Integer Sorting Algorithms for Coarse-Grained Parallel Machines

Khaled Alsabti
School of CIS
Syracuse University
kaalsabt@top.cis.syr.edu

Sanjay Ranka *
Department of CISE
University of Florida
ranka@cise.ufl.edu

Abstract

Integer sorting is a subclass of the sorting problem where the elements have integer values and the largest element is polynomially bounded in the number of elements to be sorted. It is useful for applications in which the size of the maximum value of element to be sorted is bounded. In this paper, we present a new distributed radix-sort *algorithm for integer sorting. The structure of our algorithm is similar to radix sort except that it typically requires less number of communication phases.*

We present experimental results for our algorithm on two distributed memory multiprocessors, the Intel Paragon and the Thinking machine CM-5. These results are compared with two other well known practical parallel sorting algorithms based on radix sort and sample sort. The experimental results show that the distributed radix-sort is competitive with the other two algorithms.

1 Introduction

Sorting has been a widely studied problem for parallel machines [8, 9, 12, 14, 13, 10, 1, 4, 7, 6, 11]. Integer sorting is a subclass of the sorting problem where the elements have integer values such that the value of the maximum element is polynomially bounded in the number of elements to be sorted. The work requirements of integer sorting is $O(n)$, where n is the number of elements, and the values are drawn from $[1..O(n^{\alpha})]$, where α is a positive constant.

In this paper, we present a new integer sorting algorithm, which we call *distributed radix-sort*. We compare our algorithm to two other well known sorting algorithms: *sample sort* and *radix sort*. A comparative study of these two algorithms for fine and coarse grained machines has been presented in [4, 6].

*The work of this author was supported in part by AFMC and ARPA under Contracts F19628-94-C-0057 and WM-82738-K-19 and a subcontract from Syracuse University. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

The sample sort has a single permutation step, while the radix sort requires multiple communication steps. Thus, radix sort incurs higher communication cost as compared to sample sorting. However, the computational cost of radix sort is better or comparable to the sample sort. The proposed algorithm, *distributed radix-sort* has the same structure as radix sort except that it requires less permutation steps in most cases. This makes it a competitive algorithm with sample sort especially for large number of processors for which the load imbalances generated by sample sort are higher.

We have implemented these algorithms on two distributed memory multiprocessors: the Intel Paragon and the Thinking machine CM-5. Our results, on both machines, show that our algorithm is very efficient and competitive to the other two algorithms. For 16-bit integers, our algorithm outperforms the other two algorithms. For 32-bit integers, it outperforms radix sort and is comparable to or better than sample based integer sort . However, it is inferior to sample based integer sort for 64-bit integers.

The rest of the paper is organized as follows. We describe the machine model in section 2. In sections 3 through 5, we describe and analyze the radix sort, sample sort and distributed radix-sort, respectively. Experimental results are presented in section 6. The conclusion is drawn in section 7.

2 Coarse-grained Parallel Machine

Coarse Grained Machines (CGMs) consist of a set of processors (tens to a few thousand) connected through an interconnection network. The memory is physically distributed across the processors. Interaction between processors is either through message passing or through a shared address space.

CGMs have cut-through routed networks which will be the primary thrust of this paper and will be used for modeling the communication cost of three algorithms.

Our analysis will be done for the following interconnection networks: hypercubes and two dimensional meshes. The analysis for permutation networks and hypercubes is the same in most cases. These cover nearly all commercially available machines. Although the three algorithms are analyzed for two types of interconnection networks, they are architecture independent and can be efficiently implemented on other interconnection networks.

Parallelization of applications requires distributing some or all of the data structures among the processors. Each processor needs to access all the non-local data required for its local computation. This generates aggregate or collective communication structures. Several algorithms have been described in the literature for these primitives and are part of standard textbooks [8, 9]. In what follows, p refers to the number of processors. We model the cost of sending a message from one node to another as $O(\tau + \mu m)$, where m is the size of the message. A brief description of the primitives is as follows:

1. **All-to-All Broadcasting:** This collective communication operation requires every node to send the same message of size m to all other processors.

