Selection on the Bulk-Synchronous Parallel Model with Applications to Priority Queues[1]

Alexandros V. Gerbessiotis and Constantinos J. Siniolakis
Computing Laboratory, Oxford University, Wolfson Building, Oxford OX1 3QD, UK.

Abstract

In this paper we present a new randomized selection algorithm on the Bulk-Synchronous Parallel (BSP) model of computation, along with an application of this algorithm to dynamic data structures, namely Parallel Priority Queues (PPQs). We show that our methods improve previous results upon both the communication requirements and the amount of parallel slack required to achieve optimal performance. We also establish that optimality to within small multiplicative constant factors can be achieved for a wide range of parallel machines. While these algorithms are fairly simple themselves, descriptions of their performance in terms of the BSP parameters is somewhat involved. The main reward of quantifying these complications is that it allows transportable software to be written for parallel machines that fit the model. We also present experimental results for the selection algorithm that reinforce our claims.

1 Introduction and the BSP Model

The main technical contribution of this work is the architecture independent study of the communication and synchronization requirements of a fundamental computational problem namely, selection, along with an application to Parallel Priority Queues (PPQs). A PPQ is an abstract data structure that generalizes the well-known sequential priority queue data structure [11]. A p-processor PPQ implementation allows the efficient simultaneous insertion of n new items, n/p per processor, and the simultaneous deletion of the n items associated with the smallest values. As noted in [14], PPQs can be employed to efficiently implement several parallel techniques, such as *branch-and-bound* [10], which require the solution of several sub-problems, each associated with a possibly different cost. By utilizing PPQs, the p processors can insert at each phase the newly generated sub-problems and efficiently select the n "global best" sub-problems to be solved in the next phase. In this work we establish time bounds for PPQ operations as a function of the available communication throughput of the system.

The computational model adopted is the Bulk-Synchronous Parallel (BSP) model, which deals explicitly with the notion of communication and synchronization among computational threads. The BSP model was originally proposed by L. G. Valiant [19] as a bridge between software and hardware. The BSP computer as described in [19] consists

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

of three parts: a collection of processor-memory components; a communication network with a router that can deliver messages point-to-point among the components; and facilities for global synchronization, in barrier style, of all or a sub-set of the components. A BSP computer is parametrized by p - the number of processor-memory components, L - the minimal time, measured in terms of basic computational operations, between successive synchronization operations, and g - the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the router in terms of words of information delivered. Computation on the BSP model proceeds in a succession of *supersteps*. A superstep may be thought of as a segment of a process during which each processor performs a given task using data already available there locally before the start of the superstep [12]. The task may include (i) local computations, (ii) message transmissions, and (iii) message receipts. 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.

1.1 Related Work and New Results

The selection problem, that is, the computation of the C-th smallest key from a set of n keys has always been an interesting research topic [2, 4, 5]. Median finding in linear time has been examined in [2] and lower bounds for various values of C were also presented in [2]. The minimum average number of comparisons required to find the C-th largest of n keys has also drawn considerable interest. A lower bound of $n + C - O(1)$, $1 \leq C \leq \lceil n/2 \rceil$, was presented in [4], which matches the performance, within low order terms, of the algorithm in [5]. A sequential randomized algorithm for median finding that can be further extended to a parallel one has been proposed in [7]. The sequential version requires $3n/2 + O(n^{2/3} \log^{1/3} n)$ comparisons, with high probability. In the remainder of this paper we shall use $\log x$ and $\lg x$ to denote the logarithm of x to the base e and 2 respectively. An improved sequential median finding algorithm based on a bootstrapping technique of [5] was also suggested in [7] that required with high probability $3n/2 + O(n^{1/2+\beta})$ comparisons, for any constant $\beta > 0$. In this work we generalize these median finding results to the general selection problem and we extend them by deriving parallel implementations. The parallel algorithms we present work on the BSP model of computation and require $(n + C)/p + o(n/p)$ parallel time, with probability $1 - n^{-c}$, for any constant $c > 0$. Previous parallel selection algorithms include the results in [15]. Our techniques provide tighter probabilistic bounds and optimal number of comparisons, within lower order terms, on a realistic parallel computational model. We also show that these results can be extended to efficiently solve a more interesting problem that of parallel priority queues subject to the operations of insert, find-min and delete-min [14, 16, 17, 18].

Previous approaches based on traditional *concurrent* priority queues [17] provide only limited parallelism. In this abstract, we concern ourselves with *parallel* priority queues that offer the prospect of a high degree of parallelism. The first approach on parallel priority queues is implied in the work of Karp and Zhang [10]. They derive a parallel randomized algorithm to approximate a priority queue. Their algorithm was extended to handle exact priority queues in [16, 18]. Pinotti and Pucci [14] derive deterministic algorithms for maintaining parallel priority queues. They introduce two new data structures, the n-Bandwidth-Heap and the n-Bandwidth-Leftist-Heap, that are obtained as extensions of the sequential binary-heap and leftist-heap.

These approaches fail however to deliver optimal performance when directly implemented on the Bulk-Synchronous Parallel model of computation. In this work we improve upon the communication requirements of previous results on parallel priority queues [14], and extend the randomized results of [10, 16, 18] by improving upon both

the communication requirements and the amount of parallel slack required to achieve optimal performance. We also show that optimality to within small multiplicative constant factors can be achieved for a wide range of the BSP parameters. The high-probability algorithms we present are based on a new randomized processor-mapping of a variant of n-Bandwidth-Heaps. In particular, the underlying data structure is a mixture of n-Bandwidth-Heaps [14] and concurrent (pipelined) heaps [17], and is based on an efficient selection algorithm to achieve low contention.

1.2 Primitive Operations

In this section we state results on BSP algorithms for several fundamental operations, namely *broadcast*, *parallel-prefix* and *sorting*. The proofs of these results can be found in [8, 9]. These primitives are auxiliary routines for the randomized selection and priority queue algorithms presented in the following sections.

