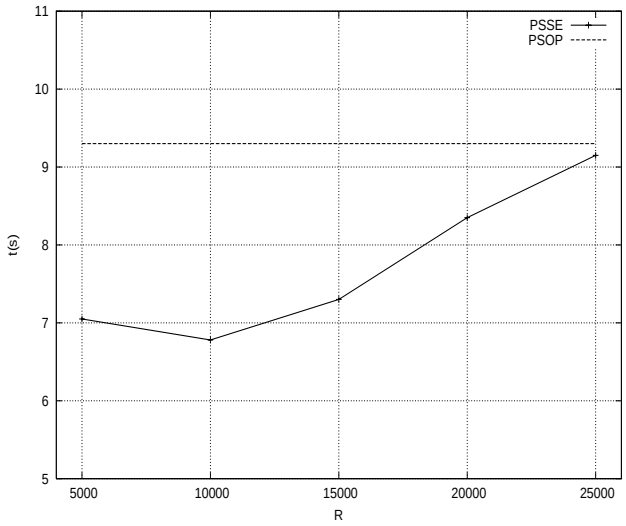

Figure 2: Comparison between PSOP and PSSE in a Stochastic Environment

In all simulation steps, α was set to 0.5 because there was no previous insight of the stochastic nature of the environment. In partially known environments that we have more information about the degree of the environment's stochastic nature, α can be highly effective in the system's performance, if it is chosen intelligently. Also $Size(l)$ was always set to 1000000, k set to 8 and s set to 5.

6 Conclusion

In stochastic environments the main factor for system performance is load-balancing. The PSSE algorithm acts well in speed-variant environments due to its dynamic load-balancing nature. The algorithm can tolerate sudden changes in processors' speed. Because of the implementation independent nature of the algorithm, it can be used in diverse and complex inter-operable software systems as well as multiprocessor architectures. Besides, the tradeoff between the communication cost and the restriction on the tasks' length R, should be well considered in each environment, depending to its own specific properties.

References

References :

[1] D. H. Baily, E. Barszczm, L. Dagum, and H. Simon. NAS parallel benchmark results. *IEEE Parallel and Distribued Technology,* 1(1):43-52, February 1993.

[2] K. Batcher. Sorting networks and their applications. In *Proc. of the AFIPS Spring Joint Computer Conference,* Volume 23, pages 307-314, 1968.

[3] http://www.fipa.org/, See the FIPA specifications.

[4] http://sharon.cselt.it/projects/jade/ See the JADE documentations.

[5] M.Ajtai, J. Komolos, and E.Szemeredi. Sorting in clog n parallel steps. *Combinatorica,* Volume 3:pages 1-3, 1983.

[6] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Toronto, 1985.

[7] H. Li and K. C. Sevcik. Parallel sorting by overpartitioning. In *Proc. of the 6t ACM symposium on Parallel algorithms and architectures,* pages 46-56, August 1994

[8] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers,* C-34(4):344-354, April 1985.

[9] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM,* 34(1):60-76, January 1987.

[10] R.Cole. Parallel merge sort. *SIAM Journal on Computing,* 17(4):770-785, August 1988.

[11] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on COmputing,* 18(3):594-607, June 1989.

[12] C. Kaklamanis, D. Kraizanc, L. Narayanan, and T. Tsantilas. Randomized sorting and selection on mesh-connected processot arrays. In *Proc. of Symposium on Parallel Algorithms and Architectures,* pages 17-28, Hilton Head, SC., July 1991.

[13] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. of Symposium on Parallel Algorithms and Architectures,* Velen, Germany, July 1993.

[14] G.C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volume I, General Techniques and Regular Problems.* Prentice Hall, 1988.

[15] E. Felten, S. Karlin, and S. Otto. Sorting on a hypercube. Technical Report, Hm 244, Caltech/JPL, 1986.

[16] E. Felten, S. Karlin, and S. Otto. Sorting on a hypercube. Technical Report, Hm 244, Caltech/JPL, 1986.

[17] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[18] B. A. Wagar. *Practical Sorting Algorithms for Hypercube Computers.* Phd thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, July 1990.

[19] M. J. Quinn. Analysis and benchmarking of two parallel sorting algorithms : hyperquicksort and quickmerge. *BIT,* 29(2):239-250, 1989.

[20] S. R. Seidel and W. L. George. Binosorting on hyper cubes with d-port communication. pages 1455-1461, 1988.

[21] Y. Won and S. Sahni. A balanced bin sort for hypercube multicomputers. *Journal of Supercomputing,* 2:435-448, 1988.

[22] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:362-372, 1992.

[23] X. Li, P.Lu, J. Schaeffer, J. Shillington, P. S. Wong and H.Shi. On the versatility of parallel sorting by regulaer sampling. *Parallel Computing*, 19)(10):543-550, October 1993.

[24] A. Borodin and J. Hopcroft. Routing, merging and sorting on parallel models of computation. *Journal of Computer Systems and Science*, 30:130-145, 1985.

[25] C. G. Plaxton. Efficient computation on sparse interconnection networks. Technical Report STAN-CS-89-1283, Stanford University, Department of Computer Science, Stanford, CA, September 1989.

[26] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 34(1):60-76, January 1987.

[27] A. Tridgell and R. P. Brent. An implementation of a general-purpose parallel sorting algorithm. Technical Report TR-CS-93-01 , Computer Science laboratory, Australian NAtional University, Australia, Februar 1993.

[28] D. T. Blackston and Ranade. Snakesort : A family of simple optimal randomized sorting algorithms. In *Proc. of 22nd International Conference on Parallel Processing*, Pages III-201-III-204, August 1993.

[29] P. P. Li and Y.-W. Tung. Parallel sorting on Symult 2010. In *Proc of 5th Distributed Memory Computing Conference*, pages 224-229, Charleston, SC., April 1990.

[30] R. S. Francis and L. J. H. Pannan. A parallel partition for enhanced parallel quicksort. *Parallel Computing*, 18:543-550, 1992

[31] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proc of the IEEE Computer Society's 7th International Computer Software and Applications Conference*, pages 627-631, 1983.

[32] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementation of randomized sorting on large parallel machines. In *Proc. of Symposum on parallel Algorithms and Architectures*, pages 158-167, San Diego, CA., July 1992.

[33] L. V. Kale′ and S. Krishnan. A comparison based parallel sorting algorithm. In *Proc. of 22nd International Conference on Parallel Processing*, pages III-196-III-200, August 1993.