Randomized Priority Queues for Fast Parallel Access

Peter Sanders
Department of Computer Science
University of Karlsruhe, 76128 Karlsruhe, Germany
Tel: (49) 721 6084337; Fax: (49) 721 698675
E-mail: sanders@ira.uka.de

October 9, 1997

Abstract

Applications like parallel search or discrete event simulation often assign priority or importance to pieces of work. An effective way to exploit this for parallelization is to use a priority queue data structure for scheduling the work; but a bottleneck free implementation of parallel priority queue access by many processors is required to make this approach scalable. We present simple and portable randomized algorithms for parallel priority queues on distributed memory machines with fully distributed storage. Accessing $\mathrm{O}(n)$ out of m elements on an n-processor network with diameter d requires amortized time $\mathrm{O}\left(d + \log \frac{m}{n}\right)$ with high probability for many network types. On logarithmic diameter networks, the algorithms are as fast as the best previously known EREW-PRAM methods. Implementations demonstrate that the approach is already useful for medium scale parallelism.

1 Introduction

Load balancing is a key issue for parallelizing irregular problems. The matter is particularly complicated if the load units have different importance or priority. If this order is not closely adhered to, other load units may be more difficult to process or superfluous work may be necessary. One example for priorities are time-stamps in optimistic discrete event simulation [18, 8]. A sequential simulator processing events in time stamp order never has to perform a roll back. For parallel simulation this is not possible. But the closer the simulator adheres to the time-stamp order, the less likely are rollbacks.

Another important application is parallel (best-first) branch-and-bound. We will use it as an example throughout this paper because the effect of different variants of priority queues on the parallel performance is easy to analyze. The load units are nodes of a combinatorial search tree. The evaluation of a node N gives a lower bound on the cost of any solution obtainable from N. An efficient parallelization can (in general) only be guaranteed if the evaluation order of nodes closely adheres to a best first order (this will be made precise in Section 2.3).

Even if the sequential algorithm does not explicitly use prioritization, it often makes sense to use the sequential evaluation order itself as the priority for parallel execution. In [12, 13] this turns out to reduce (adverse) speedup anomalies in particular in presence of strong heuristics.

For all these applications, an attractive approach to parallelization is to manage a global priority queue for all "open" load units (e.g. events or search tree nodes). The processors access this data structure to remove one of the globally best load units, evaluate it and reinsert newly generated load units. In order to support this, fast parallel access by all processors should be possible. For large queues it is additionally desirable, to uniformly distribute the queue elements over the memory of all processors.

The remainder of this paper is structured as follows: Models for the parallel machine, the analysis of randomized algorithms, branch-and-bound and parallel priority queues are introduced in Section 2. Section 3 surveys previous results. The main body of this paper is Section 4; it describes and analyzes a randomized synchronous parallel priority queue which can be described quite concisely and which turns out to be asymptotically very efficient. These formal results are complemented by Section 5 where we present enhancements which make the algorithm more effective on contemporary machines with asynchronous behavior, preference for coarse grained communication and a high penalty for collective operations. Section 6 describes an implementation which demonstrates the practicability of bottleneck-free priority queues – even on coarse-grained moderately parallel machines like the IBM-SP. The conclusions in Section 7 summarize and discuss the results.

2 The Model

We first describe our model of a parallel machine in Section 2.1, and then define measures for randomized algorithms and some tools for their analysis in Section 2.2. Section 2.3 introduces the branch-and-bound application and Section 2.4 the notion of parallel priority queue needed in this paper.

2.1 The Parallel Machine

We consider an MIMD computer with n processing elements (PEs) which asynchronously interact by exchanging messages through a network. We start with algorithms written in a synchronized data parallel style but synchronizations will not be a bottleneck.

We do not assume a specific network topology. Instead, the algorithms are based on the following set of building blocks (if not otherwise stated we assume constant message lengths):

Routing Every PE either sends up to k messages to randomly determined receivers, or receives up to k messages from randomly determined senders

(k constant).

Broadcast One PE sends a message to all other PEs.

Reduction Given value v_i on PE i determine $v_0 \oplus v_1 \oplus \cdots \oplus v_{n-1}$ for an associative commutative operator $\oplus$.

Prefix For v_i and $\oplus$ as above determine $v_0 \oplus v_1 \oplus \cdots \oplus v_i$ on PE i.

Sort$^{\frac{1}{2}}$ Sort (or rank) a sample of $n^{\frac{1}{2}}$ values located on different PEs. (This can be implemented using a constant number of the above operations, e.g. [17, Section 2.2.2].)

The analysis can focus on finding the number of necessary basic operations. The execution time for a particular network is then easy to determine by multiplying the counts with the execution times $T_{\text{Routing}}(n)$, $T_{\text{Broadcast}}(n)$, $T_{\text{Reduction}}(n)$, $T_{\text{Prefix}}(n)$ and $T_{\sqrt{\text{Sort}}}(n)$. The results are also easy to translate into abstract models like LogP [5] or BSP [19] although this may be less accurate for some machines with tuned implementations for the above collective operations.

In order to simplify the discussion, we define a common upper bound T_{coll} such that $\left\{T_{\text{Routing}}(n), T_{\text{Broadcast}}(n), T_{\text{Reduction}}(n), T_{\text{Prefix}}(n), T_{\sqrt{\text{Sort}}}(n)\right\} \cup \mathrm{O}(\log n) \subseteq \mathrm{O}(T_{\text{coll}}(n))$ with high probability. Note that $T_{\text{coll}}(n)$ is proportional to the network diameter d for many network types, e.g., r-dimensional meshes, hypercubes and related constant degree networks (butterfly, perfect shuffle, ...) or a combination of a multistage network for routing and a tree network. All the necessary results can be found in [17].

2.2 Analysis of Randomized Algorithms

The analysis of the randomized algorithms described here is based on the notion of behavior *with high probability*. Among the various variants of this notion we have adopted the one from [11].

Definition 1. *A positive real valued random variable X is in $O(f(n))$* with high probability *– or $X \in \tilde{\mathrm{O}}\left(f(n)\right)$ for short – iff*

$$\forall \beta > 0 : \exists c > 0, n_0 > 0 : \forall n \geq n_0 : \mathbf{P}\left[X > cf(n)\right] \leq n^{-\beta} \ ,$$

i.e., the probability that X exceeds the bound f by more than a constant factor is polynomially small. In this paper, the variable used to express high probability is always n – the number of PEs.

One advantage of the high probability approach is that there are quite simple rules combining results about simpler problems into more complex results. In this paper we need the following rules which we present without proof because they are based on quite straightforward elementary probability theory. (Proofs can be found in [27].)

