A Randomized Sorting Algorithm on the BSP model*

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis
Computing Laboratory
Oxford University
United Kingdom

Abstract

We present a new randomized sorting algorithm on the Bulk-Synchronous Parallel (BSP) model. The algorithm improves upon the parallel slack of previous algorithms to achieve optimality. Tighter probabilistic bounds are also established. It uses sample sorting and utilizes recently introduced search algorithms for a class of data structures on the BSP model. Moreover, our methods are within a $1 + o(1)$ *multiplicative factor of the respective sequential methods in terms of speedup for a wide range of the BSP parameters.*

1. Introduction

The *Bulk-Synchronous Parallel* (BSP) model of computation has been proposed by L.G. Valiant [19, 20], as a unified framework for the design, analysis and programming of general purpose parallel computing systems. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software and provides a framework which permits the performance of parallel and distributed systems to be analyzed and predicted in a precise way. Since its introduction [19, 20], the model has been extensively examined [3, 8, 9, 10, 11]. The BSP computer as described in [19] consists of the following three components: (i) a *collection of* p *processor/memory components* each component consisting of a sequential processor with a block of local memory, (ii) a *communication network* that can deliver messages point to point among the components, (iii) and a *mechanism* for efficient *barrier synchronization* of all or a subset of the processing units. The performance of any BSP computer can be characterized by the following three parameters: p - the number of processing units, g - the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered, and L - the minimum elapsed time, measured in basic computational operations, between successive synchronization operations.

Computation on the BSP model proceeds in a succession of *supersteps*. During a superstep each processing unit is given a task to perform using data already locally available before the start of the superstep. The task may include computation steps on values held locally at the start of the superstep and message transmissions and receipts. A superstep can complete at any time after the first L time units. It should be noted that, although the model stresses global barrier mode synchronization, pairs of processing units may synchronize pairwise; such message exchanges, however, should respect the superstep rules [8]. As it was noted in [8], any particular BSP machine may impose a lower bound on L on a twofold basis: (i) there is a minimum time requirement for doing synchronization and (ii) the theoretical bound on g may be realizable only for large enough L so that the communication network can achieve the stated capacity. On the other hand, the maximum size for a superstep is a property of the application, since a superstep can accommodate as much work as can be done without a remote data exchange. The actual values of L and g for any BSP machine are likely to depend on p. The time complexity of a superstep S in a BSP algorithm is determined as follows. Each superstep is charged $\max\{L, x + gh\}$ basic time steps, where x is the maximum number of basic computational operations executed by any processor during S, and h is the maximum number of messages transmitted or received by any processor. Along the lines of [8], and in order to simplify the description of the performance of our algorithms, we assume that each superstep is either pure computation or pure communication.

We specify the performance of a BSP algorithm $\mathcal{A}$ in three parts. We specify first a sequential algorithm $\mathcal{A}^*$ which we are comparing with and make explicit how we charge for the basic operations of both $\mathcal{A}$ and $\mathcal{A}^*$. Then we specify two ratios π and μ. The former, π, is the ratio between the computation time $C_{\mathcal{A}}$ of $\mathcal{A}$ over the time $C_{\mathcal{A}^*}$ of $\mathcal{A}^*$ divided by

*The work of the first author was supported in part by EPSRC(UK) under grant GR/K16999 and the second author was supported in part by a Bodossaki Foundation Graduate Scholarship.

p, i.e., $\pi = pC_{\mathcal{A}}/C_{\mathcal{A}^*}$. The latter, μ, is the ratio between the parallel time $M_{\mathcal{A}}$ required by the communication supersteps of $\mathcal{A}$ and the computation time of $\mathcal{A}^*$ divided by p, i.e., $\mu = pM_{\mathcal{A}}/C_{\mathcal{A}^*}$. When we specify μ it is necessary to be explicit about the amount of information that can be conveyed in a single message. Finally, we specify conditions on n, p, L and g that are sufficient for the algorithm to make sense and the claimed bounds on π and μ to hold. If $\pi = c + o(1)$ and $\mu = o(1)$ we claim algorithm $\mathcal{A}$ is *one-optimal*. All asymptotic bounds refer to the problem size as $n \to \infty$. We also specify how to charge for local operations on the BSP model. Since the mechanisms of the BSP model are similar to those used by sequential models and we are interested in performance ratios of these two we can define operations in a higher level than machine level instructions. Throughout this work, $\lg$ and $\log$ denote logarithms to the base 2 and e respectively. We charge unit cost for operations over an associative operator in parallel-prefix computations, unit cost for each comparison performed, $n \lg n$ operations for sorting n elements sequentially, and $\lceil \lg n \rceil$ for performing binary search in a sorted sequence of length n. As noted in [8], had we charged $cn \lg n$ and $b \lg n$ operations for sorting and binary search respectively, for some constants $1 < b \leq c$, then neither π nor μ would have increased in the analysis of our algorithms.

