Parallelism and Locality in Priority Queues

A. Ranade S. Cheng E. Deprit J. Jones S. Shih

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

We explore two ways of incorporating parallelism into priority queues. The first is to speed up the execution of individual priority operations so that they can be performed one operation per time step, unlike sequential implementations which require $O(\log N)$ time steps per operation for an N element heap. We give an optimal parallel implementation that uses a linear array of $O(\log N)$ processors.

Second, we consider parallel operations on the priority queue. We show that using a d-dimensional array (constant d) of P processors we can insert or delete the smallest P elements from a heap in time $O(P^{1/d}\log^{1-1/d}P)$, where the number of elements in the heap is assumed to be polynomial in P. We also show a matching lower bound, based on communication complexity arguments, for a range of deterministic implementations. Finally, using randomization, we show that the time can be reduced to the optimal $O(P^{1/d})$ time with high probability.

1 Introduction

Much of the theoretical work in parallel computing is based on the PRAM model. The PRAM is very different from real parallel machines, and as a result, the relevance of this work to real machines is unclear. Sparse network models more faithfully represent parallel computers, and hence algorithms developed for these networks are more useful in practice.

A very important issue in designing algorithms for processor networks is *data distribution*, i.e. deciding how to distribute a given data structure amongst the memories of the processors in the network. Good data distribution can substantially reduce frequency of communication or the distance over which the communication must take place, and thus reduce the total amount of data that must be moved.

In this paper, we consider the issue of designing good data distribution schemes for implementing priority queues. We consider the simplest priority queue variant, viz. a data structure that supports insert and delete-min operations. A priority queue is commonly used in sequential computing, e.g. in event-driven simulation, sorting, and a number of geometric and graph algorithms. Because of these uses, it seems natural to incorporate parallelism in priority queues.

Two natural avenues have been considered. First, if the number of processors available is very small as compared to the number of elements, we might use them to speed up individual priority queue operations. Alternatively, if a large number of processors are available, then it is desirable to allow several queue operations to proceed in parallel. Both of these alternatives have been explored on the PRAM. In this paper we present solutions that work well on processor networks, in particular, constant dimensional processor arrays.

There are several strategies for designing parallel algorithms for sparse networks. One possibility is to first design a PRAM algorithm, and then use PRAM simulation techniques to run the algorithm on the sparse network at hand. In this case, which we will refer to as *brute-force simulation*, the data structures used by the algorithm are typically distributed randomly among the processors in the network. In some cases, this can be proved to be the best possible strategy [8], but here we use it as a baseline to evaluate sparse network algorithms.

The following alternative strategy is gaining popularity. The user first writes a PRAM program, and then annotates it using directives to indicate how data structures must be stored. The annotations specify for each datum used by the PRAM program a processor in the sparse network and a location in the memory of that processor that will hold that datum. (In principle, we may have several copies to be stored for each PRAM datum, however, here we only allow a single copy.) Several languages such as pSather and CM-Fortran provide directives for data layout, and thus effectively support this strategy, which we shall call the *mapped simulation* of the PRAM program. The idea is that good data layout will reduce communication overhead, and improve the performance beyond what can be achieved using brute-force simulation. In the best case, the annotations may even enable the program to run as fast as the original on the PRAM. Given a PRAM program and a sparse network, an interesting question is, what is the best data layout? If the best data layout does not make the network algorithm as fast as the original PRAM algorithm, then we must either choose a different PRAM algorithm, or design an algorithm explicitly customized for the network at hand. (Of course, it is possible, because of communication complexity, that every network implementation is provably slower than the PRAM.)

Our work on priority queues uses mapped simulation, as well as customized algorithm design. Let N denote the number of elements stored in the priority queue at any step. For two-dimensional arrays, we will assume N to be polynomial in the number of processors P. Our main results are as follows.

