Realistic Parallel Algorithms: Priority Queue Operations and Selection for the BSP* Model [†]

Armin Bäumker, Wolfgang Dittrich,
Friedhelm Meyer auf der Heide, and Ingo Rieping
{abk, dittrich, fmadh, inri}@uni-paderborn.de

Department of Mathematics and Computer Science and Heinz Nixdorf Institute
University of Paderborn, D-33095 Paderborn, Germany

Abstract. In this paper, we explore parallel implementations of the abstract data type priority queue. We use the BSP* model, an extension of Valiant's BSP model which rewards *blockwise communication*, i.e. sending a few large messages instead of many small ones. We present two randomized approaches for different relations between the size of the data structure and the number of parallel updates to be performed. Both yield work optimal algorithms that need asymptotically less communication than computation time and use large messages. All previous work optimal algorithms need asymptotically as much communication as computation or do not consider blockwise communication. We use a work optimal randomized selection algorithm as a building block. This might be of independent interest. It uses less communication than computation time, if the keys are distributed at random. A similar selection algorithm was independently developed by Gerbessiotis and Siniolakis for the standard BSP model. We improve upon previous work by both reducing the amount of communication and by using large messages.

1 Introduction

In this paper, we investigate parallel algorithms for maintaining priority queues. A priority queue is an abstract data structure that stores keys of a totally ordered set and supports insert and delete operations. We consider a parallel modification of these operations. *Insert* adds n keys to the data structure and *DeleteMin* extracts and removes the currently best m keys. Other operations could be considered but we only deal with these two operations which are typically used in branch-and-bound algorithms. N denotes the number of keys currently stored in the data structure.

We use the BSP* model, which was introduced in [3] as an extension of Valiant's BSP model [7]. This is a general purpose model for parallel computation, representing a bridge between powerful abstractions, such as the PRAM, and real machines based on specific architectures. The BSP* model differs from

[†] This research was partially supported by DFG-SFB 376 "Massive Parallelität" and EU ESPRIT Long Term Research Project 20244 (ALCOM-IT).

the original BSP by rewarding blockwise communication, i.e. it is worthwhile to combine data in order to communicate large messages instead of communicating many small ones. With respect to actual machines, this yields better use of the bandwidth of routers and reduces thereby the overhead involved in communication. A *BSP*(p, g, L, B) machine* consists of a set of p processors communicating through some interconnection medium. The computation is organized as a sequence of ***supersteps***. In a superstep the processors operate independently performing local computation and generating a number of point-to-point messages. At the end of the superstep, the messages are delivered to their destinations and a global barrier synchronisation is realized. If each processor performs at most t local operations and sends or receives at most h messages of maximum size s, the superstep requires $\max\{L, t\}$ computation time and $\max\{g \cdot h \cdot \lceil \frac{s}{B} \rceil, L\}$ communication time. Note that messages of size smaller than B are treated as if they were of size B. Hence, in the BSP* model it is worth communicating in a blockwise fashion.

Several parallel algorithms for maintaining priority queues are known in the literature. Deo and Prasad [5] present an optimal deterministic PRAM algorithm for insertion and deletion of p keys which requires time $O(\log N)$. Another PRAM algorithm reaching the same time bound is given by Pinotti and Pucci [9]. Ranade et al. [11] use a randomized approach on a d-dimensional array network. Inserting and removing of p keys with p processors can be done in time $O(p^{1/d})$, if $N = O(p^c)$ and c constant. For the coarse grained multicomputer model (CGM), Sanders [13] has developed a randomized algorithm using a similar approach. Insertion and deletion of p keys require $O(\log \frac{N}{p} + d(p))$ amortized time, if $N \geq p \log p$ and $d(p)$ denotes the diameter of the network. None of the above mentioned algorithms takes blockwise communication into account.

We present two randomized approaches to maintain a parallel priority queue on the BSP* model. Our first one is based on the algorithm of Sanders [13] and the second on the PRAM algorithm of Deo and Prasad [5]. Our algorithms are both work optimal and need asymptotically less communication than computation time. Further, we communicate only large messages. Both approaches distribute the N priority queue keys evenly among the processors, with high probability.