The problem of sorting n keys on a parallel computing model has also drawn considerable attention. First approaches were sorting networks introduced by Batcher [13]. Since then, a wealth of literature concerning parallel sorting has been published [2, 5, 17]. These methods fail, however, to deliver optimal performance when implemented on the BSP model of computation. Previously known results on BSP sorting include deterministic [1, 9, 12] and randomized [8] algorithms. Our sorting algorithm improves upon these approaches in various aspects. In [1], it is shown that for all values of n and p such that $p = O(n^{1-\epsilon})$, $0 < \epsilon < 1$, a version of column-sort [14] provides an upper bound of $\lceil \epsilon^{-3.42} \rceil O(n \lg n/p)$ and $\lceil \epsilon^{-3.42} \rceil O(gn \lg n/p)$ for computation and communication respectively. In [9], an improved upper bound of $(1 + \lfloor \epsilon^{-1}(1-\epsilon) \rfloor ((1-\epsilon)/2) + o(1))(T_{seq}/p)$ and $O(gn/p))$ is shown for computation and communication respectively, where T_{seq} is the time required by sequential sorting. Experimental results indicate that the algorithm is also of practical value. In [12] a deterministic BSP sorting algorithm is presented. The algorithm is based on merge-sort [5]. It sorts n keys with $p \leq n$ processors in computation and communication time $O(n \lg n/p)$ and $O((n/p)(\lg n/\lg (n/p)))$ respectively. As the BSP algorithm is as involved as Cole's EREW algorithm, the constant factors involved in the analysis are large and the algorithm seems to be of little practical use. Finally the randomized algorithm of [8] requires, with high probability, computation and communication time $(1 + o(1))(T_{seq}/p)$ and $O(gn \lg n/(p \lg (n/p)))$ respectively, for all values of n and p such that $p = O(n/\lg^{1+\epsilon} n)$, for any constant $\epsilon > 0$. In this work we derive $(1 + o(1))T_{seq}/p$ and $O(gn \lg n/(p \lg (n/p)))$ computation and communication time bounds for all values of n and p such that $p = \omega(n \lg \lg n/\lg n)$, which improves upon the slack of previous randomized approaches. We also improve, in case of polylogarithmic slack, the success probability of previous algorithms [8].

2. Primitive Operations

BSP algorithms for several fundamental operations like *broadcasting* and *parallel-prefix* have already been presented [8, 3, 11]. Let the time required for broadcasting an n-word message be $T^n_{brd}(p) = M^n_{brd}(p)$, and that for for computing n independent parallel-prefix operations be $T^n_{ppf}(p) = C^n_{ppf}(p) + M^n_{ppf}(p)$. Detailed analysis of such algorithms and expressions for the corresponding bounds can be found in [11]. For $n = 1$, we write $T_{ppf}(p)$ (resp. $C_{ppf}(p)$, $M_{ppf}(p)$) for $T^1_{ppf}(p)$ (resp. $C^1_{ppf}(p)$, $M^1_{ppf}(p)$). The following integer sorting algorithm extends an idea of [16] (see [11] for a proof).

Lemma 2.1 *There exists a BSP algorithm for stable sorting n integers in the range $[0, 1, \ldots, R-1]$, that for any integer $2 \leq t \leq p$, requires time*

$$r \left(4 \max \left\{ L, O(\max \left\{ \left\lceil \frac{n}{p} \right\rceil, t \lceil \log_t p \rceil \right\}) \right\} + C_{ppf}(p) \right),$$

and

$$gr \left(3 \max \left\{ \frac{L}{g}, \max \left\{ \left\lceil \frac{n}{p} \right\rceil, t \lceil \log_t p \rceil \right\} \right\} + M_{ppf}(p) \right),$$

for computation and communication respectively, where $r = \lceil \lg R/\lg \max \{ \lceil n/p \rceil, t \lceil \log_t p \rceil \} \rceil$.