1. We show that a linear array with $\log N$ processors can support insert and delete-min operations at the rate of one operation per $O(1)$ steps, and with delay of $O(1)$ for each operation (Section 2). We thus match the performance achieved on the PRAM by Rao and Kumar [9], but we require only a sparse network. Our solution is not a mapped simulation but rather a customized algorithm for the linear array.

2. We show how a d-dimensional array of P processors with side $P^{1/d}$ can perform P operations (insert/delete-min) in time $O(P^{1/d} \log^{1-1/d} P)$, for constant d (Section 4). This is a mapped simulation of the P processor $O(\log P)$ time algorithm due to Pinotti and Pucci [7] for performing P operations in parallel (Section 3). Note that brute force simulation of [7] would take time $O(P^{1/d} \log P)$.

3. We give a matching lower bound of $\Omega(P^{1/d} \log^{1-1/d} P)$ for all mapped simulations of Pinotti and Pucci's algorithm (Section 6) on arrays. This shows that our data layout for the mapped simulation above is the best possible. Our lower bound argument is based on communication complexity. However, unlike standard arguments which use bisection, we consider multiway partitions of the network (or VLSI chip). From our general result we can also show a VLSI lower bound of $AT^2 = \Omega(\sqrt{P \log P})$. Also, for the P processor butterfly network, we can show a lower bound of $\Omega(\log^2 P)$ time. Notice that this lower bound can be matched simply by a brute-force (albeit randomized) simulation of the algorithm of [7].

4. We give a randomized priority queue implementation for a P processor d-dimensional array that performs P operations in time $O(P^{1/d})$. This matches the bisection width lower bound applicable to all priority queue implementations. We do not know of any deterministic algorithm matching the bisection width bound. (Section 7)

Our algorithms as described in Sections 2 and 4 have widely disparate memory requirements among the processors. In Section 5 we present a scheme to balance the memory requirements. Finally, in Section 8 we list some extensions and open problems suggested by our work.

2 Parallelizing Individual Operations

We show how to implement insert and delete-min operations with unit delay. Our solution uses the composite array shown in Figure 1, which can be simulated in constant time using $\log N$ processors connected in a linear array, where N denotes the number of elements stored in the priority queue.

We first describe the two components of the array and then explain how the parts are interconnected.

1. *Cache:* This is a systolic priority queue having $L = \theta(1) + \log N$ processors [6]. It can receive requests once every 2 clock cycles from the external world at processor P_1 and return the response (for delete-min requests) in one cycle, again at processor P_1. It is not adequate by itself, of course, since it can only manage $O(\log N)$ elements. In particular, if we insert too many elements into it, they overflow from processor P_L. We will omit the details of its functioning, and mention without proof that if an element x overflows from processor P_L at step t, then at least $L/2$ elements smaller than x must be present in the array at time t.

2. *Backup:* This is a linear array with $n = \log N$ processors. It is capable of managing N elements, and can satisfy requests at a constant rate, though it has a latency of $O(\log N)$. The elements to be stored are organized in a single binary heap of $\log N$ levels, with level i of the heap stored in processor i of the backup array. Requests enter at processor n, and move through the array, simulating the uniprocessor implementation in a pipelined fashion.

   Insertions require a single leftward pass through the array, in which the standard up-heap operation is implemented in a pipelined manner. Delete-min operations, on the other hand, require two passes. In the leftward pass, the delete-min request enters at processor Q_n and deletes the last element, say x, stored in the priority queue, which it carries along to processor Q_1. Here, the element stored in processor Q_1 (the root of the binary heap) is removed and returned as the response. The element x is then placed at the root, and the down-heap operation is implemented by a rightward pass from processor Q_1 toward processor Q_n. By suitably spacing out requests and pipelining successive operations, we can serve requests at a constant rate, but we omit details here. (For instance, notice that the above description requires each processor to access not only its own memory, but the memories of its neighbors