Lemma 1. *Let $X_1 \in \tilde{\mathrm{O}}(f_1)$, ... , $X_k \in \tilde{\mathrm{O}}(f_k)$ be random variables (k constant).*

$$\bigotimes_{i=1}^{k} X_i \in \tilde{\mathrm{O}}\left(\bigotimes_{i=1}^{k} f_i\right) \text{ for } \bigotimes \in \left\{\max, \sum, \prod\right\} \tag{1}$$

Lemma 2. *Let $\{X_1, \ldots, X_m\} \subseteq \tilde{\mathrm{O}}(f)$ be a set of identically distributed random variables (m at most polynomial in n).*

$$\max_{i=1}^{m} X_i \in \tilde{\mathrm{O}}(f) \tag{2}$$

$$\sum_{i=1}^{m} X_i \in \tilde{\mathrm{O}}(mf) \tag{3}$$

A keystone of many probabilistic proofs are the following *Chernoff bounds* which give quite tight bounds on the probability that the sum of 0/1-random variables, i.e., a binomial distribution, deviates from the expected value by some factor.

Lemma 3 (Chernoff bounds). *Let the random variable X represent the number of heads after n independent flips of a loaded coin where the probability for a head is p. Then [17]:*

$$\mathbf{P}\left[X \leq (1-\epsilon)np\right] \leq e^{-\epsilon^2 np/3} \text{ for } 0 < \epsilon < 1 \tag{4}$$

$$\mathbf{P}\left[X \geq (1+\epsilon)np\right] \leq e^{-\epsilon^2 np/2} \text{ for } 0 < \epsilon < 1 \tag{5}$$

$$\mathbf{P}\left[X \geq \alpha np\right] \leq e^{(1-\log\alpha)\alpha np} \text{ for } \alpha > 1 \tag{6}$$

(Throughout this paper log *denotes the natural logarithm.)*

2.3 Branch-and-Bound

We adopt the model of Karp and Zhang [14]. Let H denote the search tree with a set of nodes V. Node degrees are bounded by a constant. All node costs $c(v)$ are assumed to be different and $c(v)$ is monotonously increasing on any path from the root to a leaf. (When we compare nodes, we actually compare their costs.) We are looking for the leaf v^* with minimal cost. Let $\tilde{V} = \{v \in V : v \leq v^*\}$ and let $\tilde{H}$ be the subtree of H containing the nodes $\tilde{V}$. Let $m = |\tilde{V}|$ and h the length of the longest root-leaf path in $\tilde{H}$. Clearly $\tilde{V}$ is a set of nodes which have to be expanded by any algorithm which wants to find v^* together with a proof that there cannot be better solutions. We do not want to look at very small problems and therefore assume that $m \in \Omega(n \log n)$.

var $Q = \{\text{root node}\}$: Heap (* frontier set *)
var $c^* = \infty$ (* best solution so far *)
while $Q \neq \emptyset$ **do**
    select some $v \in Q$ and remove it
    if $c(v) < c^*$ **then**
        if v is a leaf node **then** process new solution; $c^* := c(v)$
        else insert successors of v into Q

Figure 1: Sequential branch-and-bound.

Most branch-and-bound algorithms can be viewed as a variant of the abstract sequential algorithm depicted in Figure 1. The processor cyclically selects a node v from the *frontier* set Q and expands it. A node with a higher value than the best one investigated so far cannot lead to a solution and is pruned. The remaining inner nodes are expanded and their successors are inserted into the frontier set. Leaf nodes are solution candidates. When Q becomes empty, we know that the best leaf node found so far is the solution. Expanding a node (i.e. generating its successors) takes time T_{x}; all other operations on nodes, node values and solutions take unit time.

If the best first selection strategy is used, i.e., Q is organized as a priority queue, the algorithm expands the (optimal) number of m nodes. Selection and insertion can be done in time $O(\log m)$, e.g., by using a binary heap[1] implementation of Q. So, for sequential best first branch-and-bound we get the execution time

$$T_{\mathrm{seq}} \in m(T_{\mathrm{x}} + O(\log m)) \ .$$

2.4 Parallel Priority Queues

The semantics of a sequential priority queue are quite simple: A set of *elements* with totally ordered *keys* is managed. The operation `insert` inserts an element. The operation `deleteMin` deletes the element with the smallest key and returns this element. For a parallel priority queue this is in general more difficult. Motivated by the application areas mentioned in the introduction, we concentrate on applications where there may be up to n `deleteMin` request and a possibly larger number of `insert` requests at once. We also exploit the fact that it does not really matter which of the smallest elements is assigned to which requestor. (For simplicity assume that all key values in the queue are different.[2]) We start with a simple data parallel variant and refer to Section 5.3 for a more flexible asynchronous semantics:

`insert*`: Each PE inserts up to k elements (k constant).

[1] There more sophisticated data structures for priority queues which only require constant time for some operations. However, for applications with about the same number of `insert` and `deleteMin` operations, these variants imply no asymptotic improvement.

[2] If necessary, this can be enforced transparently by appending a unique element identifier to the least significant bits of a key.

`deleteMin*`: The n smallest elements are retrieved and each PE gets exactly one of the elements. In order to avoid tedious discussions of special cases we assume throughout this paper that deleting elements from empty queues returns dummy elements with key ∞.

For example, using these operations we get the synchronous parallel global best first branch-and-bound algorithm described in [14] if we replace Q in Algorithm 1 by a parallel priority queue. The parallel algorithm expands n nodes in each iteration and the number of iterations required lies between $\max(\frac{m}{n}, h)$ and $\frac{m}{n}+h$, i.e., in $\Theta\left(\frac{m}{n}+h\right)$: The lower bound is easy to understand since every branch-and-bound algorithm must expand at least m nodes and has a sequential component of h node expansions. For the upper bound, if $|Q \cap \tilde{V}| \geq n$, all PEs get nodes from $\tilde{H}$. In all other iterations, the maximum path length to a leaf in $\tilde{H}$ is reduced.

3 Approaches to Parallelization

Perhaps the simplest implementation of a parallel priority queue is to use a centralized server PE which manages a sequential priority queue. In this case the operations `insert*` and `deleteMin*` require time $\Theta(n \log m)$ which is quite slow for large n. Also the memory of the server PE limits the size of the queue.

In principle this can be slightly improved using algorithms which are able to pipeline up to $\log m$ requests in such a way that an individual access takes constant time (e.g., [23]) but in practice this saving may be more than offset by a worsening of the communication bottleneck at the access point to the queue [24].

More scalable algorithms exploit the fact that our definition of parallel priority queue calls for a method to quickly remove a rather large number of elements at once. One approach is to use a generalized "k-bandwidth heap" which contains multiple elements in each heap node and to substitute the compare and exchange operations of the usual heap algorithm by parallel sorting and merging operations. n insertions and deletions can be performed in time $O(\log m)$ on EREW PRAMs [7] and on *pipelined hypercubes* [6]. (This algorithm requires newly inserted elements to be sorted, so we must add another $O(\log n \log\log n)$ or $\tilde{O}(\log n)$ term for the sorting operation.) The parallel sorting and merging routines required by these algorithms are slower on weaker models of parallel computation. On single-ported hypercubes and meshes, access times of $O(\log m \log n)$ respectively $O\left(\sqrt{n \log m}\right)$ have been achieved [10, 23].

A radical approach is to relax the priority queue semantics and to distribute the elements over more or less independent local queues which exchange elements in order to approximate the behavior of a global priority queue. For example, in the algorithm of Karp and Zhang [14, 22] newly inserted elements are sent to randomly selected PEs while `deleteMin` requests simply access the locally present queue. For branch-and-bound this only increases the number of expanded nodes by a constant factor (with high probability).