Lemma 1 *There exists a BSP algorithm for broadcasting an n-word message that requires time at most $M^n_{brd}(p) = (\lceil n/\lceil n/h\rceil\rceil + h - 1)\max\{L, gt\lceil n/h\rceil\}$, for any integer $1 \le t \le p$, where $h = \lceil \log_t((t-1)p+1)\rceil - 1$.*

Lemma 2 *There exists a BSP algorithm for computing n independent parallel prefix operations that requires time at most $C^n_{ppf}(p) = 2(\lceil n/\lceil n/h\rceil\rceil + h - 1)\max\{L, t\lceil n/h\rceil\}$ and $M^n_{ppf}(p) = 2(\lceil n/\lceil n/h\rceil\rceil + h - 1)\max\{L, gt\lceil n/h\rceil\}$, for computation and communication respectively, for any integer $1 \le t \le p$, where $h = \lceil \log_t p\rceil$, for a total time of $T^n_{ppf}(p)$.*

If $n = 1$, then we write $T_{ppf}(p)$ (respectively $C_{ppf}(p)$, $M_{ppf}(p)$) for $T^1_{ppf}(p)$ (respectively $C^1_{ppf}(p)$, $M^1_{ppf}(p)$). The same notational convention applies to all pipelined operations. The randomized sorting algorithm is based on random sampling techniques [6] and is a variation of an algorithm presented in [13]. The fundamental difference between our algorithm and the ones in [6] is that it works for $p > n$ and utilizes exclusively broadcast, parallel prefix and regular routing operations. The randomized algorithm selects a sample of size at most $\sqrt{p}$ that is sorted by the Nassimi and Sahni algorithm. A set of $k - 1 = o(\sqrt{p})$ splitters is then chosen. The input keys are divided into k buckets by a procedure that requires only parallel prefix and regular routing operations. The k buckets are then sorted recursively by allocating the same number of processors to each bucket. The recursion terminates when for each bucket, its size is at most the square root of the number of processors allocated to it. The Nassimi and Sahni algorithm is then used to sort the keys of each bucket. The randomized algorithm's performance is summarized in the following theorem [9].

Theorem 1 *There exists a randomized BSP sorting algorithm that with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, sorts n keys with $p = n^{1+\epsilon}$ processors, for any constant $0 < \epsilon \le 1$, in time $O((1/\epsilon + \lceil \lg(1/\epsilon)\rceil)T_{ppf}(p)) + O((1/\epsilon + \lceil \lg(1/\epsilon)\rceil)\max\{L, g\,O(1)\})$.*

2 Randomized Selection

We initially present results related to random sampling that generalize those presented in [7] for the special case of median finding. In the discussion and proofs to follow we use the notation originally used in [6]. Let $X = \{x_1, \ldots, x_N\}$ be a set of input keys ordered such that $x_i < x_{i+1}$, for $1 \le i \le N - 1$. We are assuming that the keys are distinct because by appending to each key its index we can make them so. We are interested in finding the C-th smallest of the keys in set X. We assume, without loss of generality, that $C \le \lceil N/2\rceil$. We can use symmetric arguments to find the $(N - C)$-th largest key.

Let then $c = C/N$. For integers $\lambda \geq 1$ and $s \geq 1$, let $Y = \{y_1, \ldots, y_{2\lambda s-1}\}$ be a randomly chosen subset of $2\lambda s - 1 \leq N/2$ keys of X, also ordered so that $y_i < y_{i+1}$. The following three subsets are then formed for some positive integers μ_0, μ_2, and $t \geq 1$ such that $2c\lambda s > \mu_0 t/2$: $X_0 = \{x \mid x \in X \text{ and } x < y_{2c\lambda - \mu_0 t/2}\}$, $X_1 = \{x \mid x \in X \text{ and } y_{2c\lambda s - \mu_0 t/2} < x < y_{2c\lambda s + \mu_2 t/2}\}$, and $X_2 = \{x \mid x \in X \text{ and } y_{2c\lambda s + \mu_2 t/2} < x\}$.

The randomized BSP selection algorithm we describe in the following discussion partitions the input key set into three sets as above. By establishing lower and upper bounds on the sizes of X_0 and X_2 we guarantee that the required statistic is located in set X_1. We thus need to determine lower bounds on the size of set X_0, and symmetrically for X_2, by proving that the probability of having such a set of size less than $cN - f(N)$, for some function $f(N)$ of N, is negligible. We investigate bounds on $f(N)$ for this to hold. As the two cases for X_0 and X_2 are identically handled, we show the result by substituting $\mu_0 = \mu_2 = \mu$. Similarly, upper bounds on the size of set X_0, and symmetrically for X_2, can be established. We deduce the following relationships used later for $n \geq N$. The proofs of the following results can be found in [9].

Proposition 1 *Let $C \leq \lceil N/2 \rceil$ and $c = C/N$. Let $\lambda, \mu \geq 1$, $s, t \geq 1$, $2\lambda s - 1 < N/2$, $n \geq N$, $2c\lambda s > \mu t/2$, $\mu t \geq 3$, and $\rho > 0$. If $f(N) = (N\ \mu t)/(2\ \lambda s)$ and $\mu t = 2\sqrt{3\rho\ 2c\ \lambda s\ \log n}$, the probability that set X_0 is of size more than $cN - f(N) - 1$ is at least $1 - n^{1-\rho}$.*

Proposition 2 *Let $C \leq \lceil N/2 \rceil$ and $c = C/N$. Let $\lambda, \mu \geq 1$, $s, t \geq 1$, $2\lambda s - 1 < N/2$, $n \geq N$, $2c\lambda s > \mu t/2$, and $\rho > 0$. If $\mu t \geq 2\sqrt{2\rho\ 2c\ \lambda s\ \log n}$ then the probability that set X_0 is of size at most $cN - 1$ is at least $1 - n^{1-\rho}$.*

The following bounds on the size of X_0 can then be derived for $s = t = 1$. Similar bounds can be derived by symmetric arguments for set X_2.