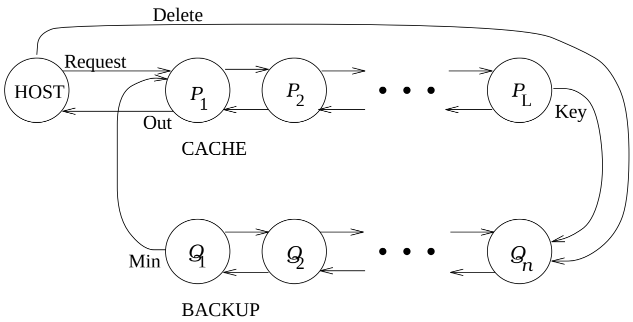

Figure 1: Composite array for parallel heap.

for executing the upheap and down-heap operation. For this we divide each cycle into 3 subcycles in which processor i accesses its own memory, the memory of processor $i-1$, and the memory of processor $i+1$. Another problem is conflict between leftgoing and rightgoing requests. We avoid these by subdividing the cycle further.)

To configure the composite array, we adjust the clocks of the cache and the backup so that each can receive a request at the rate of 1 per cycle, causing only constant slowdown for the final result. The host is only allowed to issue requests once every two cycles. Insertion requests go only into the cache, and the keys that overflow from the cache are inserted into the backup. The delete-min requests are issued to the cache, as well as the backup. The response from the cache is returned to the host, the response from the backup is reinserted into the cache. Thus, the cache is required to serve requests once every time step, alternating service between host and backup. Notice that there may be an overflow and a delete-min from the host in the same step. This is easily handled in the backup—in the left pass nothing is removed from the priority queue, instead the overflowed element that is to be inserted is moved to processor Q_1 and the down-heap operation executed using it.

We sketch the proof of correctness. Suppose some key x is present in the array at time t when a larger key is returned to the external host by the cache. We show that this gives a contradiction. Assume, without loss of generality, that t is the earliest time instant for which this happens. We know that the cache always returns the smallest of the keys in it. Thus x must be present in the backup at time t. Now consider the latest time step $t' \leq t$ at which some key $x' \leq x$ overflowed into the backup. At time t' we know using the properties of the cache that at least $L/2$ keys smaller than x must be present in the cache. Thus the earliest $L/2$ deletions cannot cause a smaller key than x to be returned. However, each of these deletions is also issued to the backup, and the smallest key in the backup is reinjected into the cache with a latency of $\log N$.

Now, consider the kth delete-min issued after step t'. Since the host can issue deletions only at the rate of once every 2 steps, this will be issued at least $\log N$ steps after the $k - (\log N)/2$th delete-min. Thus, the first $k - (\log N)/2$ deletions will already have been served by the backup and would have caused responses to be reinjected into the cache. Also, we know by definition of t' that no keys smaller than x overflow into the backup between t' and t. Thus, the number of keys in the cache that are smaller than x cannot decrease below $L/2 - (\log N)/2 = (L - \log N)/2$ between time steps t' and t. Thus if we choose $L = \Omega(1) + \log N$, we are guaranteed to have at least one key smaller than x present in the cache at step t. As a result, the cache cannot return anything larger than x at step t, giving the contradiction.

3 P-Bandwidth Heap

We next consider executing P heap operations in parallel using P processors. On the PRAM this can be done using the P-bandwidth heap of Pinotti and Pucci[7], the operation of which we outline below. The P-bandwidth heap is similar to the sequential binary heap, except that each node holds P keys instead of just one. Further, if x is any key stored in node R, and y any key stored in a child of node R, then we require that $x \leq y$. This is called *extended heap order*.

We will use $||$ to denote concatenation and lower case letters to denote keys stored in a node, i.e. r will denote the sequence of keys (sorted in nondecreasing order) associated with the node R.

Insertion. The P keys to be inserted are sorted and stored in the next available leaf in the heap, say leaf R. Then the extended heap order is reestablished by performing the *Rearrange* operation defined below on the root-leaf path $R_1, \ldots, R_h$, with R_1 being the root and $R_h = R$. Note that before the operation the extended heap order holds along the path $R_1, \ldots, R_{h-1}$, so that the sequence of keys $r_1||r_2||\ldots||r_{h-1}$ is nondecreasing.