The starting point for this paper is to use Karp and Zhang's approach of local queues with random placement of elements for the `insert*` operation but to extract the *globally* best elements for a `deleteMin*` operation. It turns out that instead of parallel sorting and merging we now only need random routing and parallel selection. The basic approach has been independently developed in [28] and [23]. These simple versions are already asymptotically optimal on mesh connected machines. In [26] improvements for logarithmic diameter networks like butterflies are introduced by avoiding work imbalance due to local queue access and by modifying an efficient selection algorithm to exploit that the elements are randomly distributed. The present paper further refines this approach.

The same selection algorithm is also used in [9, 1] for accessing $k \gg n$ priority queue elements in parallel. These algorithms can be considered a combination of k-bandwidth heaps and random placement.

4 An Efficient Algorithm and its Analysis

We now describe and analyze an algorithm for synchronous parallel access to priority queues. The algorithm is designed to combine simplicity and analyzability with asymptotic efficiency. For a practical implementation some of the modifications described in Section 5 should be used.

The description starts with an overview of the basic algorithm in Section 4.1. We then elaborate probabilistic bounds on the sizes of some intermediate data structures in Section 4.2 which play a key role in the analysis of the algorithm. After investigating a nontrivial subroutine in Section 4.3 we can finally integrate the pieces into an analysis of the full algorithm in Section 4.4.

4.1 The Basic Algorithm

Figure 2 outlines the algorithms for `insert*` and `deleteMin*`. Every PE holds a local queue stored in two parts; a sorted array Q_0 which acts as a buffer for fast access to the smallest elements, and a heap Q_1 for the remaining elements. Let the $\breve{\cdot}$-accent denote the global union operation, e.g., $\breve{Q}_0$ denotes the union of all buffers. Every $\log n$ calls of `insert*` (we call this a *cycle*), the buffer Q_0 is emptied into Q_1. In Section 4.2 we show that this measure is sufficient to keep Q_0 small ($|Q_0| \in \tilde{\mathrm{O}}(\log n)$) most of the time.

The extraction of the n smallest elements proceeds in two phases. First, elements are moved from Q_1 to Q_0 until the smallest n elements must be in the buffers. This can be checked by counting the buffer elements smaller than the minimum element in $\breve{Q}_1$. Then, a distributed algorithm to be described in Section 4.3 finds the n smallest elements in $\breve{Q}_0$. These can be enumerated using a prefix operation. Finally, element number j is returned on PE j. Figure 3 sketches an example for the flow of data during the execution of `deleteMin*`.

var i: Integer (* call counter *)
var Q_0: Sorted Array (* buffer *)
var Q_1: Heap (* main queue *)

procedure `insert*`(e: Array of Element)
  send each e_i to a randomly selected PE
  insert received elements into Q_0
  $i := i + 1$; **if** $i \equiv 0 \bmod \log n$ **then** $Q_1 := Q_0 \cup Q_1$; $Q_0 := \emptyset$ **fi**

function `deleteMin*`() : Element
  while $|\{e \in \breve{Q}_0 : e < \min \breve{Q}_1\}| < n$ **do** $Q_0 := Q_0 \cup \{\text{deleteMin}(Q_1)\}$
  extract the n smallest elements in $\breve{Q}_0$
  enumerate the smallest elements $e_0, \ldots, e_{n-1}$
  send e_i to PE i and return it

Figure 2: Parallel priority queue access.

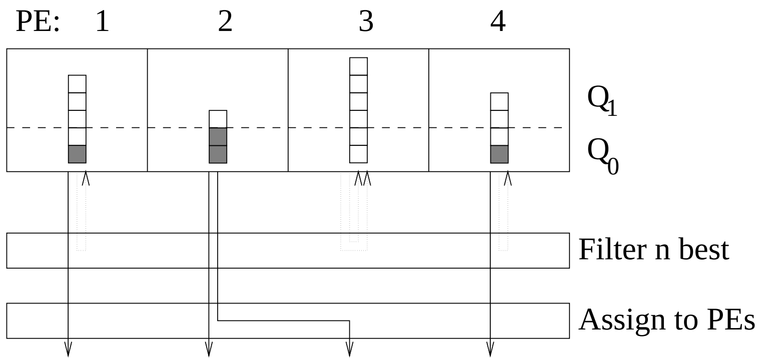

Figure 3: Example for extracting the $n = 4$ best elements.

4.2 Queue Sizes

Lemma 4. *The maximum number of new elements to be inserted into Q_0 on any PE after a call of* `insert*` *is in* $\tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right)$.

Proof. Let X_i denote the number of new elements to be inserted on PE i. Since the number of elements inserted by each PE is bounded by a constant, there is a constant k such that kn is a bound for the overall number of new elements. The placements of elements can be viewed as independent Bernoulli experiments with success probability $\frac{1}{n}$. (We count the placement of an element at PE i as a success). Therefore, we can apply the Chernoff bound (6): Let c be some constant we are free to choose.

$$\begin{aligned}\mathbf{P}\left[X_i \geq \frac{c\log n}{\log\log n}\right] &\leq \exp\left[\left(1 - \log\frac{c\log n}{k\log\log n}\right)\frac{c\log n}{\log\log n}\right]\\ &= n^{-c\left(1-\frac{\log\log\log n + \log(k/c)+1}{\log\log n}\right)} \leq n^{-\beta}\end{aligned}$$

for sufficiently large n and an appropriate choice of c. Using the maximum rule (2) we conclude that $\max_{i=0}^{n-1} X_i$ is also in $\tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right)$. □

Using similar techniques we can also derive the following bounds:

Lemma 5. *The maximum number of new elements to be inserted into Q_0 on any PE during a cycle is in $\tilde{\mathrm{O}}(\log n)$.*

Lemma 6. *The maximum number of elements moved from Q_1 to Q_0 on any PE during $\log n$ calls to `deleteMin*` is in $\tilde{\mathrm{O}}(\log n)$.*

Lemma 7. *At any moment,* $\texttt{globalMax}|Q_0| \in \tilde{\mathrm{O}}(\log n)$.

While the above lemmata are important to bound the running time of queue access, the following result establishes that the elements are very evenly distributed over the local memories.

Lemma 8. *At any moment,* $\texttt{globalMax}|Q_1| \leq \frac{m}{n} + \tilde{\mathrm{O}}\left(\sqrt{\frac{m}{n}\log n} + \log n\right)$.