Claim 1 *Let $C \leq \lceil N/2 \rceil$ and $c = C/N$. Let $\lambda \geq 1$, $2\lambda - 1 < N/2$, $n \geq N$, and $\rho > 1$. Let $\mu_0 \geq 2\sqrt{2\rho\ 2c\ \lambda\ \log n} \geq 1$ and $2c\ \lambda > \mu_0/2$. Then, with probability at least $1 - n^{1-\rho}$, set X_0 is of size at most $C - 1$.*

Claim 2 *Let $C \leq \lceil N/2 \rceil$ and $c = C/N$. Let $\lambda \geq 1$, $2\lambda - 1 < N/2$, $n \geq N$, and $\rho > 1$. Let $2c\lambda > \mu_0/2$ and $\mu_0 = 2\sqrt{3\rho\ 2c\ \lambda\ \log n} \geq 3$. Then, with probability at least $1 - n^{1-\rho}$, set X_0 is of size at least $cN - N\sqrt{3\rho\ 2c\ \log n}/\sqrt{\lambda}$.*

The randomized selection algorithm of this section derives from the ideas in [5] and extends the results presented in [7]. The general formulation of the algorithm is as follows. A random sample of the input is initially chosen, and two keys within this sample are also selected so that the C-th smallest key is between them. The iterative version of the algorithm, which improves the lower order terms of the non-iterative version, proceeds so that in the following step a new sample is selected that is a superset of the sample selected in the previous iteration. Depending on the size of the sample in the first iteration and the number of iterations, the sequential version requires time $n + C + O(n^{1/2+\beta})$ comparisons, where $\beta > 0$ depends on these choices. The number of iterations determines a lower bound on the value of β. A BSP algorithm is derived from this selection method that uses essentially parallel-prefix, broadcast, and regular routing operations for sample sorting. The procedure to manage the selection process is summarized in the following proposition.

Proposition 3 *For any constant $\rho > 1$, there exists a randomized BSP selection algorithm that, with probability at least $1 - O(n^{1-\rho})$, determines the C-th smallest of n keys*

that for any integers n and $p \leq n$, requires time

$$T_{sel}(n,C,p) = \begin{cases} \frac{n+C}{p} + O(\sqrt{\frac{C\log n}{p}} + \frac{n^{2/3+\delta}\lg^{4/3} n}{p} + g\frac{n^{2/3+\delta}\lg^{1/3} n}{p}) + O(T_{ppf}(p)) & \text{if } p \leq n^{2/3+\zeta}, \\ \frac{n+C}{p} + O(\sqrt{\frac{C\log n}{p}} + \log n) + O(T_{ppf}(p)) & \text{otherwise,} \end{cases}$$

for any arbitrarily small constants ζ and δ, such that $0 < \zeta < \delta < 1/3$.

Proof. We assume that the keys are randomly and evenly distributed among the processors. If the keys are not initially randomly distributed a pre-randomization phase is required for optimal time bounds to be derived, that is, the BSP algorithm requires $(n+C)/p + o(n/p)$ comparisons with high probability. For a non-random distribution, the BSP algorithm requires, with high probability, $2n/p + o(n/p)$ comparisons. In the latter case, the algorithm is work-optimal as the total comparison count over all processors is still $n + C + o(n)$. The process to manage the selection operation is outlined in procedure SELECT(X, C) (figure 1), where X denotes the input set and C the required statistic.

```
begin SELECT (X, C)
1.   let c = C/n, λ = n^{2/3}(3ρ log n)^{1/3}((√2c + √2(1−c))/2)^{2/3},
         such that 2λ − 1 < n/2 and ρ > 1 ;
2.   let μ_0 = 2√(3ρ 2c λ log n), μ_2 = 2√(3ρ 2(1−c) λ log n) ;
3.   select randomly a sample S = {y_1, y_2, ..., y_{2λ−1}} of size 2λ − 1 ;
4.   for all processors in parallel
5.       do SORT(S) ;
6.   denote by ⟨y*_1, y*_2, ..., y*_{2λ−1}⟩ the sequence of keys in S in sorted order ;
7.   if (c ≥ 2ρ log n/λ) then
8.       let m = y*_{2cλ−μ_0/2}, M = y*_{2cλ+μ_2/2} ;
9.   else
10.      let m = −∞, M = y*_{2cλ+μ_2/2} ;
11.  BROADCAST({m, M}) ;
12.  for each processor p^⟨k⟩, k ∈ {0, 1, ..., p − 1}, in parallel
13.      do let r^⟨k⟩ = l^⟨k⟩ = 0, R^⟨k⟩ = R_r^⟨k⟩ = R_l^⟨k⟩ = {} ;
14.         for i = 1 to |X^⟨k⟩|
15.             do if (X^⟨k⟩[i] > M) then
16.                    let r^⟨k⟩ = r^⟨k⟩ + 1, R_r^⟨k⟩ = R_r^⟨k⟩ ⋃{X^⟨k⟩[i]} ;
17.                elseif (X^⟨k⟩[i] < m) then
18.                    let l^⟨k⟩ = l^⟨k⟩ + 1, R_l^⟨k⟩ = R_l^⟨k⟩ ⋃{X^⟨k⟩[i]} ;
19.                else
20.                    let R^⟨k⟩ = R^⟨k⟩ ⋃{X^⟨k⟩[i]} ;
21.  PARALLEL_PREFIX_SUM({r^⟨k⟩, l^⟨k⟩}) ;
22.  BROADCAST({Σ_k r^⟨k⟩, Σ_k l^⟨k⟩}) ;
23   for all processors in parallel
24.      do if (Σ_k l^⟨k⟩ < C and Σ_k r^⟨k⟩ < n − C) then
25.             SORT(⋃_k R^⟨k⟩) ;
26.             denote by R* the sequence of keys in ⋃_k R^⟨k⟩ in sorted order ;
27.             return R*[C − Σ_k l^⟨k⟩] ;
28.         else
29.             re-execute procedure SELECT ;
end SELECT
```

Figure 1: Procedure SELECT.

In the remainder of this proof we briefly sketch the time complexity of procedure SELECT; a detailed proof appears in the full version of the paper [9]. The local operations

(lines 1-2, 8, 10, 13, 26) take time $O(1)$. The sample selection (line 3) is realized by a method similar to that of [7] in time that is bounded by the time complexity of the sorting operation (line 5). The sample is sorted (line 5) by one of two algorithms depending on the number of processors available. In particular, we distinguish three cases for the analysis of this stage.
[Case $p = \lambda^{1-\epsilon}$ for any constant $0 < \epsilon < 1$]] The algorithm implied in theorem 5 of [6] is employed. The time required for this step is bounded above by $O(T_{ppf}(p))+O(\lambda \lg \lambda/p)+ O(g\lambda/p)$.
[Case $p = \lambda^{1+\epsilon}$, for any constant $\epsilon > 1/3$] We use the algorithm implied by theorem 1, for $n = \lambda$. The time complexity of this step is $O(T_{ppf}(p)) + O(\max\{L, g\, O(1)\})$.
[Otherwise] The algorithm implied in theorem 5 of [6] is employed with p/n^{δ} processors, for any arbitrarily small constant $0 < \delta < 1/3$. The time required for this step is bounded above by $O(T_{ppf}(p)) + O(n^{\delta}\lambda \lg \lambda/p) + O(gn^{\delta}\lambda/p)$.