Rearrange $(R_1, \ldots, R_h)$:
  $l_1, \ldots, l_{hP} \leftarrow \text{Merge}(r_h, r_1 \,||\, \ldots \,||\, r_{h-1})$
  for $i \leftarrow 1, \ldots, h$
    $r_i \leftarrow [l_{P(i-1)+1}, \ldots, l_{Pi}]$.

Because of *Rearrange*, the largest key stored in any node R_i along the path can only decrease. As a result, the extended heap order is guaranteed between any R_i and its children.

Delete-min.

1. Find the *min-path* μ, defined by Pinotti and Pucci as follows: μ begins with the root R_1, and for each non-leaf $R \in \mu$, if S and T are the children of R, then S is in μ if either T is void, or $\max(s) \leq \max(t)$.

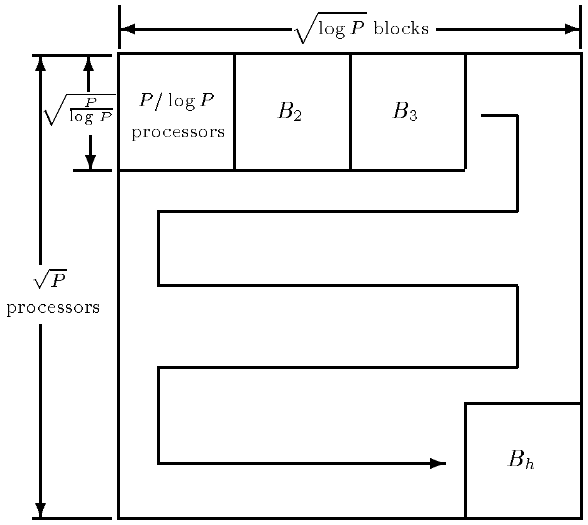

Figure 2: The mesh of processors.

2. For each $R_i \in \mu$, we consider its sibling, say S_i. We merge the lists at R_i and S_i, and store the smaller elements in R_i, and the larger in S_i.

3. The list at the root is returned as the result.

4. For every other node on μ, we shift its list one step upward, i.e. for $i = 1$ to $h-1$, $r_i \leftarrow r_{i+1}$.

5. We move the list associated with last element in the heap into R_h, and then perform *Rearrange* $(R_1, \ldots, R_h)$.

The key observation guaranteeing correctness is that because of step 2, steps 4 and 5 do not violate the extended heap order.

Time. *Rearrange* takes time $O(h + \log P)$ using standard merging algorithms [5]. For delete-min, the min-path is constructed sequentially in time $O(h)$, and steps 2 and 5 can be performed in time $O(h + \log P)$ by using P/h processors at each level of the heap. For N polynomial in P, the total time for each parallel operation is $O(\log P)$.

4 Array Implementation

We first consider mapped simulations of [7] on 2-dimensional arrays of P processors. The P-bandwidth heap is distributed among processors as follows. We divide the mesh into square blocks of $P/\log P$ processors and number the blocks in a snakelike order, from B_1 to $B_{\log P}$ (see Figure 2). Each block manages a level of the heap, so that B_1 holds the elements for the root, B_2 holds the elements for the children of the root, and so forth. We assume, for convenience in the following description, that $N = P^2$, so that the number of levels is equal to the number of processors ($h = \log P$). Each block of $P/\log P$ processors stores some number of nodes in the heap, i.e. their associated lists of length P. These lists are stored, $\log P$ elements per processor, sorted in a snakelike order within the block, and in a sorted list on each processor. For the *Rearrange* operation, we also need the vector l defined above. We store this in an analogous manner, i.e. elements $l_{P(i-1)+1}, \ldots, l_{Pi}$ are held in block B_i. Within each block each processor stores $\log P$ elements of l according to the snakelike order within each block.