Proofs of Lemmata 5–8. Similar to the proof of Lemma 4. For Lemma 5 we change the number of Bernoulli trials to $kn\log n$. For Lemma 6 observe that during $\log n$ calls the $n\log n$ globally smallest elements are removed. In the worst case they all have to be extracted from $\breve{Q}_1$. But since they have been placed randomly, no PE will have more than $\tilde{\mathrm{O}}(\log n)$ of them. Lemma 7 is a consequence of the Lemmata 5 and 6, and of the summation rule (1). For $m \in \mathrm{O}(n\log n)$ Lemma 8 can also be proved in an analogous way. Otherwise, let X_i denote the number of elements in Q_1 at PE i. We can use the Chernoff bound (5) to see that

$$\mathbf{P}\left[X_i \geq \frac{m}{n} + \sqrt{2\beta}\sqrt{\tfrac{m}{n}\log n}\right] = \mathbf{P}\left[X_i \geq \left(1 + \sqrt{2\beta\tfrac{n}{m}\log n}\right)\frac{m}{n}\right] \leq e^{-\frac{2\beta n\log n}{2m}\cdot\frac{m}{n}}$$

$= n^{-\beta}$ for sufficiently large n. The remainder of the argument is again analogous to the proof of Lemma 4. □

4.3 Extracting Elements

Assume we want to remove the n' smallest elements from $\breve{Q}_0$. (In our case $n' = n$.) Figure 4 lines out an efficient probabilistic algorithm for doing this. We maintain a candidate set Q' (initially Q_0) and a set of elements Q_{out} known to belong to the smallest ones. m' is the size of $\breve{Q}'$. The algorithm is related to the well known sequential quicksort-like median selection algorithm [4, Section 10.2]. However, instead of choosing a single pivot for partitioning the remaining candidates, we try to partition Q' into three parts; elements which are certain to belong to the smallest elements, those which are certainly *not* among the smallest ones, and a (hopefully small) set of remaining candidates for the next iteration. (Very similar algorithms are also described in [25, 21].)

First, a random sample of size $n^{\frac{1}{2}}$ is selected. (It simplifies the analysis to assume that this is done with replacement, i.e., elements may be selected for multiple samples). We then rank the sample elements and choose two pivots u and l, spaced by $n^{\frac{1}{4}+\epsilon}$ from an estimate for an element with rank close to n'. $0 < \epsilon < \frac{1}{4}$ is a small constant we are free to choose. We argue that with high probability at least n' elements will be smaller than u such that elements larger than u can be excluded from consideration and that there are no more than n' elements smaller than l such that those that *are* smaller can savely be selected. This randomized partitioning process is repeated until a trivial case occurs.

Lemma 9. *There is a choice of $\epsilon > 0$ such that the number of partitioning iterations is in $\tilde{\mathrm{O}}(1)$.*

We omit the proof which is again based on Chernoff bounds and on the fact that for an appropriate choice of ϵ the number of elements between l and u is small with high probability. Very similar arguments can also be found in [25, 21].

Lemma 10. *Extracting the n smallest elements from Q_0 takes time $\tilde{\mathrm{O}}(T_{\mathrm{coll}})$.*

Proof. An iteration involves the following communication operations:

- A prefix sum for enumerating the elements in Q'.
- Routing randomly selected samples to different PEs. Since the elements have been placed randomly, the samples will be evenly distributed over the PEs such that routing them is possible in time $\tilde{\mathrm{O}}(T_{\mathrm{coll}})$.

forall PEs **dopar** synchronously
  $Q' := Q_0$ (* solution candidates *)
  $m' := |\breve{Q}'|$ (* number of remaining candidates *)
  $Q_{\mathrm{out}} := \emptyset$ (* elements already found *)
  while $n' > 0 \wedge m' > n^{\frac{1}{2}} \wedge m' > n'$ **do**
    randomly choose $n^{\frac{1}{2}}$ samples from $\breve{Q}'$ with replacement
    let s_i denote the the i-th smallest sample (* sort them *)
      ($s_i = -\infty$ for $i \leq 0$, $s_i = \infty$ for $i > n^{\frac{1}{2}}$)
    $u := s_{\frac{n'}{m'}n^{\frac{1}{2}}+n^{\frac{1}{4}+\epsilon}}$ (* upper pivot *)
    $l := s_{\frac{n'}{m'}n^{\frac{1}{2}}-n^{\frac{1}{4}+\epsilon}}$ (* lower pivot *)
    if $|\{v \in \breve{Q}' : v < u\}| \geq n'$ **then** $Q' := Q' \setminus \{v \in Q' : v \geq u\}$
    if $|\{v \in \breve{Q}' : v \leq l\}| \leq n'$ **then** $Q' := Q' \setminus \{v \in Q' : v \leq l\}$
      $Q_{\mathrm{out}} := Q_{\mathrm{out}} \cup \{v \in Q' : v \leq l\}$; $n' := n' - |\{v \in \breve{Q}' : v \leq l\}|$**fi**
    $m' := |\breve{Q}'|$
  if $m' \leq n'$ **then** $Q_{\mathrm{out}} := Q_{\mathrm{out}} \cup Q'$
  else if $n' > 0$ **then** (* $m' \leq n^{\frac{1}{2}}$; sorting is easy now *)
    sort $\breve{Q}'$ and insert the n' smallest elements into their local Q_{out}

Figure 4: Finding the n smallest elements from $\breve{Q}_0$.

- Sorting the samples.
- A constant number of reductions and broadcasts for counting elements and disseminating pivots.

All other operations are performed locally. By representing Q' and Q_{out} as two indices into the sorted array Q_0, the local operations can be performed in time $\mathrm{O}(\log \texttt{globalMax}|Q_0|) \subseteq O(\log m') \subseteq O(\log n)$ using binary search. All operations inside the loop can be performed in time $\tilde{\mathrm{O}}\,(T_{\text{coll}} + \log n) \subseteq \tilde{\mathrm{O}}\,(T_{\text{coll}})$. The same is true for the operations outside the loop. Since the number of iterations is in $\tilde{\mathrm{O}}\,(1)$ we can conclude that the total time for selection is in $\tilde{\mathrm{O}}\,(T_{\text{coll}} + \log n) \subseteq \tilde{\mathrm{O}}\,(T_{\text{coll}})$ (using the product rule (1)). □

4.4 Putting the Pieces Together

We now have all the required results to come back to the analysis of Algorithm 2.

Theorem 1. *A call to* `insert*` *or* `deleteMin*` *requires amortized time in* $\tilde{\mathrm{O}}\left(T_{\text{coll}} + \log \frac{m}{n}\right)$ *(amortized over* $\log n$ *calls).*

Proof. We look at the total execution time for $\log n$ calls of both functions. Since $\texttt{globalMax}|Q_0| \in \tilde{\mathrm{O}}\,(\log n)$ (Lemma 5), emptying Q_0 into Q_1 takes time $\tilde{\mathrm{O}}\left(\log \frac{m}{n} \log n\right)$. Due to Lemma 6, there are only $\tilde{\mathrm{O}}\,(\log n)$ iterations of the while loop (summed over $\log n$ calls) for moving elements from Q_1 to Q_0 each of which involves two reductions and one priority queue access (time $\mathrm{O}\left(T_{\text{coll}} + \log \frac{m}{n}\right)$).