The time complexity of the broadcast and parallel-prefix operations (lines 11, 21-22) is bounded above by $O(T_{ppf}(p))$. The iterative comparison sequence (lines 15-20) is realized in a single superstep. As each processor has at least $\lfloor n/p \rfloor$ keys altogether, we can prove, by way of Chernoff bounds, that the total number of comparisons at this step is bounded by $(n+C)/p+\sqrt{(3\rho \log n)p/C}C/p$ if $C/p > 3\rho \log n$ and $(n+C)/p+O(\log n)$ otherwise, with probability $1 - O(n^{1-\rho})$.

By way of claims 1 and 2, for $N = n$, the total number of keys between pivot keys m and M is at most 2λ, with probability $1-O(n^{1-\rho})$. Moreover, since the input is randomly distributed among the processors, we can show, by way of Chernoff bounds that, with the same probability, each processor has in its local memory at most $\max\{2\lceil 2\lambda/p \rceil, 6e\rho \log n\}$ such keys if $p < n^{2/3+\zeta}$, for any constant $0 < \zeta < 1/3$, and $O(1)$, otherwise. Thus, the time complexity of the sorting operation (line 24) is identical to that of the sorting operation in (line 5). We note, however, that the keys to be sorted are first evenly distributed among the processors. The information collected through the parallel-prefix and broadcast operations (lines 21-22) can be used to effectively realize the required even distribution. By way of the preceeding analysis, the actual distribution (routing operation) takes time $\max\{L, g \max\{2\lceil 2\lambda/p \rceil, 6e\rho \log n\}\}$, if $p < n^{2/3+\zeta}$, for any constant $0 < \zeta < 1/3$, and $\max\{L, O(g)\}$, otherwise. Combining the costs over all steps of procedure SELECT and substituting for λ gives the desired result. □

In order to improve lower order terms an iterative version of the aforementioned algorithm has also been designed. The algorithm is based on a bootstrapping technique of [5] and is presented in the full version of the paper [9]. The main result is stated in the following proposition [9].

Proposition 4 *For any constant $\rho > 1$, there exists a randomized BSP selection algorithm that, with probability at least $1 - O(n^{1-\rho})$, determines the C-th smallest of n keys, that for any integers n and $p \leq n$, requires time,*

$$T_{sel}(n, C, p) = \begin{cases} \frac{n+C}{p} + O(\sqrt{\frac{C \log n}{p}} + \frac{n^{1/2+\delta} \lg^{3/2} n}{p} + g\frac{n^{1/2+\delta} \lg^{1/2} n}{p}) + O(T_{ppf}(p)) & \text{if } p \leq n^{1/2+\zeta}, \\ \frac{n+C}{p} + O(\sqrt{\frac{C \log n}{p}} + \log n) + O(T_{ppf}(p)) & \text{otherwise,} \end{cases}$$

for any arbitrarily small constants ζ and δ, such that $0 < \zeta < \delta < 1/3$.

In the parallel priority queue algorithms we describe in the following section, it is sometimes desirable that h independent selection operations are performed. Each such selection operates on disjoint data sets of size n and the same statistic is involved in all operations. The probabilistic arguments in the analysis of procedure SELECT related to the data distribution, can be extended, due to the independency of the involved problems, to handle simultaneously all h selection problems. This observation can be exploited to decrease the amount of parallel slack required by as much as h. Moreover,

since the operations involved in the selection process include broadcast and parallel-prefix we could improve the selection process by pipelining these operations (refer to section 1.2). Hence, we can verify the following analogue of proposition 3 [9]. A similar result can be obtained for proposition 4.

Proposition 5 *For any constant $\rho > 1$, there exists a randomized BSP algorithm that, with probability at least $1-O(n^{1-\rho})$, solves h independent selection problems on the same statistic C and on disjoint data sets, each of size n, that for any integers n and $p \leq n$, requires time,*

$$T^h_{sel}(n,C,p) = \begin{cases} h(\frac{n+C}{p} + O(\sqrt{\frac{C\log n}{hp}} + \frac{n^{2/3+\delta}\lg^{4/3} n}{p} + g\frac{n^{2/3+\delta}\lg^{1/3} n}{p})) + O(T^h_{ppf}(p)) & \text{if } p \leq n^{2/3+\zeta}, \\ h(\frac{n+C}{p} + O(\sqrt{\frac{C\log n}{hp}})) + O(\log n) + O(T^h_{ppf}(p)) & \text{otherwise,} \end{cases}$$

for any arbitrarily small constants ζ and δ, such that $0 < \zeta < \delta < 1/3$.

3 BSP Priority Queues

In this section we study a recently introduced data structure, namely the *Parallel Priority Queue* (PPQ) [14], and formulate a PPQ implementation on the BSP model. We present results on PPQs that improve upon the communication requirements of previous approaches [14] and extend the randomized algorithms of [10, 16, 18] by improving upon both the communication requirements and the amount of parallel slack required to achieve optimal performance. We also show that optimality to within small multiplicative constant factors can be achieved for a wide range of the BSP parameters.

A PPQ is an abstract data structure for maintaining a collection of (possibly duplicate) items and selecting the n items associated with the smallest values. The following operations are defined on a PPQ Q. INSERT($\{y_1, y_2, \ldots, y_n\}$, Q) for the insertion of items $\{y_1, y_2, \ldots, y_n\}$ in Q. FINDMIN(n, Q) for the determination of the n smallest items of Q. DELETEMIN(n, Q) for the deletion of the n smallest items of Q. The implicit assumption is that there exists an operation MAKEQUEUE(Q) for the construction of an empty queue Q. The high-probability algorithms we present in this paper for supporting the pre-mentioned PPQ operations are based on a new randomized processor-mapping of a variant of n-Bandwidth-Heaps introduced in [14]. Formally, the new data structure, which we call (n,d)-*Bandwidth-Concurrent-Heap* (in short, (n,d)-BCH) is defined as a concurrent d-ary heap whose nodes, each containing n items, are arranged in *extended heap order*. By extended heap order we imply that the value of an item at any node is no greater than the value of the items at each of its children. By employing (n,d)-BCHs (in contrast to $(n,2)$-BCHs) we achieve a speedup for insertions at the expense of deletions. We can therefore adapt the parameter d so as to select a data structure that fits the relative frequencies of the operations.