Result 1. *Let* $n = \Omega(p \log^2 n)$, $m = \Omega(p \log^2 m)$ *and* k *an arbitrary constant. Algorithm* Insert *needs* $O(g\frac{n}{pB} + L)$ *communication time with* $B \leq \frac{n}{p \log n}$ *and Algorithm* DeleteMin $O(g\frac{m}{pB} + (L + g)\log p)$ *communication time with* $B \leq \frac{m}{p}$. *The computation times are:*

computation time	Insert	DeleteMin
leftist heap	$O(\log \frac{N}{p} + \frac{n}{p} + L)$	$O(\frac{m}{p} \log \frac{N}{p} + L \log p)$
search tree	$O(\frac{n}{p} \log \frac{N}{p} + L)$	$O(\log \frac{N}{p} + \frac{m}{p} + L \log p)$
standard heap	$O(\frac{n}{p} \log \frac{N}{p} + L)$	$O(\frac{m}{p} \log \frac{N}{p} + L \log p)$

We reach these time bounds with probability at least $1 - 1/n^k$ *(Insert) and* $1 - 1/m^k$ *(DeleteMin).*

This result improves upon Sanders' algorithm by achieving blockwise communication. The second approach benefits from small values of N/n.

Result 2. *Let* β *be constant,* $n = \Omega(p \log^4 n)$, $N \leq \beta^n$ *and* n *sufficient large. For any constant* k *it holds: The algorithms* Insert *and* DeleteMin *need with probability at least* $1 - 1/n^k$ *computation time* $O(\frac{n}{p} \log \frac{N}{n} + L \log p)$ *and communication time* $O(g\frac{n}{pB} + \frac{g}{B}\sqrt{n/p} \log \frac{N}{n} + (L+g) \log p)$*, for* $B \leq \frac{1}{\log n}\sqrt{n/p}$.

We improve upon Deo and Prasad's algorithm by reducing the amount of communication and by communicating in a blockwise fashion.

Moreover, we present a selection algorithm which is crucial for the efficiency of our algorithms, and which is of interest on its own. Given a set X of n keys and a number $r \in \{1, ..., n\}$, selection determines the key with rank r from X.

Several optimal PRAM algorithms for selection exist. Cole [4] presents an optimal deterministic selection algorithm for the EREW PRAM requiring $O(n)$ work and $O(\log n \log^* n)$ time. Further, Rajasekaran [10] presents a selection algorithm for the hypercube which solves selection for n keys using $n/\log n$ processors in time $O(\log n)$. On the Distributed Memory Model (BDM), Bader and JáJá [1] developed a practical parallel selection algorithm which needs computation time $O(n/p)$ and communication time $O((\tau + p) \log \frac{n}{p^2} + n/p)$ if $n \geq p^2$, where τ is the latency. Recently, Gerbessiotis and Siniolakis [6] independently developed a randomized selection algorithm for the standard BSP model which is similar to our algorithm. They need $3n/(2p) + o(n/p)$ number of comparisons and asymptotically less communication than computation time for a wide range of BSP parameters. Again, none of the mentioned selection algorithms considers blockwise communication.

Our selection algorithm is based on the technique used by Rajasekaran. It is work optimal and improves upon previous work by reducing the communication time. If X is randomly distributed, it requires asymptotically less communication than computation time.

Result 3. *If* $n = \Omega(p \log^4 n)$ *holds and* k *is any constant, then Algorithm* Select *needs computation time* $O(n/p + L \log p)$ *and communication time* $O(\frac{g}{B}\sqrt{n/p} + (L+g) \log p)$ *for* $B \leq \sqrt{n/p}$*, with probability at least* $1 - 1/n^k$.

The paper is organized as follows: In Section 2 we describe the randomized selection algorithm. Section 3 presents the method for distributing the new keys among the processors used in the priority queue algorithms. Our two approaches to implement priority queues are given in Section 4 and 5. Most results are only sketched. Further explanations and some experimental results can be found in [12] and [2].