The remaining operations have to be counted for each call (i.e. $\log n$ times): Extracting the n smallest elements takes time $\tilde{\mathrm{O}}\,(T_{\text{coll}})$ (Lemma 10), and assigning these elements to the PEs can be done using a prefix sum and a routing operation (time $O(T_{\text{coll}})$). $\tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right)$ new elements (for each call of `insert*`) (Lemma 4) can be inserted into Q_0 in time $\tilde{\mathrm{O}}\,(\log n)$ by first sorting them (e.g. by heap-sort) and then merging them with Q_0. Summing all this together (Relations (1,3)) yields time $\tilde{\mathrm{O}}\left(\left(T_{\text{coll}} + \log \frac{m}{n}\right) \log n\right)$. □

This result can for example be used to devise a parallel branch-and-bound algorithm which is provably efficient if $T_{\text{x}} \in \Omega\left(T_{\text{coll}} + \frac{m}{n}\right)$:

Theorem 2. *The parallel execution time of synchronous best first branch-and-bound using algorithm 2 is*

$$T_{\text{par}} \in \left(\frac{m}{n} + h\right)\left(T_{\text{x}} + \tilde{\mathrm{O}}\left(T_{\text{coll}} + \log \frac{m}{n}\right)\right) \quad .$$

Proof. $\frac{m}{n} + h$ iterations are sufficient to complete the search. For m polynomial in n the theorem is an immediate consequence of Theorem 1 and the summation rule (3). Else, we must additionally exploit that the execution times of different cycles are independent of each other because a cycle deterministically removes the $n \log n$ smallest elements without moving the other elements. Therefore, with high probability, only a polynomially small fraction of the cycles will require a time exceeding the limit from Theorem 1. These "slow" cycles cannot significantly change the total execution time because the worst case time for a cycle involves only a polynomial number of operations. □

5 Refinements

Algorithm 2 is mainly designed for simplicity and analyzability. We now describe improvements which might be useful for a real implementation. After introducing some simple enhancements, in Section 5.1 we show how the number of communication operations can be reduced at the cost of increasing message sizes in Section 5.2, and we explain how an asynchronous machine can be used more efficiently in Section 5.3. In Section 5.4 we line out how the algorithm can be implemented efficiently on shared memory machines.

5.1 Simple Enhancements

As a first improvement note that the loop in Algorithm 2

$$\textbf{while } |\{v \in \breve{Q}_0 : v < \min \breve{Q}_1\}| < n \textbf{ do } Q_0 := Q_0 \cup \{\text{deleteMin}(Q_1)\}$$

yields two kinds of information that can be used to speed up the subsequent extraction process. While the condition is true, all elements smaller than $\min \breve{Q}_1$ can savely be selected. When it is false, elements larger than $\min \breve{Q}_1$ can be excluded from consideration for this iteration.

Queue maintenance costs can be reduced by using the *leftist tree* variant for representing Q_1 and Q_0 [15]. Emptying Q_0 into Q_1 can then be performed in time $O(\log m)$ by merging the two trees. In addition, those elements which are immediately fetched back into Q_1 when `deleteMin` is called after the end of a cycle, can be retrieved in constant time using this representation.

Furthermore, if sending elements involves long messages we will only put the key and a reference to the element in Q. This reduces the number of times, an element is moved from two to one. In many branch-and-bound applications, most inserted elements are never actually expanded and do not need to be moved at all.

5.2 Using Coarser Grained Communication

Algorithm 2 employs a relatively large number of fine-grained communication operations. On many contemporary machines, it is better to use more coarse grained but fewer operations. Starting with some simple measures to decrease the number of collective operations we go on to a new very simple algorithms for parallel selection and finally look at the consequences of accessing much more than n elements at once.

Coalescing Loop Iterations

We can coalesce several iterations of the while loop for moving elements from Q_1 to Q_0 into two vector operations for minimum determination and counting.

At the cost of increasing the candidate set for selection, we can even completely eliminate the above while loop: Since the n best elements are randomly placed, we know that considering the first $\mathrm{O}\left(\frac{\log n}{\log\log n}\right)$ elements of Q_0 is sufficient to catch all the n best elements with high probability. The rare cases when this is insufficient can be caught later on by checking on each PE wether the result of the selection routine is smaller than the smallest remaining local element. In this case an exception can be raised, triggering a fallback implementation which can be arbitrarily slow if it is needed sufficiently rarely. For example, in our implementation we looked at the $\min(n, 2\frac{\log n}{\log\log n} + 10)$ locally smallest elements and using the Chernoff bound (6) it is easy to show that the probability of a buffer overflow on any PE is below $2 \cdot 10^{-9}$ for all n.

Similarly, the number of iterations of the selection procedure 4 can be reduced by sorting a larger sample or using more than two pivots at a time. A particularly interesting scheme for choosing pivots is to use $u_i = s_{\frac{n'}{m'}n^{\frac{1}{2}}+2^i}$, $l_i = s_{\frac{n'}{m'}n^{\frac{1}{2}}-2^i}$ $(i = 1, \ldots, \log n/2)$. The number of pivots is quite moderate and the pivots lie very dense around the estimate for the n'-th smallest element. Also, there are no tuning parameters (like ϵ) which we would have to adapt for maximal performance.

Selection with a Single Reduction

For moderate n or machines with very coarse grained communication characteristics we can use a very simple approach to perform the selection with a single collective communication: Perform a reduction over Q_0 using `merge` as the associative commutative operator. The root PE will then receive Q_0 in sorted order and simply needs to broadcast the n-th smallest element.

This rather crude approach can be improved by once again exploiting the fact that the elements are allocated randomly. It is rather improbable that the n-th smallest element is ever very far away from the n'-th position in a subsequence based on the data from n' PEs. Therefore, we can use the function `prunedMerge` depicted in Figure 5. It is described for the general case of selecting the k-th smallest out of m randomly allocated and locally sorted elements based on sequences of size $2k' + 1$.

The function processes two 5-tuples $(s_j, x_j, \Delta_j, y_j, n'_j)$ $(j \in \{0, 1\})$ containing the sequences to be merged, the largest element omitted as too small, the number of elements omitted as too small, the smallest element omitted as too large and the number of processors from which the data has been collected. The two sequences are merged and the position i of the k-th smallest element is estimated based on the available information. If this information comprises all n processors (at the root of the reduction tree) $s[i]$ will be the correct answer if $x \leq s[i] \leq y$.

function select(k, m, k', s)
(s, x, Δ, y, n') := globalReduce(prunedMerge(k, m, k', ...),
prune(k, m, k', $(s, -\infty, 0, \infty, 1)$))
if $0 \le k - \Delta \le 2k' + 1 \wedge x \le s[i] \le y$ **then return** $s[k - \Delta]$
else return slowSelect(k, m, s)

function prunedMerge(k, m, k', $(s_1, x_1, \Delta_1, y_1, n'_1)$, $(s_2, x_2, \Delta_2, y_2, n'_2)$)
return prune(k, m, k',
(merge(s_1, s_2), $\max(x_1, x_2)$, $\Delta_1 + \Delta_2$, $\min(y_1, y_2)$, $n'_1 + n'_2$))

function prune(k, m, k', (s, x, Δ, y, n'))
i := round $\left(k\frac{n'}{n}\right) - \Delta$
... **(*** omitted cases where not both sides of the sequence need pruning ***)** ...
return $(s[i - k', \dots, i + k'], \max(x, s[i - k' - 1]), \Delta + i - k',$
$\min(y, s[i + k' + 1]), n')$

Figure 5: Selection using a merge operator which throws away the largest and smallest elements.