Let m be the number of items stored in a (n,d)-BCH Q, with $k = m/n$ being the number of nodes in Q. The depth h of Q is $\lceil \log_d((d-1)k+1) \rceil - 1$. Such a heap Q can be conveniently implemented in terms of the following data structure. Each processor $p^{\langle i \rangle}$, $i \in \{0, 1, \ldots, p-1\}$, maintains a one-dimensional array $A^{\langle i \rangle}[1 \ldots k]$ of size k, where each position of the array is associated with a list $L_j^{\langle i \rangle}$ of items, $j \in \{1, 2, \ldots, k\}$. Each such list $L_j^{\langle i \rangle}$ contains the items of node v_j that reside on processor $p^{\langle i \rangle}$. We note that the order of the items within these lists is irrelevant. Associated with Q are the data fields *full_level* and *last_elem* [17]. Data field *full_level* is the index of the first node in the deepest non-empty level h of Q, and *last_elem* is the index of the last non-empty node of Q. For an empty heap, *full_level* and *last_elem* are both set to 0. The path from the root to the first empty node (target) of Q, denoted by σ_{ins}, is called the *insertion-path* of Q. Then, as shown in [17] for the binary case, the insertion path σ_{ins} of Q can be

represented by a sequence of $h-1$ digits in radix d, and hence, the next node in the insertion path can be determined in $O(1)$ time.

3.1 The BSP PPQ Algorithm

In this section we present a new approach for implementing PPQs. The general formulation of our randomized technique is as follows. The n insert (respectively, delete-min) processes are realized in a sequence of phases. The number of phases corresponds to the height h of the underlying (n,d)-BCH. At each phase l, $l \in \{1,2,\ldots,h-1\}$, the processes advance in a pipelined top-down fashion through the data structure along their insert (respectively, delete-min) paths from level l to level $l+1$. Each phase is realized by utilizing the selection algorithm of section 2. The algorithm terminates within h phases. We note that previous approaches employing Bandwidth-Heaps [14] are quite complex and are based on communication intensive merging and sorting algorithms. The procedures to manage the insert and delete-min operations are summarized in the following proposition.

Proposition 6 *Let ω_n be any function of n, such that $\omega_n \to \infty$ as $n \to \infty$. Then, for any constant $\rho > 0$, there exists a BSP algorithm for supporting the PPQ operations* Insert *and* DeleteMin*, that, with probability at least $1 - O(dhn^{-\rho})$, for all integers n, p, and h, such that $n/p = \omega_n \lg n$, requires time,*

$$h\left(T_{sel}(2pr,n,p) + T_{ppf}(p) + \max\{L, r+O(1)\} + O(1)\right) + \max\{L, gr\},$$

$$(h-1)\left((d-1)T_{sel}(dpr,n,p) + T_{sel}(2pr,n,p) + T^d_{brd}(p) + T_{ppf}(p) + \max\{L, O(d\lg d)\}\right),$$

for procedures Insert *and* DeleteMin *respectively, where h is the depth of the underlying (n,d)-BCH, $r = \lceil (1+\varepsilon)(n/p)\rceil$, and $\varepsilon = O(\sqrt{1/\omega_n})$.*

Proof. We initially present the procedures to realize the insertion and deletion operations on a (n,d)-BCH; later, we provide a correctness proof and an analysis of the resources required. Some notation is required before proceeding to the details of the algorithms. Let v be a node in a (n,d)-BCH Q. By E_v we denote the set of n items stored in node v of Q, and by $\max\{E_v\}$ the maximum item of E_v. Moreover, let nodes $\{u^1, u^2, \ldots, u^d\}$ be the children of v. By $\min_j\{u^j\}$ we denote the child of v that has the minimum $\max\{E_{u^j}\}$. Finally, the path from the root to the first empty node of Q, denoted by σ_{ins}, is called the insertion-path of Q.

The process to manage the insertion operation on a (n,d)-BCH Q is outlined in procedure Insert$(\{y_1, y_2, \ldots, y_n\}, Q)$ (figure 2). We note that in order to achieve communication efficiency we maintain the following randomized mapping of Q on the processors. The items $\{y_1, y_2, \ldots, y_n\}$ are randomly distributed among the p processors at the beginning of the insertion process (line 8). In later phases, whenever items are to be transferred from one node to another they are simply re-labeled locally (refer to lines 13, 15-16), and therefore, no inter-processor communication is required during these subsequent phases. This step (line 8) is the only place where physical movement of items occurs (beyond input and output). Thus, in the remainder of this analysis whenever a data movement is implied, a local re-labeling operation actually takes place. This re-labeling technique guarantees the low-contention of our algorithms and preserves the random distribution of items to processors.

We observe that the insertion process preserves the extended heap order of Q. Particularly, for each node $v^{(i)}$, $i \in \{1,2,\ldots,|\sigma_{ins}|\}$, in the insertion path σ_{ins} of Q, the value of $\max\{E_{v^{(i)}}\}$ is not incremented by procedure Insert. Consequently, the items stored in the children of $v^{(i)}$ are no less than $\max\{E_{v^{(i)}}\}$ and the correctness of the procedure follows. The resources required for a single iteration of procedure Insert are determined

```
begin INSERT ({y_1, y_2, ..., y_n}, Q)
1.    for i = 1 to h
2.        do denote by Y set {y_1, y_2, ..., y_n} ;
3.            let node v^(i) at level i, i ∈ {1, 2, ..., h}, be the next node in the
                  insertion path σ_ins of Q ;
4.            if (i = 1) then
5.                let last_elem = last_elem + 1 ;
6.                if ((last_elem ≥ 2full_level)) then
7.                    let full_level = last_elem ;
8.                distribute randomly the items of Y among the p processors ;
9.            for all processors in parallel
10.               do let e = SELECT(E_{v^(i)} ⋃ Y, n) ;
11.           for each processor p^⟨k⟩, k ∈ {0, 1, ..., p − 1}, in parallel
12.               do if (i = h) then
13.                       let E_{v^(i)}^⟨k⟩ = Y^⟨k⟩ ;
14.                   else
15.                       let E_{v^(i)}^⟨k⟩ = {x | x ∈ E_{v^(i)}^⟨k⟩ ⋃ Y^⟨k⟩ and x ≤ e} ;
16.                       let Y^⟨k⟩ = {x | x ∈ E_{v^(i)}^⟨k⟩ ⋃ Y^⟨k⟩ and x > e} ;
17.   FINDMAX(E_{v^(i)}) ;
end INSERT
```