2 The Selection Algorithm

In this section, we describe our randomized selection algorithm *Select.* It determines, for a given integer r and a set X, $|X| = n$, the key with rank r from X. It is based on techniques previously developed by Rajasekaran [10]. Let the processors be divided in $\sqrt{p}$ *groups* of $\sqrt{p}$ processors each. For further application in priority queue operations we need that X is randomly distributed among the processors such that the following condition holds with probability at least $1 - 1/n^k$ for any constant k:

Condition 4. *a)* $O(\frac{|X|}{\sqrt{p}})$ *keys of set X are stored in each processor group.*

b) Each processor group stores $O(\frac{|S|}{\sqrt{p}} + \log n)$ *keys of a random subset S of X.*

This somewhat artificial condition is later needed in order to guarantee blockwise communication. It is fulfilled if we choose, for example, for each key a processor at random. Thus, *Select* may be applied in many situations.
Algorithm Select: Beginning with X, we recursively choose a small subset Y of X containing the key with rank r. We repeat this until Y has size $\sqrt{n}$. Then, the key with rank r is determined by sorting the $\sqrt{n}$ keys.

The crucial part is the way we choose set Y. Let X_ℓ be the subset of X considered in round ℓ. We use the technique over-sampling presented by Valiant and Gerbessiotis in [7] to determine a small subset Y of X_ℓ containing the key with rank r. Therefore, we choose a random subset S of X_ℓ of size $\sqrt{|X_\ell|}$, the samples, by marking every key with probability $1/\sqrt{|X_\ell|}$. We sort the sample set S. The samples $s_{\bar{s}i}$, $i \in \{1, ..., \bar{r}-1\}$, partition X_ℓ into $\bar{r}$ buckets of size $O(|X_\ell|/\bar{r})$ with high probability, for suitable $\bar{r}$ and $\bar{s}$. Thus, we can determine a constant number of buckets which contain the key with rank r with high probability. The keys $t_1 := s_{\bar{s}i}$ and $t_2 := s_{\bar{s}j}$ bound these buckets. The keys of X_ℓ which are greater than t_1 and smaller than t_2 form Y. We prove that with high probability Y is small ($\leq |X_\ell|^\alpha$ with $\alpha < 1$) and the key with rank r is in Y.

The crucial observation for the efficiency of this algorithm is the following. The algorithm needs only constantly many recursive rounds and the "random distribution" of X yields an almost even distribution of each subset of X among the processors. Therefore, all sets, we sort, are evenly distributed to the processors and thus the communication time is small.

Lemma 5. *For every $k > 0$, if n is sufficiently large, the following holds with probability at least $1 - 1/n^k$: (a) the set S has size of $\Theta(\sqrt{|X_\ell|})$, (b) the key with rank r is in Y and (c) $|Y| \leq |X_\ell|^\alpha$ with $\alpha < 1$ and α constant.*

Proof. Part (a) can easily be proved by Chernoff bounds. To prove (b) and (c) we use the following fact proved in [7]: If you have a chosen sample of size $\bar{s}\bar{r}$ of set X_ℓ and look at each s-th key r_i (with $r_i = s_{i\bar{s}}, i \in \{1, ..., \bar{r}-1\}$) of this sorted sample set, these keys separate set X_ℓ into buckets $B_i = \{x \in X_\ell | r_i < x \leq r_{i+1}\}$ $(r_0 = -\infty, r_{\bar{r}} = \infty)$ which are with high probability nearly of the same size $(\leq \lceil (1+\epsilon)(|X_\ell| + 1)/\bar{r} \rceil, \epsilon < 1)$. We have to choose $\bar{s}$ and $\bar{r}$ that $\bar{s}\bar{r} = \tilde{c}\sqrt{|X_\ell|}$ holds. □

The operations we use in Algorithm *Select* are Broadcast, Parallel Prefix (see [3]) and Ranking (modify a PRAM algorithm from Nassimi and Sahni [8]). With Cond. 4 and Lemma 5 we get the running times of Result 3.

Algorithm *Select* can easily be extended to an algorithm that works on d sets and searches one key in each set. We only have to execute every step for all d sets concurrently. Therefore, we can extend the block size and do not need more synchronizations.