The following lemma answers the question how large the sequence size k' needs to be in order to make a successful selection highly probable. (We can use a slow fallback implementation if the selection fails.)

Lemma 11. *It suffices to choose* $k' \in \mathrm{O}\left(\max(\sqrt{k \log n}, \log n)\right)$ *in order to make selection with pruned merge succeed with high probability.*

Proof. Let $e_1, \dots, e_m$ denote the considered element. Let X denote the number of elements $e_i \le e_k$ allocated to the n' processors under consideration and consider some $\beta > 0$. If the selection fails, there must be at least one call to `prune` where s still contains e_k but at a position where it is thrown away. We now prove that the probability for this event is polynomially small for any of the $2n - 1$ calls to `prune` – regardless of the order in which the reduction is performed. Hence, the total failure probability is also polynomially small.

Case 1: e_k is thrown away as too small. This implies that $X < k\frac{n'}{n} - k' = (1 - \frac{nk'}{n'k})k\frac{n'}{n})$. The probability of this event can be bounded using Chernoff bound (4):

$$\mathbf{P}\left[X < k\frac{n'}{n} - k'\right] \le e^{-\frac{n^2 k'^2 kn'}{3n'^2 k^2 n}} = n^{-\frac{k'^2 n}{3kn' \log n}} \le n^{-\beta}$$

if $\frac{k'^2 n}{3kn' \log n} \ge \beta$. This is equivalent to $k' \ge \sqrt{3\beta k\frac{n'}{n} \log n} \le \sqrt{3\beta k \log n}$.

Case 2: e_k is thrown away as too large.

Case 2.1: If $k' < k\frac{n'}{n}$ we can use Chernoff bound (5) in a way analogous to Case 1.

Case 2.2: If $k' \geq k\frac{n'}{n}$ but $n' \geq \frac{2\beta n \log n}{\epsilon^2 k}$ for some $\epsilon < 1$ we can also use bound (5) by estimating $\mathbf{P}\left[X > k\frac{n'}{n} + k'\right] \leq \mathbf{P}\left[X > (1+\epsilon)k\frac{n'}{n}\right]$.

Case 2.3: Otherwise, a rather crude estimation using bound (6) suffices.

$$\mathbf{P}\left[X > k\frac{n'}{n} + k'\right] \leq \mathbf{P}\left[X > k'\right] \leq e^{\left(1-\log\frac{k'n}{n'k}\right)k'} \leq n^{-\beta}$$

for $\left(1 - \log\frac{k'n}{n'k}\right)k' \leq \beta \log n$. For $n' < \frac{2\beta n \log n}{\epsilon^2 k}$ it suffices to have $\left(\log\frac{k'\epsilon^2}{2\beta \log n} - 1\right)k' \geq \beta \log n$ and this is true if $k' \geq \frac{2e^2\beta}{\epsilon^2}\log n \in \mathrm{O}(\log n)$. □

Lemma 11 can be applied to the selection problem in `deleteMin*` by setting $k = n$. Note that for the case that we apply selection to the $\mathrm{O}\left(\frac{\log n}{\log\log n}\right)$ best elements on each PE these are not placed independently. But the entire set of m elements was placed independently and the n best elements are among the considered elements with high probability, so we will get the same result.

Corollary 1. *Selecting the best n elements in* `deleteMin*` *can be done using a single reduction with input data of length* $\mathrm{O}\left(\sqrt{n}\log n\right)$.

This is asymptotically not as efficient as the more complicated algorithms but for many practical machines it will be faster. It is also a considerable improvement over the unpruned merge.

Batched Operation

The number of communications per element can be further decreased by inserting and deleting larger batches of elements at once. This is quite straightforward for `deleteMin*`. In [1] it is proved, that for inserting $k = \Omega\left(n(\log n)^2\right)$ elements it is not necessary to place them completely independently. Rather, it suffices to randomly put them into $\Omega(\log n)$ "containers" and to place the containers randomly. In addition, by using a generalized "$\frac{k}{n}$-bandwidth" heap which stores $\frac{k}{n}$ elements in every node, the local queue access can be made more efficient [1, 9].

For the class of applications outlined in the introduction, using large batches is equivalent to emulating k logical PEs on each physical PE. This only works if there is sufficient parallelism in the problem instance. Furthermore, the required work can increase. For example, for branch-and-bound the number of expanded nodes can grow from $\Theta\left(\frac{m}{n} + h\right)$ to $\Theta\left(\frac{m}{n} + kh\right)$.

5.3 Asynchronous Operation

For many applications of the type mentioned in the introduction, synchronized parallel calls of `deleteMin*` are not quite adequate. The time required to process a load unit will vary. Therefore it would be wasteful to wait until all n PEs have completed their work before a new queue access is started. One way to solve this problem is to separate priority queue server threads from client threads doing the real work. The server threads could start a collective queue access whenever ϵn client threads have sent a request. Depending on the machine architecture, the server threads could run on the same PE as the clients, on communication coprocessors or on a smaller set of dedicated server PEs.

However, the above solution implies a delicate tradeoff between long waiting times for large ϵ and high communication overhead for small ϵ. It is more elegant to always have some of the smallest queue elements in reserve which can be retrieved completely asynchronously using a distributed FIFO queue. Such a data structure can be implemented in a bottleneck-free way using parallel counting algorithms (e.g. [30]). When (or even before) this reserve is exhausted a new batch of (say n) elements is retrieved.

A price we pay for this approach is that the elements in the reserve might already be outdated when they are retrieved since smaller elements have been inserted in the meantime. But a closer consideration shows that this is not really a new problem. No priority queue which allows parallel insertion can guarantee that every element passed to `insert` will instantaneously become visible everywhere. Infact, if we use a centralized queue, it may take time in $\Omega(n \log m)$ before an element is inserted. This is much more than the delay incurred by holding a reserve. The faster our routines, the better can we approximate the behavior of a sequential algorithm.

The maximum time it takes until new elements become visible can be bounded by periodically invalidating and recomputing the reserve. Invalidation need not be triggered as long as no elements smaller than the largest reserve element are inserted. Invalidation can be triggered ealier then prescribed by the timeout when more than a certain number of elements smaller than reserve elements have been inserted. Since such a trigger point need not be 100 % accurate we can also use approximated triggers as described in [27] which incur very low overhead.

Asynchronous access implies a number of additional benefits:

- The total work required for inserting or deleting $O(n)$ elements is in $O\left(n \log \frac{m}{n}\right)$ so for m not too large, most server threads spend most of their time waiting for the completion of collective communication primitives. This time can be used by client threads.

- We do not need to determine exactly the n-th smallest element e_n during selection since the size of the reserve is flexible. So we could break the loop in Algorithm 4 after a single iteration or invent a variant of Algorithm 5 which does not keep track of all elements around the estimated position of e_n but only of a sample which gets the sparser the farther away it is from the estimated position.