2. **Global Combine and Prefix Scans:** Assume that each processor contains a vector $V_i[0...m]$. Let p be the number of processors. The global combine operation computes an element-wise sum of the local vectors in each processor. The resultant vector is stored an all processors. The global vector prefix-sum performs an element-wise prefix-scan of the local vectors in each processor.

3. **Transportation Primitive:** This operation performs many-to-many personalized communication with possibly high variance in message size. Let m be the maximum of outgoing or incoming traffic at any processor. If $m \geq p^2$, the running time of this operation can be shown to be equal to two all-to-all communication operations with a maximum message size of $O(\frac{m}{p})$ [3].

4. **Order Maintaining Data Movement:** Consider the following data movement problem : Initially, processor P_i has two integers s_i and r_i and s_i elements of data such that $\sum_{i=0}^{p-1} s_i = \sum_{i=0}^{p-1} r_i$. Let $s_{max} = max_{i=0}^{p-1} s_i$ and $r_{max} = max_{i=0}^{p-1} r_i$. The objective is to redistribute the data such that processor P_i contains r_i elements. Suppose that each processor has its set of elements stored in an array. We can view the $\sum_{i=0}^{p-1} s_i$ elements as if they were globally sorted based on processor and array indices. For any $i < j$, any element in processor P_i appears earlier in this sorted order than any element in processor P_j. In the order maintaining data movement problem, this global order should be preserved after the distribution of the data.

For detailed description and analysis of the above primitives the reader is referred to [9, 3, 8].

3 Radix Sort

Radix sort is a count-based sort that relies on the binary representation of the elements to be sorted [5]. The parallel radix sort has the same structure as the sequential radix sort. It performs $\lceil \frac{b}{r} \rceil$ iterations, where b is the number of bits in the binary representation of the elements and r is the number of bits examined at a time. We assume that each processor initially has $\frac{n}{p}$ elements, where n is the total number of elements and p is the number of processors. In each iteration of the algorithm , the rank of each element is computed using the current r-block. Then, the elements are permuted to their new destinations. The new destination of each element is determined using its rank.

The core step of each iteration of the algorithm is to compute the ranks of the elements. This can be done in parallel by finding the rank of the first element of each the 2^r *count* array for each processor. The rank of the first element of each entry of the count array is sufficient to find the rank of all the elements locally. After this step, the elements are redistributed among the processors using the transportation primitive. The destination processor of an element with rank i is the processor with address $\lfloor \frac{i}{\frac{n}{p}} \rfloor$. The structure of communication is balanced because each processor will send/receive $\frac{n}{p}$ elements and it is irregular because the individual messages may have different sizes. The final step, of each iteration, locally reorders the elements based on their ranks.

The total time [1] for performing the radix sort on hypercube and mesh is $O(\lceil \frac{b}{r} \rceil \delta (2^r + \frac{n}{p}) + \tau \log p + \mu 2^r + \tau p + \mu \frac{n}{p})$ and $O(\lceil \frac{b}{r} \rceil \delta (2^r + \frac{n}{p}) + \tau(\sqrt{p} - 1) + \mu 2^r (p - 1) + (\tau + \mu m)\sqrt{p})$, respectively [2].

4 Sample Sort

Sample sort partitions the list into m intervals such that the elements of each interval are smaller than the elements of the next interval. These intervals can then be independently sorted to sort the overall list.

[1] Many of the details have been omitted because of the space limitations.

[2] δ is the time to perform a unit computation on data available in the local memory

A sample of size s is selected from the n elements, and the range of the intervals is determined by sorting the sample and choosing $m-1$ elements called splitters. They represent the interval boundaries. The element values within an interval are between these boundaries. Many variations of sample sort have been studied [6, 11, 12, 4]. They differ in the method used for sampling, the method for sorting the sample, the size of the sample chosen and the number of splitters. Our sample based integer sort is very similar to the algorithm proposed by [12]. It uses regular sampling method to derive the splitters.

First, each processor sorts its local list using a sequential sorting algorithm. From the experiments we have performed, the local radix sort with radix 2^{11} outperformed the quick sort for 32-bit integers and large data set on both Intel Paragon as well as the CM-5. We have chosen the radix sort to perform the local sorting step.