The randomized sorting algorithm to be introduced utilizes an efficient search procedure on *ordered h-level graphs* [6], a class of graphs that includes most of the standard data structures. For completion we define ordered h-level graphs. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. An *ordered h-level partitioning* of G is a partitioning of V into h ordered sequences of disjoint sets $V_1, V_2, \ldots, V_h$, together with an ordering of the elements in each subset V_i, $i \in \{1, 2, \ldots, h\}$. For every h-level partitioning of G we define, $\forall\, v \in V$, two parameters namely, $Level(v)$ and $Level\_Index(v)$ as follows: (i) $Level(v) = l$ if and only if $v \in V_l$, $l \in \{1, 2, \ldots, h\}$ and (ii) $Level\_Index(v)$ is the rank of v with respect to the ordering of the vertices in $V_{Level(v)}$. A directed graph $G = (V, E)$ with no double edges is called an *ordered h-level graph* if it has the following properties: (a) every vertex v in V has out degree of at most d, (b) the number of sources d_s of G is at most d, and (c) there exists an ordered h-level partitioning $V_1, V_2, \ldots, V_h$, of G such that: (i) every source of G is contained in V_1, (ii) if (u, v) is an edge of G, then $Level(v) = Level(u) + 1$, (iii) if (u, v) and $(\bar{u}, \bar{v})$ are two edges of G such that $Level(u) = Level(\bar{u})$ and $Level\_Index(u) < Level\_Index(\bar{u})$, then, we have that $Level(v) = Level(\bar{v})$ and $Level\_Index(v) \leq Level\_Index(\bar{v})$. As it is noticed in [6], any d-ary tree can be expressed as an ordered h-level graph. Let U be a universe of possible search queries on G. A *search path* for a query $q \in U$ is a sequence $Path(q) = \langle v_1, v_2, \ldots, v_h \rangle$ of h vertices of G, $v_i \in V_i$, prescribed by a *next* function $f : U \times (V \cup \{start\}) \to \{1, 2, \ldots, d\}$ as follows: (i) $f(q, start) = Index(v_1)$ and (ii) $f(q, v_i) = Index(v_{i+1})$, $i \in \{1, 2, \ldots, h-1\}$, where $(v_i, v_{i+1}) \in E$ and for every $v \in V$, $Index(v) = \sum_{i=1}^{Level(v)-1} |V_i| + Level\_Index(v)$. We do not assume that the search path is given in advance, only that it is constructed during the search process by successive applications of the *next* function f. Algorithms for efficiently searching an ordered h-level graph have been presented in [10, 11]. The following proposition summarizes some of the results obtained in [10, 11].

Proposition 2.1 *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of n queries and let $G = (V, E)$ be an ordered h-level graph, with vertex set $V = V_1 \cup V_2 \cup \ldots \cup V_h$ and edge set E, where $|V| = m \leq n$. Let the maximum out-degree of such a graph be $d = o(n/p)$, and the number of source vertices be $d_s \leq d$. If f is the next function, and $\tau_l(f)$ is the time required to evaluate f at level l, then, there exists a BSP algorithm for computing the next vertex in the search path of q_i, $\forall\, i \in \{1, 2, \ldots, n\}$, that for any level l, and using the charging policy of section 1, has,*

$$\pi = \frac{p}{\tau_l(f) n} \left(\tau_l(f) \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{n}{p} + \frac{m}{p} + L \lg p \right) \right),$$

and

$$\mu = g \frac{p}{\tau_l(f) n} O\left(\frac{n}{p} + \frac{m}{p} + \max\left\{ \frac{L}{g}, 1 \right\} \lg p \right).$$

The algorithm requires $O(n/p + m/p)$ space.

3. Randomized BSP Sorting

We apply these algorithms to obtain an efficient BSP algorithm for sorting n keys, that achieves one-optimality for a wide range of the parameters n, p, L and g. The algorithm derives from ideas of [7, 18], and utilizes the notion of *ordered h-level graphs* [6, 10] as well as the technique of *over-sampling* [8, 17]. A detailed exposition of the algorithm, with all the proofs, can be found in [11].