3 The Block Distribution

In this section we want to solve the following distribution problem: *Input:* (a) Set M, distributed among the processors such that each subset of size n is distributed according to Cond. 4. (b) Set S of size n, distributed evenly among the processors. *Aim:* Distribute S among the processors such that each subset of size n of $M \cup S$ is distributed according to Cond. 4.

The following algorithm for this problem communicates the keys in a block-wise fashion, therefore we call it *block distribution.* With S_i we denote the subset of S of size n/p at processor P_i. Algorithm *block distribution (s)* works, for each S_i, as follows: (1) Choose randomly one out of s buckets for each key. (2) Store the keys belonging to the j-th bucket ($1 \leq j \leq s$) on a randomly chosen processor.

The following two lemmas show that Cond. 4 holds for the block distribution. Furthermore, the first one is used by the priority queue algorithms. By Lemma 7 we know that we need assumption $n = \Omega(p \log^4 n)$ to fulfil Cond. 4 in Result 3.

Lemma 6. *If $t > 0$, $n \geq ps \log t$, and k is an arbitrary constant, then with probability at least $1 - 1/t^k$: (a) the bucket size of an arbitrary bucket is $O(\frac{n}{ps})$ and (b) each processor has $O(\frac{t}{p})$ of t arbitrarily chosen keys from $M \cup S$, assuming additionally $t = \Omega(\frac{n \log t}{s})$.*

Proof. Part (a): Use Chernoff bounds. Part (b): We investigate how many keys a given processor P gets. The t arbitrary keys are distributed to at most t buckets B_i' by Step 1 of the block distribution. We conclude from part (a), that $|B_i'| = O(\frac{n}{ps})$, $\forall i \in \{1, ..., t\}$, with probability at least $1 - t/t^{k'}$ for any constant k'. Let $X_1, ..., X_n$ be independent random variables. Set $X_i = |B_i'|$, if B_i' is placed on processor P and $X_i = 0$ else. Let $X = \sum_{i=1}^{n} X_i$ be a random variable which represents the number of keys placed on processor P. We have $X_i \in \{0, ..., c\frac{n}{ps}\}$ for a constant c, $Prob(X_i = |B_i|) = 1/p$ and $EX = t/p$. By Hoeffding bound holds for a constant $\alpha \geq e$: $Prob(X \geq \alpha \frac{t}{p}) \leq \exp(-\frac{ts}{cn}) \leq t^{-k}$ with $t \geq \beta \frac{n \log t}{s}$ and constant $\beta \geq c \cdot k$. □

Lemma 7. *Let $n = \Omega(p \log^4 n)$, $s = \Theta(\sqrt{n/p} \log n)$, k an arbitrary constant and X an arbitrarily chosen subset of $M \cup S$ of size n. With probability at least $1 - 1/n^k$: (a) every processor group owns $O(n/\sqrt{p})$ keys of set X and (b) every processor group owns $O(\max\{\sqrt{n/p}, \log n\})$ keys of a random subset of X of size $O(\sqrt{n/p})$.*

Proof. We prove (a) in the same way as Lemma 6 (b). To bound the bucket size we use Lemma 6 (a) and set $t = n$, $n = \Omega(p \log^4 n)$ and $s = \Theta(\sqrt{n/p} \log n)$ to fulfil the assumptions of Lemma 6. Part (b): Chernoff bounds and (a). □

4 Algorithms for the Randomized Heap

The algorithms of this section are based on Sanders' heap algorithm [13]. In contrast to the algorithm of Sanders, which uses a standard heap, we use a search tree or a leftist heap as local data structure which either allow a faster extraction of the $O(m/p)$ smallest elements or a faster insertion of $O(n/p)$ new keys. Our algorithm also differs from Sanders' algorithm in the way we distribute the new keys. We use the *block distribution*, which allows to communicate the new keys blockwise.

Each processor P_i possesses its own local data structure. Now we present the algorithms *Insert* and *DeleteMin* without using a specific data structure.
Algorithm Insert: The n keys are distributed by block distribution and then inserted into the local data structure.
Algorithm DeleteMin: Each processor repeats removing from its local data structure $\alpha m/p$ keys until the altogether best m keys are taken. Then, the key with rank m from these removed keys is selected. The best m keys are distributed among the processors and the others are added back to the local data structures.