This scheme requires $O(N)$ memory for the block managing the bottom row. Section 5 will alleviate this problem with a method of distributing the heap which balances the number of nodes stored in each block. Also, we can extend our algorithm to the case $N = \mathrm{Poly}(P)$ by simulating with constant slowdown a mesh with $h = O(\log P)$ blocks of processors.

The core of the insertion and delete-min algorithms is the operation *Rearrange* $(R_1, \ldots, R_h)$. This requires the keys in the sorted sequence $r_1 || \ldots || r_{h-1}$ to be displaced down so that smaller keys from r_h can fill the gaps, and we get a sorted sequence. We do this essentially by computing for each element its rank in the final order. This is done as follows:

1. We broadcast a copy of r_h to each block. Let z_i denote the local copy of r_h in block B_i.

2. We merge r_i and z_i locally within each block. Let k_{ij} denote the jth smallest key in r_i. As a result of the merge operation, we know the number d_{ij} of keys in z_i that are smaller than k_{ij}. This enables us to determine the rank of k_{ij} in the final order, which is simply $(i-1)P + j + d_{ij}$.

3. We next move the keys in $R_1, \ldots, R_{h-1}$ to their proper positions in $l_1, \ldots, l_{hP}$. Notice that since $d_{ij} \leq P$, each key k_{ij} either stays within B_i, or moves into B_{i+1}.

4. We next fill the gaps in l left after the above step. This can be done in parallel in each block. Notice that each block already has a copy z_i of r_h. The elements of z_i which move into the gaps can be determined by merging z_i and the elements that have moved into $l_{P(i-1)+1}, \ldots, l_{Pi}$.

5. Finally, elements of l are moved back into r_i. Because of the manner in which l is stored, this is a local move within each processor.

The broadcast step (1) copies the P elements of r_h, which are stored $\log P$ per processor, from B_h to corresponding processors in all other blocks. We do this in two phases: in the first we broadcast to all blocks that are in the same column of blocks as B_h, then we broadcast within rows of blocks. For the first phase, consider any single column of processors passing through B_h. The $\sqrt{P/\log P}$ processors in B_h that are in this column send all the values they have to all processors within the column. There are

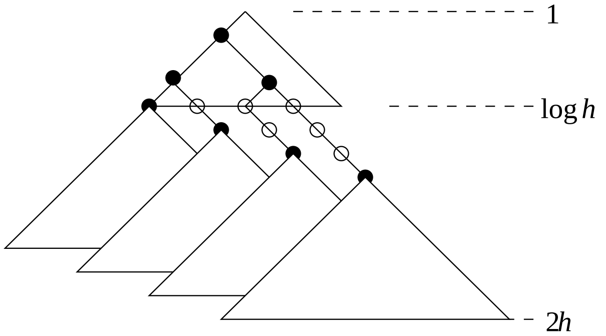

Figure 3: Balancing heap nodes among levels.

$\sqrt{P/\log P} \times \log P = \sqrt{P \log P}$ values to be moved a maximum distance less than $\sqrt{P}$. By pipelining, this movement can be achieved in time $O(\sqrt{P \log P})$. The time for broadcasting across rows is the same.

Next, in step 2, each processor block merges r_h and r_i to form x_i. Clearly, we can sort these $2P$ elements into a snakelike order within the block in time $O(\sqrt{P \log P})$. We perform 3 in two phases. First we move keys between blocks, and then within each block. Note that at most P keys must move from any block B_i to adjacent block B_{i+1}. This can be done in $O(\sqrt{P \log P})$ steps. Within each block we only need to move around at most P keys, which are initially placed at most $O(\log P)$ keys per processor. Again the time is $O(\sqrt{P \log P})$. Step 4 is little more than a merge, and requires $O(\sqrt{P \log P})$ steps. Step 5 is local and takes $O(\log P)$ steps. The total time is thus $O(\sqrt{P \log P})$.