The algorithm is based on an efficient partitioning scheme and is recursive in nature. In particular, it realizes m recursive calls, where m may be equal to 1 or 2 in most practical situations. In each iteration i, $k_i - 1$ splitters are selected from the current set of size N_i, where $k_i = \lfloor (\lfloor \sqrt{N_i} \rfloor + 1)/s \rfloor$, and the set is partitioned into k_i buckets of approximately the same size. The next iteration performs a similar process of sub-partitioning each of these buckets into k_{i+1} further buckets, and so on. The size of every bucket produced in any of the m iterations is within a factor of $(1 + \varepsilon)$ of the mean N_i/k_i with high probability. The remainder of this section is devoted to the derivation of the following proposition.

Proposition 3.1 *Let ω_n be any function of n such that $\omega_n > 2$ for sufficiently large n. Then, for any positive integers n and p such that $n/p = \Omega(\lg n / \lg\lg n)$, there exists a randomized BSP sorting algorithm that for any δ such that $1 < \delta < (\lg\lg n - 1)/2$, and using the charging policy of section 1, has, with probability $1 - o(1)$,*

$$\pi \leq \frac{p}{n \lg n} \left((1 + \frac{2}{\omega_n}) \frac{n \lg n}{p} + O\left((\frac{n}{p} + L \lg n) \frac{\lg n}{\lg \frac{n}{p}} \right) \right),$$

and

$$\mu \leq \frac{gp}{n \lg n} O\left(\left(\frac{n}{p} + \frac{L \lg n}{g} \right) \frac{\lg n}{\lg \frac{n}{p}} \right).$$

An upper bound of the failure probability is of the algorithm is the following one.

$$\lg\lg n \left(n^{-\Omega\left((n/p) \lg\lg^2 n \right)} + n^{-\Omega\left(\lg^{\delta-1} n / (\lg\lg n\ \omega_n)^2 \right)} \right).$$

Proof: The algorithm is recursive in nature, and each recursive call is divided into a sequence of stages. Initially, the variables N, l and r have the values $N = n$, $l = 0$ and $r = p-1$ respectively. In any recursive call of the algorithm, these variables give respectively the size of the problem, the leftmost processor and the rightmost processor participating in the call. The oversampling factor s is set to $\lceil \lg^{\delta} n \rceil$, for any δ such that $1 < \delta < (\lg\lg n - 1)/2$. The input is assumed to be evenly distributed among the p processors at the beginning of the execution of the algorithm. Moreover, we assume that the keys are distinct since we can always make them so by, for example, appending to them the code for the memory location in which they are stored. We note that the algorithm requires subset processor synchronization.

Program RAND_SORT (N, l, r)

begin
Stage 1 : If $(l = r)$ or $(l \neq r$ and $N \leq \lg^{\lg\lg n} n)$, then the $q = r - l + 1$ processors participating in this recursive call of the algorithm sort the input by some sequential deterministic sorting method in the former case and some deterministic

parallel sorting method in the latter case. Subsequently, the algorithm terminates.

[Analysis of Stage 1: The sorting procedure is realized in one of two ways depending on the values of l, r and N. The analysis of this stage is treated in detail in the discussion following the description of this algorithm.

[Case $(l = r)$**]** Processor l sorts, by using a deterministic sequential sorting algorithm, the keys in its possession.

[Case $(l \neq r$ **and** $N \leq \lg^{\lg \lg n} n)$**]** The $q = l - r + 1$ processors participating in this recursive call of the algorithm sort the keys in their possession, by using an adaptation on the BSP model of odd-even merge-sort [13]. **]**

Stage 2 : The q processors participating in this recursive call of the algorithm, choose at random without replacement a sample of size $ks - 1 = \lfloor (\lfloor \sqrt{N} \rfloor + 1)/s \rfloor s - 1 \leq \lfloor \sqrt{N} \rfloor$.