Figure 2: Procedure INSERT.

as follows. The local operations (lines 3, 5, 7) require $O(1)$ time. The random distribution of items to processors (line 8) requires at most $\max\{L, gr\}$ time with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, where $r = \lceil (1+\varepsilon)(n/p) \rceil$, and $\varepsilon = O(\sqrt{1/\omega_n})$. This claim can be derived by way of Chernoff bounds on the right tail of the binomial distribution [1]. For the remainder of this proof, we therefore assume that each processor has at most $r = \lceil (1+\varepsilon)(n/p) \rceil$ items from any node in Q. The median selection process (line 10) takes at most $T_{sel}(2pr, n, p)$ time. We note that we use the expression $2pr$ to compensate for the uneven distribution of items to processors. Finally, the re-labeling operation can be realized as a by-product of the selection process, and thus, the local operations (lines 13, 15-16) require $O(1)$ time. For implementation purposes, we require that the maximum item of the newly added node is determined and maintained (line 17). This step can be implemented by performing a linear scan over the items of the node followed by a parallel-prefix operation, for a total time of $\max\{L, r + O(1)\} + T_{ppf}(p)$.

The process to manage the deletion operation on a (n, d)-BCH Q is more complicated. We first define an auxiliary routine ADJUST on a set of sibling nodes that does not violate the extended heap property and facilitates the deletion process. In particular, this procedure rearranges the items stored in the children of $v^{(i)}$ so that we can shift up the items of one of children and still maintain the extended heap order [14]. The process is outlined in procedure ADJUST$(n, \{u^1, u^2, \ldots, u^d\})$ (figure 3), where n is the size of a node and $\{u^1, u^2, \ldots, u^d\}$ is a set of at most d sibling nodes.

Procedure ADJUST re-labels the items stored in nodes $\{u^1, u^2, \ldots, u^d\}$, while maintaining the extended heap order, so that after its execution, node $u^{\pi(1)}$ holds the n smallest items of the set, node $u^{\pi(2)}$ holds the next n smallest items of the set, and so on, where initially $\langle u^{\pi(1)}, u^{\pi(2)}, \ldots, u^{\pi(d)} \rangle$ is the sequence of nodes in $\{u^1, u^2, \ldots, u^d\}$, such that $\max\{E_{u^{\pi(k)}}\} < \max\{E_{u^{\pi(m)}}\}$, for all $k < m \in \{1, 2, \ldots, d\}$. The preservation of the extended heap order follows directly from the fact that $\max\{E_{u^j}\}$, $j \in \{1, 2, \ldots, d\}$, is not incremented by the process. The time complexity of procedure ADJUST is derived as follows. The determination of sequence $\langle u^{\pi(1)}, u^{\pi(2)}, \ldots, u^{\pi(d)} \rangle$ (line 2) can be realized by broadcasting items $\max\{E_{u^j}\}$, for all $j \in \{1, 2, \ldots, d\}$, to all processors in

begin ADJUST $(n, \{u^1, u^2, \ldots, u^d\})$
1. **for** all processors in parallel
2. **do let** $\langle u^{\pi(1)}, u^{\pi(2)}, \ldots, u^{\pi(d)} \rangle$ be the sequence of nodes in $\{u^1, u^2, \ldots, u^d\}$, such that $\max\{E_{u^{\pi(k)}}\} < \max\{E_{u^{\pi(m)}}\}$, $\forall\, k < m \in \{1, 2, \ldots, d\}$;
3. **let** $e_j = \text{SELECT}(\bigcup_j E_{u^j}, jn)$, $\forall\, j \in \{1, 2, \ldots, d-1\}$;
4. **for** each processor $p^{\langle k \rangle}$, $k \in \{0, 1, \ldots, p-1\}$, in parallel
5. **do let** $E^{\langle k \rangle}_{u^{\pi(1)}} = \{x \mid x \in \bigcup_j E^{\langle k \rangle}_{u^j}$ and $x \leq e_1\}$;
6. **let** $E^{\langle k \rangle}_{u^{\pi(j)}} = \{x \mid x \in \bigcup_j E^{\langle k \rangle}_{u^j}$ and $e_{j-1} < x \leq e_j\}$, $\forall\, j \in \{2, \ldots, d-2\}$;
7. **let** $E^{\langle k \rangle}_{u^{\pi(d)}} = \{x \mid x \in \bigcup_j E^{\langle k \rangle}_{u^j}$ and $x > e_{d-1}\}$;

end ADJUST

Figure 3: Procedure ADJUST.

time $T^d_{brd}(p)$, and subsequently, by sorting locally these items in time $\max\{L, O(d \lg d)\}$ [11]. The selection process (line 3) takes at most $(d-1)T_{sel}(dpr, n, p)$. We note that for sufficiently large d it might be more efficient to sort the dpr items instead of performing $d-1$ selection processes. The re-labeling operation can be realized as a by-product of the selection process, and therefore the local operations (lines 5-7) take $O(d)$ time.

The deletion process is realized by procedure DELETEMIN(n, Q) (figure 4), where n denotes the number of items to be deleted from Q.

begin DELETEMIN (n, Q)
1. **for** $i = 1$ **to** $h - 1$
2. **do if** $(i = 1)$ **then**
3. **return** $E_{root(Q)}$;
4. $root(Q) = v_{last_elem}$;
5. **let** node $v^{(1)} = root(Q)$ be the first node in the deletion path of Q ;
6. **let** $last_elem = last_elem - 1$;
7. **if** $((last_elem < full_level))$ **then**
8. $full_level = full_level/2$;
9. denote by $u^1, u^2, \ldots, u^d$ the children of node $v^{(i)}$;
10. ADJUST $(n, \{u^1, u^2, \ldots, u^d\})$;
11. **for** all processors in parallel
12. **do** $e = \text{SELECT}(E_{v^{(i)}} \bigcup E_{\min_j \{u^j\}}, n)$;
13. **for** each processor $p^{\langle k \rangle}$, $k \in \{0, 1, \ldots, p-1\}$, in parallel
14. **do** $E^{\langle k \rangle}_{v^{(i)}} = \{x \mid x \in E^{\langle k \rangle}_{v^{(i)}} \bigcup E^{\langle k \rangle}_{\min_j \{u^j\}}$ and $x \leq e\}$;
15. $E^{\langle k \rangle}_{\min_j \{u^j\}} = \{x \mid x \in E^{\langle k \rangle}_{v^{(i)}} \bigcup E^{\langle k \rangle}_{\min_j \{u^j\}}$ and $x > e\}$;
16. **let** node $v^{(i)} = \min_j \{u^j\}$ be the next node in the deletion path of Q ;

end DELETEMIN

Figure 4: Procedure DELETEMIN.