Rearrange can be used for insertions as follows. First, read the input elements $i_1, \ldots, i_P$ one per processor and gather it into B_h with shift operations similar to the broadcast above. Processor block B_h then sorts $i_1, \ldots, i_P$ and stores the result into r_h, the leftmost vacant leaf of the heap. The final step is *Rearrange*. All of these steps can be completed in time $O(\sqrt{P \log P})$.

It is easily seen from the above that delete-min can also be performed in time $O(\sqrt{P \log P})$, and we omit the description.

d-dimensional arrays. We divide the array into blocks of size $P/\log P$, with side-length $(P/\log P)^{1/d}$. Each basic operation described above can be performed in time $O(P^{1/d} \log^{1-1/d} P)$, which is also the total time.

5 Balancing Memory Needs

The algorithms in Sections 2 and 4 require the nodes of the tree to be distributed unevenly among the processors in the network. We now present a scheme that ensures a more uniform distribution of heap nodes.

The main idea is to alter the shape of the heap. In the sequential case, the shape is a complete binary tree. However, this is not crucial, we only require small height, and moderate degree. We alter the complete binary tree so that no single level in it contains too many nodes. This is illustrated in Figure 3. We start with a complete binary tree of height h. The top $l = \log h$ levels remain unchanged. There are h subtrees rooted at level l. We stagger the ith subtree from the left by adding i nodes as shown.

We can define a canonical numbering of the tree nodes, e.g. a breadth-first-numbering and insert nodes into the tree in that order. Given any integer j, we need somewhat more complicated bookkeeping to determine the position of the jth node in the tree, but it is still easy as the shape of the tree is fixed once and for all.

The simplest way to incorporate this tree into the preceding algorithms is to use a linear array with $2h$ processors for the algorithms of Section 2, and assign level k to processor k. It is easily seen that no processor gets more than its fair share, or $O(2^h/h)$ nodes. For the algorithm of Section 4, we use a mesh with $2h$ blocks, and assign one level to each block. The algorithms for insertion and delete-min remain essentially the same.

6 Lower Bound

In any mapped simulation of Pinotti and Pucci's algorithm, it is easily seen that the rearrange operation must perform a merge of lists of length $O(P \log P)$ and P. We can lower bound the time for this using a communication complexity argument. Standard arguments consider information transfer across the bisection of the network (or smaller dimension of a VLSI chip, for AT^2 bounds). In our case these do not give the best bounds. Bilardi and Preparata[2], and recently Adler and Byers[1], consider information transfer between smaller regions of a VLSI chip. Our argument builds upon theirs, and is similar to Cypher's use of pin limitation arguments to prove lower bounds[3]. First, we need the notion of a *q-section width* of a graph, which is a natural extension of the bisection width.

Definition 1 *Let $V(G)$ denote the vertex set of a graph G. Let $G_1, \ldots, G_q$ denote a partition of G, i.e. $\sum_i |V(G_i)| = |V(G)|$. Call a partition* balanced, *if for all $i > 0$, $|V(G_i)| = \Omega(|V(G)|/q)$. Let $X(G_i)$ denote the number of edges with one endpoint in G_i and the other outside G_i. Let $X = \max_i X(G_i)$.*

The q-section width *of G, denoted by $W(G, q)$ is the minimum value of X over all possible balanced partitions of G. Any balanced partition that achieves the minimum value is called a* q-section *of G.*

We consider the general problem of merging lists of length m and n, on a network with graph G, such that $m \geq n \geq |V(G)|$. We further assume that the smaller list is stored in a distributed manner, i.e. each processor in the network holds $n/V(G)$ elements from it. For the priority queue problem, this is equivalent to assuming that the elements to be inserted are initially distributed one per processor. We do not make any assumptions about how the longer list is stored. For simplicity, we assume $m \geq 2n$.