5.4 Exploiting Shared Memory

In the architecture of parallel computers there is a trend to support shared memory (with nonuniform memory access costs (NUMA)) even for quite large n. Our algorithm does not require shared memory but it can profit from the fact that remote memory accesses are often much faster than sending and receiving small messages.[3] In particular, delivery of result elements in the `deleteMin*` operation can be replaced by a single remote write. The collective communications can also profit if there is an efficient way to wait for the arrival of a remote write. (Busy waiting is one possibility but this is expensive for the asynchronous algorithms of Section 5.3.) The `insert*` operation is more complicated because multiple elements can queue up at a PE; but some machines support an efficient `fetch-and-increment` operation which can be used on the element counter of the target PE. Hardware support for this operation is also very useful for maintaining the FIFO queue required by the asynchronous algorithm.

6 Implementation

In order to get an idea how practical the bottleneck free priority queues are on contemporary machines, we have implemented the algorithm in a portable way using the library MPI (Message Passing Interface [31]). Since a quite large message startup overhead is common, it was clear from the start that the cost of local queue access was not so important. Also, the size of the queue elements was bound to have little influence (besides limiting the maximal size of the queue). So, we used integer keys and integer data fields for the measurements. On the other hand, the number of communication operations had to be minimized. Therefore, a simplified algorithm was implemented which keeps the size of Q_0 below $\min(n, 2\frac{\log n}{\log\log n} + 10)$ all the time. Furthermore, the reduction based selection method from Section 5.2 was used. So a `deleteMin`-operation boils down to one call of `MPI_Reduce` and one call of `MPI_Bcast` for the selection, a call to `MPI_Scan` for enumerating the best elements, $\tilde{\mathrm{O}}\left(\frac{\log n}{\log\log n}\right)$ sends for delivering the results and a single `MPI_Recv`. The complete source code is available electronically at `http://liinwww.ira.uka.de/~sanders/ppq/`.

Figure 6 shows the median[4] execution times for one collective call to `deleteMin*` plus one call to `insert*` inserting n elements into a queue of size 2^{24} distributed over n processors of the IBM-SP at the University of Karlsruhe.[5] A centralized implementation employing a dedicated server processor receiving

[3]Partly, because there are few commercial machines with sufficient hardware support for message passing. (With the notable exception of the *shared memory* machine Cray T3E [29].)

[4]Unfortunately we cannot give the *average* execution time since at the time of the measurements the machine was operated in such a way that some huge delays due to external reasons occured. But the results on the T3D show that the median is within one percent of the average value under normal conditions.

[5]I would like to thank the computing centre and its staff for making available 64 “Wide Nodes” (with 77 MHz clock) connected by a “High Performance Switch 3”. This required a special setup wich could not be repeated for the measurements of the centralized algorithm. But for n not too small, the measured data is modeled quite reliably by a linear function in n.

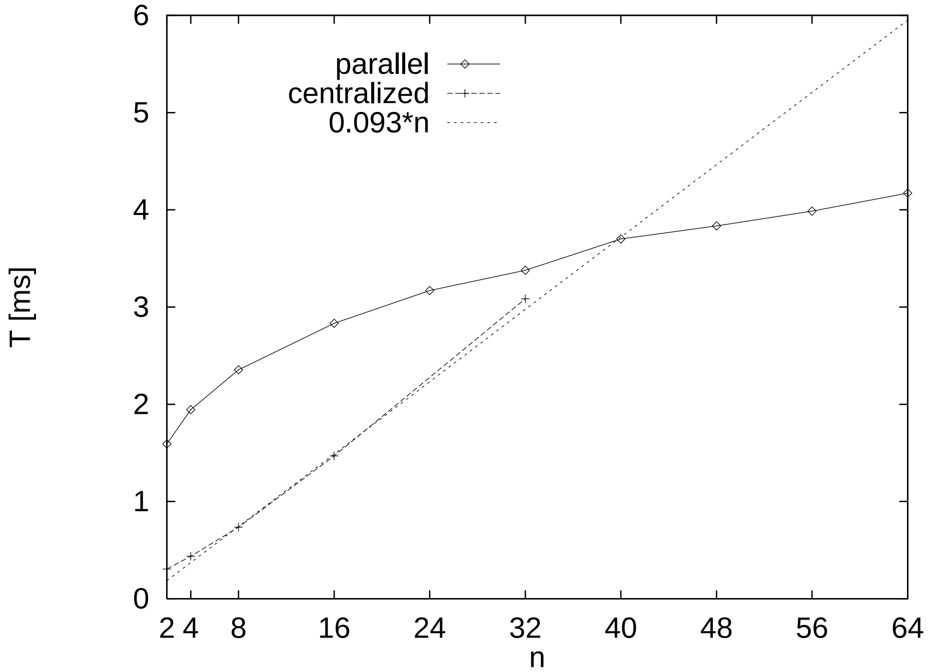

Figure 6: Median execution times for inserting plus extracting n elements on an IBM-SP with n nodes (based on 1000 repetitions).

requests using `MPI_Waitsome` needs about 93 μs per queue access for large n. So, for more than about 40 PEs the new algorithm is faster. More than half of the execution time is consumed by the reduction and scan operations. Most of the rest is due to the other communication operations. A local queue access takes about 7 μs for $m = 2^{24}$, so it does not influence the execution time very much.

Both MPI-programs were also executed on a Cray T3D. For $n = 256$ and $m = 2^{24}$ the parallel priority queue needs an average of 3.73 ms for inserting plus deleting n elements. This is about 7.5 times faster than the centralized code (with $m = 2^{21}$) which needs about 110 μs for inserting plus deleting one element, (i.e. 28.16 ms for 256 elements). We did not have the opportunity to run a complete measurement series on the T3D but even if the parallel program would not be faster for smaller n, the break-even point where the parallel algorithm is faster than a centralized queue would lie below 34 on the T3D. The histogram in Figure 7 for the run times of 1000 calls to `insert*` plus `deleteMin*` demonstrates that substantial variations are very rare.

7 Conclusions

Parallel priority queues based on random placement of elements and parallel selection are very efficient for parallel access by all processors as it is needed for load balancing applications. On powerful models of computation like EREW-PRAMs the algorithm is as efficient as previously known but more complex