There is a tradeoff in determining the sample size s. A large sample size leads to better load balance but results in higher time requirements in finding the splitters. The sample points are selected using regular sampling [12], i.e. it selects the elements at relative indices 1, $d+1$, ..., $(s-1)d+1$, where $d = \frac{\frac{n}{p}}{s}$.[3]

The sample points are globally sorted using Bitonic sort. Each processor i (except the last processor) picks the last number as splitter i in its sublist and performs *All-to-All Broadcast* with unit message size.

Then, each processor divides its list into p sublists using the $p-1$ splitters. This is done by performing binary search on the local list of size $\frac{n}{p}$ for the $p-1$ splitters. After forming the sublists, the elements are redistributed using the transportation primitive. The communication phase is irregular since the messages may have different sizes and is unbalanced since some processors may receive more than $\frac{n}{p}$. The p sublists are sorted to obtain the final sorted list. We use a sequential radix sort to carry out this step. [4].

Bucket expansion β is a measure of the degree of the load- balance achieved and is defined as the maximum number of elements assigned to a given processor divided by the average number of elements assigned to each processor. Assuming that the data set does not have any duplicates [5], the upper bound on the maximum number of elements assigned to any processor is given by $\frac{3}{2}\frac{n}{p} - \frac{n}{ps} + 1$ [12].

For comparing different algorithms, we require that the number of elements assigned to each processor after sorting are equal. To achieve this a *Order-Maintaining data movement* primitive is used in the final step of the algorithm.

The total time requirement of the sample sort on the hypercube and the mesh interconnection networks is $O(\delta(\lceil\frac{b}{r}\rceil(2^r + 2\frac{n}{p}) + s + (p-1)\log\frac{n}{p}) + (1 + \log p)\log p(\tau + \mu s) + (\tau p + \mu\beta\frac{n}{p}))$ and $O(\delta(\lceil\frac{b}{r}\rceil(2^r + \frac{n}{p}) + s + (p-1)\log\frac{n}{p}) + \tau\sqrt{p} + \mu s\sqrt{p} + (\tau + \mu\beta\frac{n}{p})\sqrt{p})$, respectively. This excludes the time of the Order-Maintaining Data Movement primitive. From the experiments, the time taken by this primitive is miniscule compared to the total running time of the algorithm.

5 Distributed Radix-Sort

The structure of the new proposed algorithm, distributed radix-sort, is similar to the structure of the radix sort. It requires $\lceil\frac{b}{r}\rceil$ iterations, where b is the number of bits in the binary representation and r is the number of bits examined at a time, of permuting the elements across the processors. Each iteration has eight steps.

The distributed radix-sort assumes that the count array (used in the radix sort) is distributed across all the processors. This has several advantages over radix sort:

1. The value of radix used in the distributed radix-sort 2^{r_1} is larger than the value used in radix sort 2^{r_2} [6].

2. The size of the prefix calculation required for distributed radix-sort is much smaller than the radix sort.

However unlike the radix sort, in each iteration the distributed radix-sort sends the elements to their destination processor using an intermediate phase. Each processor participates as source, intermediate as well as destination processor. The processing of these steps are much more complicated than the radix sort and are described in the following.

We assume that the count array (range) of the distributed radix sort is partitioned into pk buckets, where p is the number of processors and k is some positive constant. In every source processor, an array of size kp is created to record the number of the hits

[3] For the sake of simplicity, we assume that n is divisible by p and $\frac{n}{p}$ is divisible by s.

[4] Since the p sublists are sorted, we can merge them together instead of sorting the whole list. Although, this is asymptotically inferior, our experimental results suggest that this approach achieve better results.

[5] This can be easily generalized for presence of duplicate elements by concatenating the element index. However, this may increase the communication time.

[6] Typically $r_1 = r_2 + \log(kp)$.

to each bucket. Thus, each bucket corresponds to an interval of size $\frac{2^r}{kp}$.

In Step 1, we count the number of local hits to each of these buckets (element i belongs to bucket j where $j = \lfloor\frac{the\ \ current\ \ r-block\ \ of\ \ i}{kp}\rfloor$). This vector is referred to as *local-Bucket-Hits*.