The following lemma tells us how keys are distributed to the processors and that the removal of $\alpha m/p$ keys, α constant, is sufficient to have the best m keys. Furthermore, it is easy to prove that every processor owns nearly the same number of the N priority queue keys.

Lemma 8. *Let $n \geq sp \log n$, $s \geq \log n$, $m = \Omega(\frac{n \log m}{s})$, $N = \Omega(n \log n)$ and k an arbitrary constant. For one insert operation, the following holds with probability at least $1 - 1/n^k$: (a) every bucket $S_{i,j}$ at the block distribution has a size of $O(\frac{n}{ps})$ and (b) every processor gets $O(s)$ buckets. (c) With the same probability every processor holds $O(N/p)$ of the total N priority queue keys and (d) with probability at least $1 - 1/m^k$ each processor holds $O(m/p)$ keys of the best m keys.*

Proof. Use Lemma 6 and Chernoff bounds. □

Now, we investigate the influence of different local data structures on the running times of our parallel priority queue. With Lemma 8 follows Result 1.

Leftist Heap: Beside Insert and DeleteMin the leftist heap supports the operation *Meld* which constructs a new leftist heap out of two leftist heaps, refer to [14]. Meld can be used in the operation *Insert.*

Search Tree: We use a search tree (e.g. a B-tree or Red-black-tree [14]) which supports the operations *Split* and *Join* beside the operations DeleteMin and Insert, where Split divides a search tree into two search trees and Join merges two search trees to one. One can use these operations in routine *DeleteMin* to access and to reinsert the keys.

5 Algorithms for the Parallel Heap

This section describes another randomized heap algorithm. It is based on a PRAM algorithm from Deo and Prasad [5] which uses a parallel data structure similar to the usual heap, the *parallel heap.* We introduce a novel way to map the parallel heap to the processors, to reduce the communication costs. This mapping is crucial for the efficiency of our algorithm and allows us to implement the PRAM algorithm of Deo and Prasad so that it needs asymptotically less communication than computation time.

A *parallel heap* is a binary tree, where each node stores n keys. A parallel heap which represents N keys has N/n nodes and height $O(\log \frac{N}{n})$. We now sketch the operations Insert and DeleteMin like they are presented by Deo and Prasad [5]. Then we point out how the keys are distributed among the processors and prove that this distribution works well.

Algorithm Insert: In order to insert n new keys we process the nodes along the *insert-path,* the path from root to the first node without keys at the leaf-level. At every level we have to merge the new keys with the appropriate node of the level.

Algorithm DeleteMin: Remove the n keys stored in the root and place the keys stored in the last node (a node of the leaf-level) in the root. Now, we let the keys move down the parallel heap. At every level we have to merge three nodes.

Insertion and removing are made in a pipelined fashion. Each new opeation modifies the root and afterwards the parallel heap is updated. After such an update, the root is up-to-date, so that the next operation can take place.

In the following we describe how we map the nodes of the parallel heap to the processors: Every node is divided into p buckets, where P_i owns bucket i, $i \in \{0, ..., p-1\}$. These buckets store the keys. The n keys of each node are distributed randomly (use block distribution, Sect. 3). The main idea is that after distributing the keys at random to the processors no key ever leaves its processor until it is removed from the heap by a DeleteMin operation. If a key (in bucket i) changes the node of the parallel heap, it does not change the processor. It is always stored in bucket i of an arbitrary node, thus on P_i. Every processor has a portion of every node of the parallel heap.

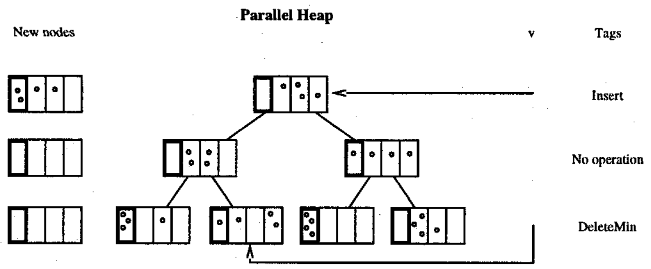

Fig. 1. Mapping of the parallel heap, including the buckets. The fat buckets are the ones stored in P_0.