Theorem 1 *Let G be the graph of a parallel computer with $V(G)$ corresponding to processors, and the edges corresponding to communication links. Then merging two sequences of length m and n, $m \geq 2n$ must take time $T = \Omega(n/W(G, m/n))$.*

Proof: Call the two input lists X and Y, which have lengths m and n, respectively, and call the output list Z, which has length $m + n$. Let $G_1, \ldots, G_q$ denote a q-section of G, for $q = m/n$. Clearly, there must be some i such that G_i is required to produce at least $(m+n)/q \geq m/q = n > n/2$ elements of Z. We also know that at least $n/2$ elements of Y are read outside G_i. By choosing the values of X and Y, we can force the $n/2$ elements that are read outside G_i to be output inside G_i. But there are only $W(G, q)$ edges leaving G_i. Hence the time required must be at least $(n/2)/W(G, q) = \Omega(n/W(G, m/n))$. ∎

By choosing G to be a mesh, we have the following corollary.

Corollary 1 *Any chip which merges two lists of lengths m and n must have $AT^2 = \Omega(mn)$, where A is the area of the chip and T is the execution time.*

It can be shown that the q-section width of a d-dimensional array of P processors is $O((P/q)^{1-1/d})$, for constant d, and that the q-section width of a P-processor butterfly is $O(P/(q \log P/q))$. Choosing $m = P \log P$ and $n = P$ we have the following.

Corollary 2 *Any mapped simulation of Pinotti and Pucci's algorithm on a P processor d-dimensional array (d is a constant) must take time $\Omega(P^{1/d} \log^{1-1/d} P)$ for P insertions or deletions.*

The time on the P-processor butterfly is $\Omega(\log^2 P)$.

Our upper bounds for arrays from the previous section are thus optimal. We also note that a (randomized) brute-force simulation of Pinotti and Pucci's algorithm on the butterfly matches the lower bound given above.

Remarks. Note that a standard bisection based argument would give the time bound to be $\Omega(n/W(G, 2))$, which is in general weaker than our result. Also, we can obtain bounds for different networks using Corollary 1, e.g. for the butterfly, the VLSI layout area $A = O(P^2/\log^2 P)$, which gives $T^2 = \Omega((P)(P \log P)/(P^2/\log^2 P))$, i.e. $T = \Omega(\log^{1.5} P)$, which is weaker than what we have in Corollary 2.

7 Randomized Priority Queues

No mapped simulation of [7] can be faster than the lower bound given above; however, this does not rule out the possibility of faster customized implementations. We present one such implementation, but this involves randomization. Karp and Zhang [4] use a technique similar to the following, but only approximate a priority queue. We maintain a local priority queue at each processor. For the insert operation, each processor sends its request to a randomly chosen processor, where it is inserted into the priority queue of that processor. With high probability, this will cause $O(\log P/\log\log P)$ requests to be inserted into any single local priority queue. Thus, insertions can complete in time $O(P^{1/d})$ including the time for routing the requests and for insertions into the local heaps.

Delete-min can be performed as follows:

1. Make a copy of the two smallest elements in each processor. Let Q denote the set of these $2P$ elements.
2. Find M = the Pth smallest element in Q.
3. Let R_i denote the set of elements in processor i that are not larger than M.
4. Find S = the smallest P elements from $\bigcup_i R_i$.
5. Return S as the result and delete the elements in S from the priority queues that they were stored in.

Clearly, step 1 takes constant time. Step 2 can be finished in time $O(P^{1/d})$ by sorting Q. We will prove below that $|R_i| = O(\log P)$, and that $\sum |R_i| = O(P)$, both with high probability. As a result, step 3 requires time $O(P^{1/d})$ for broadcasting M, and time $O(\log P \log N)$ for forming R_i. Step 4 can be performed using sorting, in time $O(P^{1/d})$. Finally, step 5 can be done in time $O(P^{1/d} + \log P \log N)$. The total time is thus $O(P^{1/d})$, for N polynomial in P and constant d.

Lemma 1 *Let $R = \bigcup_i R_i$. Then $|R| = O(P)$ with high probability. Further, for all i, $|R_i| = O(\log P/\log\log P)$, with high probability.*

Proof: Let R' denote the smallest $4P$ elements overall at the time of the delete-min operation. We shall show that with high probability, the set Q computed above will contain at least P elements from R'. Thus with high probability, we will have $R \subseteq R'$, i.e. $|R| \leq 4P$.