In Step 2, a global vector sum-combine is performed on *local-Bucket-Hits* which results in an identical vector in every processor. This vector *Bucket-Hits* gives the total number of hits to each bucket. The rank of the first of element for each bucket is obtained by performing a local sequential prefix-sum-scan on *Buckets-Hits*. These kp entries give the contention for every bucket. A bucket is classified as being *sparse* if it has $\leq \frac{n}{kp}$ hits , and as *dense* otherwise. The intermediate processor for sparse bucket i is processor $\lfloor\frac{i}{k}\rfloor$. Sparse buckets, by definition, do not have more than $\frac{n}{kp}$ hits. Each intermediate processor has k buckets initially, and therefore will not receive more than $\frac{n}{p}$ elements for its sparse buckets. After assigning sparse buckets to intermediate processors, we stretch the dense buckets to intermediate processors in such that no processor will receive more than $\frac{n}{p}$ elements (from both sparse and dense buckets). Further, each dense bucket is stretched across consecutive numbered [7] processors. This is done as follows:

> For dense bucket i with b_i hits: look for the first intermediate processor j which is assigned less than $\frac{n}{p}$ elements , say n_j, and make it the first intermediate processor for bucket i. If $n_j + b_i \leq \frac{n}{p}$, then assign bucket i to processor j entirely and increment n_j by b_i ; otherwise, assign $\frac{n}{p} - n_j$ elements of bucket i to processor j, set n_j to $\frac{n}{p}$, decrement b_i by $\frac{n}{p} - n_j$ and examine the next intermediate processor for potential assignment.

The above process is repeated until all the elements of bucket i have been assigned.

Thus, a dense bucket can be split across several intermediate processors. Several source processors may have local data corresponding to a given dense bucket. One can allocate appropriate portions of the dense bucket to each of these source processors by performing a global vector prefix-sum-scan on *local-Buckets-Hits*. At the end of this step each processor can determine the exact portions (and corresponding intermediate processors) of each of the kp buckets it needs to communicate the data items.

[7]Note that some of these processors may be assigned zero elements. This might happen in case that they have been already assigned $\frac{n}{p}$ elements from the sparse buckets.

The communication between the source and intermediate processors (Step 3) can be shown to be a balanced transportation primitive since each intermediate processor will receive exactly $\frac{n}{p}$ elements.

Step 4 processes the data elements on intermediate processors. Finding the rank for elements belonging to the sparse bucket bucket is done by using a direct-address table of size $\frac{2^r}{kp}$. We calculate the count for the number of elements of each entry in this table. This can then be easily used to compute the global rank as rank of the first element of each bucket is locally available. This rank can be used to determine the destination processor for a given element. Element i with rank j is assigned to processor $\lfloor\frac{j}{\frac{n}{p}}\rfloor$.

For dense buckets, the processing is more complicated. We distinguish between two types of dense buckets: a *full* bucket and a *partial* bucket. The buckets that are contained entirely in a processor are called full buckets, while those that stretch across processors' boundaries are called partial buckets. Of the buckets in each processor, only the first and last buckets could be partial. If they exist, they are called a *preceding* partial bucket and a *succeeding* partial bucket, respectively. Processing of full dense buckets is the same as processing sparse buckets. The only difference is that the number of elements received for sparse buckets is smaller.

For partial dense buckets (Step 5), a global segmented-sum-scan and segmented-sum with a vector size of $\frac{2^r}{kp}$ is required to calculate the global rank of each of the elements. Each segment corresponds to a stretched bucket. If a processor has a succeeding as well as preceding partial bucket, we ignore the preceding partial buckets in such a processor for executing the segmented scan. These results are then augmented by having each processor with ignored preceding partial buckets send its contents to the first processor corresponding to a dense bucket. This step requires a simple one-to-one communication vector of size $\frac{2^r}{kp}$.

Step 6 assigns elements into destination processors as follows. An element with rank i is assigned to processor $\lfloor\frac{i}{\frac{n}{p}}\rfloor$.

Step 7 transfers data from intermediate to destination processors. It can be shown to be a balanced personalized communication since each intermediate processor has exactly $\frac{n}{p}$ and each destination processor will receive exactly $\frac{n}{p}$.