[Analysis of Stage 2: The sample selection method is as follows. Each processor selects uniformly at random with replacement $\lceil (ks-1)/q \rceil$ (or $\lfloor (ks-1)/q \rfloor$) processor identifiers for a total, over all processors, of $\lfloor \sqrt{N} \rfloor$ identifiers. The total number of identifiers selected by any processor is bounded above by $\lceil (ks-1)/q \rceil \leq \lceil \lfloor \sqrt{N} \rfloor / q \rceil$. The processor identifiers selected are routed to the so identified processors. Processor p_i that received t_i identifiers in the previous step, chooses uniformly at random without replacement a set of size t_i among its keys. A parallel-prefix sum operation is performed on the number t_i of selected keys. Using the information of the previous step the keys are distributed evenly among the q participating processors. By way of Chernoff bounds [4] it is proved that the overall computation and communication time required for this stage is bounded above by, $\max\{L, \Theta(\lceil \sqrt{N}/q \rceil)\} + \max\{L, \Theta(N/q)\} + C_{ppf}(p)$ and $2\max\{L, \Theta(gN/q)\} + M_{ppf}(p)$ respectively, with probability at least $1 - qe^{-\Omega\left(N \lg \sqrt{N}/q\right)}$. **]**

Stage 3 : The sample of size $\lfloor \sqrt{N} \rfloor$ is sorted using a BSP adaptation of an algorithm in [15].

[Analysis of Stage 3: When this step is executed there are $\lfloor \sqrt{N} \rfloor$ sample keys and q processors. Therefore, each physical processor is assigned a $\lceil \sqrt{N/q} \rceil \times \lceil \sqrt{N/q} \rceil$ segment of computation. We thus get that this stage takes at most

$$\max\left\{L, O\left(\frac{N}{q}\right)\right\} + C_{ppf}^{\lfloor \sqrt{N/q} \rfloor}(\sqrt{q}),$$

$$\max\left\{L, O\left(g\frac{\sqrt{N}}{q}\right)\right\} + 2M_{brd}^{\lfloor \sqrt{N/q} \rfloor}(\sqrt{q}) + M_{ppf}^{\lfloor \sqrt{N/q} \rfloor}(\sqrt{q}),$$

time for computation and communication respectively. **]**

Stage 4 : A balanced d-ary tree T is constructed over the $k = \lfloor (\lfloor \sqrt{N} \rfloor + 1)/s \rfloor$ buckets induced by the $k-1$ sorted splitters that evenly partition the sample.

[Analysis of Stage 4: We set $d = 2\lceil (N/q)^{\alpha} \rceil$, for any $\alpha < 1$. In order to simplify the exposition we assume that k is an integral power of d. This assumption can be removed without affecting the asymptotic analysis of the algorithm. The total computation and communication time complexity of this stage is bounded respectively, as $k \leq q$, by $\max\{L, O(k/q)\} + \lceil \log_d q \rceil \max\{L, O(d)\}$ and $\lceil \log_d q \rceil \max\{L, O(gd)\}$. **]**

Stage 5 : Each processor decides the bucket destination of every key it holds with respect to the $k-1$ splitters.

[Analysis of Stage 5: This stage can be realized by employing the algorithm implied by proposition 2.1. Given that the height of the balanced d-ary tree of stage 4 is at most $\log_d k$ by substituting in proposition 2.1, for $n = N$, $m = k$ and $p = q$ we get that this stage takes time $\max\{L, \lg k \lceil N/q \rceil\} + \log_d k O(N/q + L) + 3\log_d k C_{ppf}(q)$ and $\log_d k O(gN/q + L) + 3\log_d k M_{ppf}(q)$ for computation and communication respectively. **]**

Stage 6 : The N input keys are redistributed according to their bucket destination. If $q/k \geq 1$ we assign $\lfloor q/k \rfloor$ processors per bucket, otherwise we assign $\lceil k/q \rceil$ buckets per processor.

[Analysis of Stage 6: We assume that q (respectively, k) divides k (respectively, q). This assumption can be removed without affecting the asymptotic analysis of the algorithm, i.e., by assuming that p is a power of two and that the number of splitters selected in stage 4 is a power of two as well. This stage is equivalent to one of integer sorting, where the N integers to be sorted belong to the range $[0, 1, \ldots, k-1]$. Thus, the keys can be redistributed according to their bucket destination by employing the radix-sort algorithm implied in lemma 2.1. Substituting for $n = N$, $p = q$, $R = k$ and $r = \lceil \lg k / \lg \max\{\lceil N/q \rceil, t\lceil \log_t q \rceil\} \rceil$ in lemma 2.1 the computation and communication time of this stage is derived. **]**

Stage 7 : Recursively sort the elements in each bucket, for all buckets concurrently.