We note that the deletion process maintains the extended heap order of Q. In fact, for each node $v^{(i)}$, $i \in \{1, 2, \ldots, h\}$ in the deletion path of Q, the value of $\max\{E_{v^{(i)}}\}$ is not incremented by procedure DELETEMIN by way of procedure ADJUST (line 10). The time required for a single iteration of procedure DELETEMIN is determined as follows. The deletion of the n smallest items (line 3) and the local operations (lines 4-5, 6, 8, 16) can be realized in $O(1)$ computation time. The time required for the ADJUST procedure (line 10) was analyzed in the previous paragraphs. The selection process (line 12) takes at most $T_{sel}(2pr, n, p)$ time. The re-labeling operations can be realized as a by-product of the selection process, and therefore, the local operations (lines 14-15) take $O(1)$ time.

The stated probability bound derives from the fact that procedures INSERT, ADJUST, and DELETEMIN access a total of at most $O(hd)$ nodes of the underlying graph. Since the success probability (per node) of the stated operations is $1 - O(n^{-\rho})$, the desired bound follows. □

By substituting appropriate values for the parameters in proposition 6 a multiplicity of bounds on the time required for PPQ operations can be obtained. The following is such a bound [9]. If we replace in the following result $o(.)$ and $\omega(.)$ by $O(.)$ the time bounds still hold.

Theorem 2 *For any constant $\rho > 0$, there exists a BSP algorithm for supporting the PPQ operations* INSERT, FINDMIN, *and* DELETEMIN, *that, with probability at least $1 - O(dhn^{-\rho})$, for all integers n, p, L, g, d, and h, such that $n/p = \omega(\lg n)$, $d \lg d = o(n/p)$, $L = o(n/(p \lg p))$, and $g = o(h)$, requires time, $h(3(n/p) + o(n/p))$ per* INSERT, *$O(1)$ per* FINDMIN, *and $(h-1)((d^2+2)(n/p) + o(n/p))$ per* DELETEMIN *operation, where h is the depth of the underlying (n,d)-BCH.*

3.2 Introducing Concurrency

The procedures presented in the previous section to manage the insert and delete-min operations can be conveniently pipelined, since both manipulate the underlying (n,d)-BCH heap from top to bottom. The pipelining of these operations can be exploited on a twofold basis. First, procedures INSERT and DELETEMIN heavily utilize parallel-prefix and broadcast operations and therefore, the underlying algorithms are expected to benefit from pipelining these operations. Second, pipelining can be exploited in order to decrease the amount of parallel slack required by as much as h, where h is the height of the underlying heap. In the remainder of this section we focus on the latter case. The former case can be treated by a similar line of arguments. The pipelining process is straightforward when procedures INSERT and DELETEMIN do not operate simultaneously. Then we can verify the following analogue of proposition 6.

Proposition 7 *Let ω_n be any function of n, such that $\omega_n \to \infty$ as $n \to \infty$. Then, for any constant $\rho > 0$, there exists a BSP algorithm for supporting the PPQ operations* INSERT *and* DELETEMIN *(not simultaneously) in a pipelined fashion, that, with probability at least $1 - O(dhn^{-\rho})$, for all integers n, p, and h, such that $\bar{h}n/p = \omega_n \lg n$, requires time,*

$$T^h_{sel}(2pr, n, p) + T^h_{ppf}(p) + h\left(\max\{L, r + O(1)\} + O(1)\right) + M_{distr},$$

$$2\left((d-1)T^{\frac{h-1}{2}}_{sel}(dpr, n, p) + T^{\frac{h-1}{2}}_{sel}(2pr, n, p) + T^{d\frac{h-1}{2}}_{brd}(p) + T^{\frac{h-1}{2}}_{ppf}(p)\right) + (h-1)\max\{L, O(d \lg d)\},$$

for procedures INSERT *and* DELETEMIN *respectively, where h is the depth of the underlying (n,d)-BCH, $r = \lceil (1+\varepsilon)(n/p) \rceil$, $\varepsilon = O(\sqrt{1/\omega_n})$, $\bar{h} = \min\{(h-1)/2, \omega_n \lg n\}$, and*

$$M_{distr} = \begin{cases} \max\left\{L, g\, O\left(\min\left\{\lg n, \frac{n}{p}\frac{\lg n}{\lg\lg n}\right\}\right)\right\} & \text{if } \frac{n}{p} = O(\lg n), \\ \max\left\{L, g(1+o(1))\frac{n}{p}\right\} & \text{otherwise.} \end{cases}$$

Proof. The proof follows a similar line of arguments to that of proposition 6 and is briefly sketched in the following discussion. Pipelining is possible since procedures INSERT and DELETEMIN both manipulate the underlying (n,d)-BCH from top to bottom and they both require a small portion of the heap at a time, consisting of one level for the INSERT procedure and two levels for the DELETEMIN procedure. By substituting therefore in the analysis of proposition proposition 6 for the broadcast and parallel-prefix operations, their pipelined versions, and by employing proposition 5 we obtain the desired result. We note that the expression for $\bar{h}$ expresses the portion of the degree of concurrency that

is exploited to decrease the amount of parallel slack required in proposition 6. The only operation that cannot be efficiently pipelined is the initial distribution of items (line 5) of procedure INSERT (this is due to the synchronous nature of the BSP model). In the remainder of this proof we focus on bounding the time M_{distr} required for this initial distribution. We distinguish two cases.
[Case $\frac{n}{p} = O(\lg n)$] For any constant $\rho > 0$, the random distribution of items to processors results, with probability $1 - O(n^{-\rho})$, in at most $O(\min\{\lg n, (n/p)\lg n/\lg\lg n\})$ items per processor. This claim can be derived by way of Chernoff bounds on the right tail of the binomial distribution [3].
[Otherwise] This case can be treated with arguments similar to that of the analysis of the non-pipelined version of the algorithm (see proposition 6). □