Step 8 reorders the the elements received based on their ranks.

The total time requirement of the distributed radix-sort is the time required by a single iteration times the number of iterations, $\lceil\frac{b}{r}\rceil$. The total computation time of a single iteration of the algorithm is $\delta O(kp + \frac{n}{p} + \frac{2^r}{kp}\log p + \frac{2^r}{p})$. The total communication cost of a single iteration of the algorithm for the hypercube and mesh interconnection networks is given in table 1.

We choose the number of buckets k per processor to satisfy $\frac{n}{p} > O(kp)$. There is a great deal of flexibility in choosing the value of k. k being close to 1 reduces the time for the prefix-scan/combine with the array of size kp, but increases the time for the segmented-sum-scan/segmented-sum with the array of size $\frac{2^r}{kp}$. k being close to p does the opposite.

Network	The time requirement
Hypercube	$O(\tau \log p + \mu kp + \tau \log p + \mu \frac{2^r}{kp} \log p + \tau p + \mu \frac{n}{p})$
Mesh	$O(\tau(\sqrt{p} - 1) + \mu(kp + \frac{2^r}{kp}(p-1)) + (\tau + \mu \frac{n}{p})\sqrt{p})$

Table 1: The time requirement of the communication cost for a single iteration of the distributed radix-sort on different interconnection networks

The distributed radix-sort allows us to choose a larger radix per iteration as compared to radix sort. This is because the count array is distributed across all the processors and the the parallel prefixes required are of size pk (as compared to 2^r for radix sort). However, an extra stage of communication and processing is required.

6 Experimental Results

We implemented the three algorithms on the Intel Paragon and the Thinking Machine's CM-5. We conducted extensive performance evaluation of these algorithms for a variety of parameters. In this section we briefly summarize our experimental results due to space limitations. For a detailed description and discussion of the experimental results the reader is referred to [2]. For each algorithm, we have assumed $16K$, $32K$, $64K$, $128K$ and $256K$ elements per processor using 32, 64, 128 and 256 processors. Two sets of element values have been generated for each of the above parameters assuming that b bits are used for representation of the elements:

1. *Alpha0*: The elements are uniformly distributed in the range $[0..2^b]$.

2. *Alpha1*: The elements are uniformly distributed from $[i\frac{2^b}{p}..(i+1)\frac{2^b}{p} - 1]$ where i is an arbitrary processor number.

The above two data sets represents uniform and skewed distributions for the target algorithms. For each of the algorithms, we have compared the total running time for the *Alpha0* and *Alpha1* data sets. All the algorithms are only marginally affected by data skew. The results in the rest of this section are for *Alpha0* data set.

For each data point, we ran the algorithm several times; the median of the running times is reported. The performance of the two radix sort algorithms depends on the radix while the sample sort depends on the size of the sample. We also studied, the effect of these parameters on the overall performance.

The optimal radix 2^r for radix sort depends on the cost of the prefix-sum/combine, the cost of the transportation primitive and the cache size. A large value of r results in less iterations needed to sort the elements, more time spent in prefix-sum/combine, and more time spent in counting and computing the ranks due to the cache misses, while smaller radix results in the opposite. We empirically determined this to be 2^{11}. Thus three iterations are required to sort the 32-bit integers.

The performance of the sample sort depends on the bucket expansion β. We ran the sample sort using different sample sizes s and found that choosing $s = n^{0.3}$ gives a good overall performance for sample sort in most of the cases. For sorting local data, we used a local radix sort (radix 2^{11}). This resulted in a slightly better performance than quick-sort for 32-bit integers.

The performance of the distributed radix-sort depends on the number of the buckets per processor (k) and the radix (2^r). A small value of k reduces the time for the prefix-scan/combine with the array of size kp, but increases the time for the segmented-sum-scan/segmented-sum of size $\frac{2^r}{kp}$. A small value of r results in a larger number of iterations needed to sort the elements and a smaller value of the bucket size. We empirically determined the value of k to be 16. The overall radix of the distributed radix-sort depends on the number of processors. We have determined the radix for each machine size empirically and used a value of 2^{18}, 2^{18}, 2^{19} and 2^{21} for 32, 64, 128 and 256 processors, respectively. Thus, the distributed radix-sort needs two iterations to sort the 32-bit integers. When using a radix 2^{21}, we switch to radix sort for the second iteration since only sort on 11 bits are required.