end

It is first shown that the size N_i of a bucket in the i-th recursive call can be bounded so that the number of recursive calls realized by Algorithm RAND_SORT is decided to be at most $\lceil \lg \lg n - \lg(\lg(\max\{\lg^{\lg \lg n} n, \Omega(\frac{n}{p})\}) - (2\delta + 1)\lg \lg n - 1) \rceil$, with probability at least $1 - n^{-\Omega(\lg^{\delta - 1} n/(m\omega_n)^2)}$. By establishing upper and lower bound for the quantity N_i/q_i in the i-th recursive call, upper bounds for $\sum_i \lg q_i$ and $\sum_i \lg k_i$, and noting that the failure probability for each recursive call of stage 2 is at most $n^{-\Omega\left((n/p) \lg \lg^2 n\right)}$, the result follows. A detailed proof can be found in [11]. ■

By substituting appropriate values for the parameters in proposition 3.1 a variety of bounds on the runtime of algorithm RAND_SORT can be derived. By making, however, more generic selections for these parameters and worst-case bounds for p, L and g more general (but less tight) results in terms of n can be obtained. The following are such bounds.

Theorem 3.1 *For any positive constants c and ϵ, $0 < \epsilon < 1$, there exists a BSP algorithm for sorting n keys that for all positive integers n, p and L satisfying $p = n^{1-\epsilon}$ and $L = o(n^{\epsilon})$ has, with probability $1 - n^{-\Omega(\lg^c n)}$, $\pi = 1 + o(1)$, and if $g = o(\lg n)$ also, $\mu = o(1)$.*

Theorem 3.2 *For any constant $\alpha \geq 1$, there exists a BSP algorithm for sorting n keys that for all positive integers n, p and L satisfying $p = n/\lg^{\alpha} n$ and $L = o(\lg^{\alpha-1} n \lg\lg n)$, has, with probability $1 - n^{-\Omega\left(\lg^{\alpha} n \lg\lg^{2} n\right)}$, $\pi = 1 + o(1)$, and if $g = o(\lg\lg n)$ also, $\mu = o(1)$.*

Theorem 3.3 *There exists a BSP algorithm for sorting n keys that for all positive integers n, p and L satisfying $p = n\lg\lg n/\lg n$ and $L = O(1)$, has, with probability $1 - n^{-\Omega(\lg n \lg\lg\lg n)}$, $\pi = O(1)$, and if $g = O(1)$ also, $\mu = O(1)$.*

We note that not only can the bounds given above be expressed in a variety of ways, but also the algorithm itself leaves room for several improvements. For example, sample sorting could be performed by a more efficient randomized parallel algorithm [8]. We also note that the recursion may terminate for smaller values of N than $\lg^{\lg\lg n}$ such as $\lg^{\beta} n$, for any $\beta = o(\lg\lg n)$; this improves the asymptotic performance of the algorithm. Furthermore, the analysis of the ordered h-level graph algorithm can be improved by taking into consideration that the underlying structure is a d-ary tree, and the queries progress in a balanced fashion through the tree.

References

[1] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of the 7-th ACM Symposium on Parallel Algorithms and Architectures,* pp. 129-136, 1995.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(\log n)$ sorting network. In *Proceedings of the 15-th Annual ACM Symposium on Theory of Computing,* pp. 1-9, 1983.

[3] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of ESA '95*, LNCS, Springer-Verlag, 1995.

[4] B. Bollobás. *Random Graphs*. Academic Press, New York, 1984.

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

[6] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing,* 8:367-375, 1990.

[7] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM,* 17(3):496-507, July 1970.

[8] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267, 1994.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM SPAA*, Padova, Italy, 1996.

[10] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of EURO-PAR'96*, LNCS, Springer-Verlag, 1996.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, 1996.

[12] M. T. Goodrich. Communication-Efficient Parallel Sorting. *Proceedings 26th ACM STOC*, 1996.

[13] D. E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, Reading, 1973.

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

[15] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *J.ACM,* 29(3):642-667, 1982.

[16] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing,* 18:594-607, 1989.

[17] H. J. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *J.ACM,* 34:60-76, 1987.

[18] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing,* 14(2):396-409, 1985.

[19] L. G. Valiant. A bridging model for parallel computation. *C.ACM,* 33(8):103-111, 1990.

[20] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science,* (J. van Leeuwen, ed.), North Holland, 1990.