Suppose for the purpose of analysis that the elements of R' are placed on the processors sequentially, in increasing order. Let Q_x denote a random variable that takes value 1 if the xth smallest element enters Q or if P of the smallest $x-1$ elements have already entered Q, and 0 otherwise. Note that $\sum_x Q_x < P$ if and only if fewer than P elements from R' enter Q. Thus we must determine the likelihood that $\sum_x Q_x < P$.

$Q_x = 0$ if and only if the xth smallest element of R' is placed on a processor that holds at least two elements smaller than it, and fewer than P of the smallest $x-1$ elements have already entered Q. But if fewer than P elements are in Q when the xth smallest gets placed, there can be at most $P/2$ processors with 2 or more elements. Thus the probability that $Q_x = 0$ is at most $1/2$, no matter how the previous elements get placed, and the probability that $Q_x = 1$ is at least $1/2$.

Thus, even though the variables Q_x are not independent,

$$p\left(\sum_x Q_x < P\right) < B(4P, P, 1/2)$$

where $B(n, m, p)$ denotes the probability of getting fewer than m successes in n independent Bernoulli trials with probability p. From Chernoff bounds,

$$B(n, m, p) < \exp\left(-\left(1 - \frac{m}{np}\right)^2 np/3\right),$$

and thus after simplifying we get

$$p\left(\sum_x Q_x < P\right) < \exp(-P/6)$$

The second part follows from the observation that when $4P$ elements are randomly stored in P processors, each processor gets $O(\log P/\log\log P)$ elements with high probability. ■

8 Concluding Remarks

The main open problem is whether there are deterministic $O(P^{1/d})$ time parallel priority queue implementations on arrays, or $\log P$ time implementations on Butterflies. We have shown that these cannot be mapped simulations of Pinotti and Pucci's algorithm.

Another intriguing question is how the results change if we allow data to be replicated.

The idea of mapped simulations is very common in practice. Mapped simulations naturally generalize the strategy of embedding dataflow graphs into processor networks to devise parallel algorithms. We expect that the idea of using communication complexity arguments to prove lower bounds will be useful in other contexts besides priority queues.

Acknowledgements

Abhiram Ranade is supported in part by NSF-DARPA grant CCR-9005448. Szu-Tsung Cheng is supported in part by Cadence Design Corporation. Etienne Deprit is supported in part by an NDSEG fellowship. Jeff Jones is supported in part by National Science Foundation Grant No. CCR-9210260. This material is based in part upon work supported by the the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, the Department of Energy, and the National Science Foundation under Infrastructure Grant No. CDA-8722788.

References

[1] M. Adler and J. Byers. AT^2 Bounds for a Class of VLSI Problems and String Matching. ACM Symposium on Parallel Algorithms and Architectures, 1994.

[2] G. Bilardi and F. Preparata. Area–Time Lower Bound Techniques with Applications to Sorting. *Algorithmica*, 1(1):65–91, 1986.

[3] R. Cypher. Theoretical Aspects of VLSI Pin Limitations. *SIAM Journal on Computing*, 22(2):356-378, 1993.

[4] R. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Annual Symposium on Theory of Computing*, pages 290–300, 1988.

[5] C. Kruskal. Searching, merging and sorting in parallel computation. *IEEE Transactions on Computers*, C-32(10):942–946, 1983.

[6] F.T. Leighton. *Introduction to parallel algorithms and architectures.* Morgan-Kaufman, 1991.

[7] M. Pinotti and G. Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33-40, 1991.

[8] A.G. Ranade. A Framework for Analyzing Locality Issues in Parallel Computing. In *Proceedings of the International Heinz-Nixdorf Symposium on "Parallel Architectures and their Efficient Use", University of Paderborn, Paderborn, Germany*, November 1992.

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