Even though procedures INSERT and DELETEMIN both manipulate the underlying heap from top to bottom, the problem of realizing them concurrently turns out to be more complicated [17]. In particular, if procedure INSERT is still in progress when procedure DELETEMIN begins its execution, then the target node for procedure DELETEMIN is empty. We are then faced with two options, neither of which is satisfactory: (i) procedure DELETEMIN selects some other leaf node, which might result in an unbalanced heap, and (ii) procedure DELETEMIN delays its operation until the most recent INSERT procedure finishes, which results in limited concurrency. To alleviate these limitations we adopt an implementation that is based on the locking scheme of [17], that achieves concurrency, while preserving the consistent and deadlock-free operation of the underlying algorithms. Instead of locking the entire heap, we lock only a small portion of the heap at a time, consisting of one node for the INSERT procedure and $d+1$ nodes for the DELETEMIN procedure. Distinct insert and delete-min operations access the contents of a node only after locking it to ensure mutual exclusion. The implementation details follow. We associate a field called *status* with every node in the heap. This field may obtain two distinct values, each associated with the following semantics [17]: *Pending* - an insertion is in progress which will ultimately insert a key at the node, and *Wanted* - a delete-min operation is waiting for the key. When procedure INSERT starts, the status of its target is set to *Pending*. Accordingly, when procedure DELETEMIN starts (and an insert operation is still in progress), the status of the target of the last insert operation is set to *Wanted*, and the DELETEMIN procedure is delayed for one phase. Moreover, at each iteration of procedure INSERT, the status of the target node is checked, and if it has been set to *Wanted*, node $v^{(i)}$ (the node associated with status *Wanted*) is placed at the root and the insertion process for that node is abandoned. Once node $v^{(i)}$ is placed at the root, the DELETEMIN procedure resumes its operation. Finally, data fields *full_level* and *last_elem*, are modified only in the initialization phase of procedures INSERT and DELETEMIN. These additional operations can be implemented through a parallel-prefix operation, and therefore require $T_{ppf}(p)$ time. This additional cost does not affect (asymptotically) the time bounds of section 3.1. We can therefore obtain a result similar that of proposition 7 for the case that procedures INSERT and DELETEMIN operate concurrently. It should be noticed, however, that the cost of procedures INSERT and DELETEMIN can no longer be separated (due to the concurrent nature of the implementation and the synchronous nature of the BSP model), and therefore, we should consider, in each iteration, the maximum cost of the two procedures. Hence, we can no longer benefit from adapting the parameter d so as to select a data structure that fits the relative frequencies of the insert and delete-min operations. We thus limit the notion of d-ary heaps to that of binary ones. The main result is summarized in the following theorem [9]. A result similar to that of theorem 2 can then be established [9]. If we replace in the following result $o(.)$ and $\omega(.)$ by $O(.)$ the time bounds still hold.

Theorem 3 *For any constant $\rho > 0$, there exists a BSP algorithm for supporting in a pipelined fashion the PPQ operations* INSERT, FINDMIN, *and* DELETEMIN, *that for*

Table 1: Speedup for Randomized Selection.

Processors:		$p=2$			$p=4$			$p=8$		
Size (× 1000):		256	1024	4096	256	1024	4096	256	1024	4096
Machine	*Statistic*									
IBM SP2	1/2	1.9	1.9	1.9	3.7	3.7	3.8	5.3	6.6	7.2
	1/4	1.8	1.9	1.8	3.5	3.4	3.8	5.0	6.0	7.2
SGI	1/2	1.9	1.9	1.9	2.6	3.1	3.4	-	-	-
	1/4	1.8	1.9	1.9	2.4	3.0	3.5	-	-	-

all integers n, p, L, g, h, *and* $\bar{h}$ *such that* $\bar{h}n/p = \omega(\lg n)$, $L = o(hn/(p \lg p))$, *and* $g = o(\min\{h, n/p, hn/(p \lg n)\})$, *with probability at least* $1 - O(hn^{-\rho})$, *requires time,* $h(3(n/p) + o(n/p))$ *per* INSERT, $O(1)$ *per* FINDMIN, *and* $(h-1)(6(n/p) + o(n/p))$ *per* DELETEMIN *operation, where* h *is the depth of the underlying* $(n, 2)$*-BCH and* $\bar{h} = \min\{h, \lg n\}$.

4 Experimental Results

We briefly outline experimental results obtained for procedure SELECT using the Oxford BSP Library. Two statistics were selected, the $n/4$-th smallest and the median of a set of n keys. The speedup indicated in table 1 refers to the average speedup over a series of experiments on a 4-processor SGI Power Challenge shared memory system and an 8-processor IBM SP2 distributed-memory system. The input consisted of integers drawn uniformly at random and the input size varied from 256000 to 4096000 items.

References

[1] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18:155-193, 1979.

[2] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448-461, 1973.

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

[4] W. Cunto and J. I. Munro. Average case selection. *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing* pp. 369-375, 1984.

[5] R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *CACM*, 18:165-172, 1975.

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

[7] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP Model. To appear in *Proceedings of the 8-th ACM SPAA*, June 1996.

[8] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report, Computing Laboratory, Oxford University, May 1996.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Concurrent heaps on the BSP model. Technical Report, Computing Laboratory, Oxford University, June 1996.

[10] R. M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Symposium on theory of computing,* pp. 290-300, 1988.

[11] D. E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching.* Addison-Wesley, Reading, 1969.

[12] W. F. McColl. Scalable parallel computing: A grand unified theory and its practical development. In *Proceedings of IFIP World Congress*, 1:539-546, Hamburg, August 1994.

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

[14] M. C. Pinotti and G. Pucci. Parallel priority queues. *IPL,* 40:33-40, 1990.

[15] S. Rajasekaran. Randomized parallel selection. In *10-th Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 215-224, LNCS 472, Springer-Verlag 1990.

[16] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *6-th IEEE Symposium on Parallel and Distributed Processing,* pp. 97-103, 1994.

[17] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers,* C-37(12):1657-1665, 1988.

[18] P. Sanders. Fast priority queues for parallel branch-and-bound. In *Second International Workshop, IRREGULAR '95*, *LNCS* 980, pp. 379-393, Springer Verlag, 1995.

[19] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33:103-111, 1990.