Figure 1 shows the communication and the computation times on 64 and 256 processors for each of the three algorithms for 32-bit integers. The radix sort spends almost equal time on the communication and computation. The sample sort on the other hand has large computation cost as compared to the communication cost. The time taken by order-maintaining data movement in sample sort is very small compared to the overall time. The computational requirements of the distributed radix-sort is smaller than the sample sort and the communication requirements are significantly lower than the radix sort. The overall time requirements of the radix sort is larger than the other two algorithms for all cases. The performance of distributed radix-sort is better or comparable to the sample sort. It outperforms sample sort for larger data sets and higher number of processors.

We also ran the three algorithms using 16-bit and 64-bit integers on the CM-5. For 16-bit integers, the distributed radix-sort outperforms the other two algorithms for all the data set sizes. This is because the distributed radix-sort requires only one iteration. For 64-bit integers, the sample sort outperforms the other two algorithms [8].

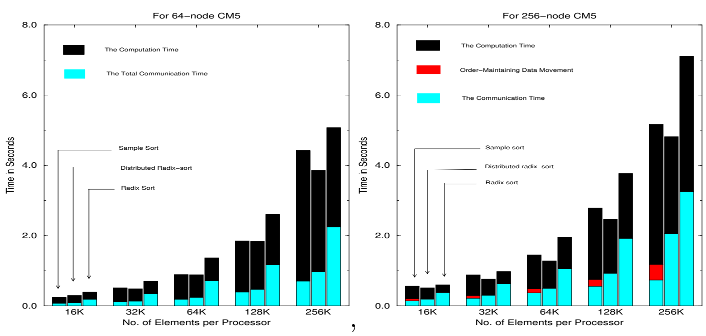

Figure 1: The Communication and computation times of the three algorithms on 64-node and 256-node CM-5

The CM-5 machine has relatively low unit computation to unit communication ratio. We also executed the three algorithms for 32-bit integers on Intel Paragon for 64 and 256 processors, which has higher unit computation to communication ratio than the CM-5. The distributed radix-sort outperforms radix sort in almost all the cases. Its performance is comparable or better than sample sort especially for larger number of processors. For 16-bit integers, the distributed radix-sort outperforms the other two algorithms for all the data set sizes on 64 and 256 processors.

7 Conclusion

In this paper, we have developed a new integer sorting algorithm called distributed radix-sort. Our experimental on the CM-5 and Intel Paragon suggest that it has superior performance than radix sort for the practical integer sizes used in practice. Further the performance of our algorithm is comparable or superior to the sample based integer sort for small sized integers (16-bit or 32-bit) especially for large data sets or number of processors.

[8] We have used a quick sort to perform the local sorting in the sample sort which gives a better performance than a local radix sort for 64-bit elements.

References

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

[2] K. Alsabti and S. Ranka. Integer Sorting Algorithms for Coarse-Grained Parallel Machines. *Technical Report, Department of CISE, University of Florida*, 1997.

[3] K. Alsabti, S. Ranka, and R. Shankar. The Transportation Primitives. *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[4] G. E. Blelloch et al. A Comparison of Sorting Algorithms for the Connection Machine CM-2. *In Proc. of Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, 1990.

[6] D. Culler et al. Fast Parallel Sorting under LogP: from theory to practice. *Proc. of the Workshop on Portability and Performance for Parallel Processing*, (Wiley), England, 1993.

[7] Leonardo Dagum. Parallel Integer Sorting with Medium and Fine-Scale Parallelism. *International Journal of High-Speed Computing.*

[8] G. Fox et al. *Solving Problems on Concurrent Processors: Vol. 1.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[9] V. Kumar et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc, 1994.

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

[11] Hui Li and Kenneth C. Sevcik. Parallel Sorting by Overpartitioning. *Technical Report CSRI-295*, February 1994.

[12] X. Li et al. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10), October 1993.

[13] S. Rajasekaran and J. H. Reif. Optimal and sublogrithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, June 1989.

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