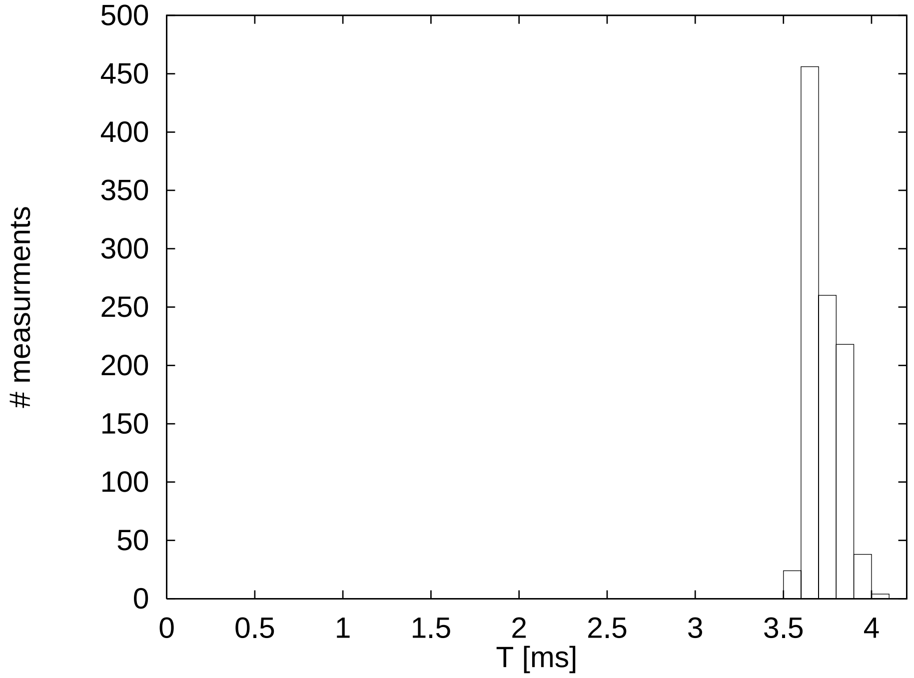

Figure 7: Execution time distribution for an `insert*-deleteMin*` call on a Cray T3D with $n = 256$, $m = 2^{24}$.

algorithms. More importantly the algorithm also performs well on networks of processors. Both `insert*` and `deleteMin*` of n out of m elements can be performed in time $\tilde{\mathrm{O}}\left(d + \log \frac{m}{n}\right)$ on most usual networks with n PEs and diameter d. In particular, on (single ported) hypercubes; constant degree hypercubic networks like butterflies, CCC, perfect shuffle or DeBruijn; and combinations of multistage networks with tree networks, the execution time is in $\tilde{\mathrm{O}}\left(\log n + \log \frac{m}{n}\right)$ which improves all previously known algorithms.

The algorithm is also of practical interest since it outperforms the centralized approach for $n > 40$ on mainstream parallel computers. Previous implementations could only clearly demonstrate this for batched access to $k \gg n$ elements (using a similar approach). In part, this success is due to the new selection algorithm based on a single reduction.

Nevertheless, the asymptotically more efficient selection algorithm from Section 4.3 is of independent interest since with its execution time in $\tilde{\mathrm{O}}\left(T_{\mathrm{coll}}\right)$ for randomly placed data it beats the worst case lower bound for deterministic algorithms given in [20].

Future Work

An efficient and portable implementation of asynchronous parallel priority queues raises some interesting questions. On the implementation side, there is some hope that in the near future, efficient implementations of MPI combined with POSIX Threads will be available. This might be the right platform for developing algorithms along the lines of Section 5.3.

An interesting topic that was not relevant for the present implementation is the question which sequential priority queues should be used as a basis [16, 2]. The local queue implementation might become relevant if more efficient communication interfaces are used (which are currently only available as proprietary systems on a few machines.) This might in turn imply a larger advantage of parallel priority queues over centralized ones.

The usage pattern and even the type of operations used for a parallel priority data structure very much depends on the underlying application. For example, *branch-and-cut* (e.g. [32]) can be considered a variant of branch-and-bound where most nodes have degree one and it is more efficient to perform the expansion of a child node on the PE of the parent. In this context an operation `insertAndDeleteMinIfBetter` would be useful that either returns a node better than the child node and inserts the child node or returns the child node without changing the queue. Other applications like discrete event simulation could be investigated. Also, the question whether batched access to many elements is useful can only be answered for a particular application. In other cases, the combination of priority queues and a specific application will only be a starting point for developing a new integrated algorithm (e.g. [3]).

References

[1] A. Bäumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Europar*, 1996.

[2] G. S. Brodal. Priority queues on parallel machines. In *5th Scandinavian Workshop on Algorithm Theory*, number 1097 in LNCS, pages 416–427. Springer, 1996.

[3] G. S. Brodal, C. D. Zaroliagis, and J. L. Träff. A parallel priority data structure with applications. In *11th International Parallel Processing Symposium*, 1997. to appear.

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

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subromonian, and T. v. Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, 1993.

[6] S. K. Das, M. C. Pinotti, and F. Sarkar. Optimal and load balanced mapping of parallel priority queues in hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):555–564, 1996.

[7] N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, Mar. 1992.

[8] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the bulk-synchronous parallel model with applications to priority queues. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1996.

[10] A. K. Gupta and A. G. Phoutiou. Load balanced priority queue implementations on distributed memory parallel machines. In *Sixth International Conference on Parallel Architectures and Languages Europe*, number 817 in LNCS, pages 689–700, Athens, July 1994. Springer.

[11] D. Ierardi. 2d-bubblesorting in average time $O(\sqrt{N \lg N})$. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 36–47, 1994.

[12] L. V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In R. H. Halstead, editor, *Parallel Symbolic Computing*, number 748 in LNCS, pages 12–41. Springer, 1993.

[13] L. V. Kale, B. H. Richards, and T. D. Allen. Efficient graph coloring with prioritization. In *Parallel Symbolic Languages and Systems*, number 1068 in LNCS, pages 190–208. Springer, 1995.

[14] R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765–789, 1993.

[15] D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 1973.

[16] B. Le Cun and C. Roucairol. Concurrent data structures for tree structured search algorithms. In A. Ferreira and J. D. P. Rolim, editors, *Parallel Algorithms for Irregular Problems: State of the Art*, chapter 7, pages 135–155. Kluwer, 1995.

[17] T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, 1992.

[18] F. Mattern and H. Mehl. Diskrete Simulation - Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung. *Informatik Spektrum*, 12:198–210, 1989.

[19] W. F. McColl. Scalable computing. In *Computer Science Today*, number 1000 in LNCS, pages 46–61. Springer, 1996.

[20] C. G. Plaxton. On the network complexity of selection. In *Foundations of Computer Science*, pages 396–401. IEEE, 1989.

[21] S. Rajasekaran. Randomized parallel selection. In *Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, number 472 in LNCS, pages 215–224, Bangalore, 1990. Springer.

[22] A. Ranade. A simpler analysis of the Karp-Zhang parallel branch-and-bound method. Technical report, University of California Berkeley, 1990.

[23] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *Sixth IEEE Sypmposium on Parallel and Distributed Processing*, pages 97–103, October 1994.

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

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

[26] P. Sanders. Fast priority queues for parallel branch-and-bound. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 379–393, Lyon, 1995. Springer.

[27] P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI, 1997.

[28] P. Sanders. Flaschenhalsfreie parallele Priority queues. In *PARS Workshop, Potsdam*, number 13 in PARS Mitteilungen, pages 10–19, September 1994.

[29] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS-VII Architectural Support for Programming Languages and Operating Systems*, pages 26–36. ACM, 1996.

[30] N. Shavit and A. Zemach. Diffracting trees. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 167–176, Cape May, New Jersey, 1994.

[31] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the Complete Reference*. MIT Press, 1996.

[32] P. Van Hentenryck. Constraint solving for combinatorial search problems: A tutorial. In *Principles and Practice of Constraint Programming*, number 976 in LNCS, pages 565–587, 1995.