To merge nodes of the parallel heap, we use algorithm Select (Sect. 2) instead of a merging algorithm. This guarantees the small amount of communication. Since we perform $O(\log \frac{N}{n})$ updates at the same time (because of pipelining) we select from $O(\log \frac{N}{n})$ sets, concurrently. For analyzing the computation and communication times it is necessary to show that each processor has nearly the same number of keys of every node and thus each processor has nearly the same number of keys of $\log \frac{N}{n}$ arbitrary nodes, each one chosen from a different level.

Lemma 9. *Let k and β be arbitrary constants, $n \geq sp\log n$, $N \leq \beta^n$ and $s = \Omega(\log n)$. Then the following holds with probability at least $1 - 1/n^k$: (a) every processor holds $O(\frac{n}{p})$ keys of an arbitrary node and (b) every processor holds $O(\frac{n}{p}\log\frac{N}{n})$ keys of $\log\frac{N}{n}$ arbitrarily chosen nodes.*

Proof. Part (a): Use Lemma 6 (b) with $t = n$, $s = \Omega(\log n)$ and the fact that no key leaves its processor. □

To prove Result 2, we use $s = c\sqrt{n/p}\log n$ with c constant, $n = \Omega(p\log^4 n)$ and Lemma 9. The least block size is given by algorithm *Insert*: $B \leq \frac{n}{ps} \leq \frac{1}{\log n}\sqrt{n/p}$.

References

1. D. A. Bader, J. JáJá. Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding and Selection. Technical report, Institute for Advanced Computer Studies and Department of Electrical Engineering, University of Maryland, 1995.
2. A. Bäumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic Parallel Algorithms: Priority Queue Operations and Selection for the BSP* Model. Technical Report University of Paderborn, 1996, http://www.uni-paderborn.de/cs/inri.html.
3. A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP model. Proc. of European Symposium on Algorithms, 1995.
4. R. J. Cole. An Optimally Efficient Selection Algorithm. Information Processing Letters, no. 26, pp. 295-299, 1987/88.
5. N. Deo, S. Prasad. Parallel Heap: An Optimal Parallel Priority Queue. Journal of Supercomputing, 1992, vol. 6, no. 1, pp. 87-98, 1992.
6. A. V. Gerbessiotis and C. J. Siniolakis. Deterministic Sorting and Randomized Median Finding on the BSP model. Proc. of the 8th Symposium on Parallel Algorithms and Architectures, 1996.
7. A. V. Gerbessiotis and L. Valiant. Direct Bulk-Synchronous Parallel Algorithms. Journal of Parallel and Distributed Computing, 1994.
8. D. Nassimi and S. Sahni. Parallel Permutation and Sorting Algorithms and a new generalized Connection Network. Journal of the ACM, 29:3, pp. 642-667, 1982.
9. M.C. Pinotti, G. Pucci. Parallel Priority Queues. Information Processing Letters, no. 40, 1991.
10. S. Rajasekaran. Randomized Parallel Selection. Foundations of Software Technology and Theoretical Computer Science, 10: pp. 215-224, 1990.
11. A. Ranade, S. Chang, E. Deprit, J. Jones, and S. Shih. Parallelism and Locality in Priority Queues. Proc. of the 6th IEEE Symp. on Parallel and Distributed Processing (SPDP), IEEE Society Press, 1994.
12. I. Rieping. Realitätsnahe parallele Priority Queues, Analyse und experimentelle Untersuchungen. Diploma Thesis, University of Paderborn, January, 1996.
13. P. Sanders. Fast Priority Queues for Parallel Branch-and-Bound. Workshop on Algorithms for Irregularly Structured Problems, LNCS, Lyon, 1995.
14. R. E. Tarjan. Data Structures and Network Algorithms. Society for Industrial and applied Mathematics, Philadelphia, Pennsylvania, 1983.