Load Sharing with Parallel Priority Queues

Ian Parberry*
Department of Computer Sciences
University of North Texas

Abstract

For maximum efficiency in a multiprocessor system the load should be shared evenly over all processors, that is, there should be no idle processors when tasks are available. The *delay* in a load sharing algorithm is the larger of the maximum time that any processor can be idle before a task is assigned to it, and the maximum time that it must wait to be relieved of an excess task. A simple parallel priority queue architecture for load sharing in a p-processor multiprocessor system is proposed. This architecture uses $O(p \log(n/p))$ special-purpose processors (where n is the maximal size of the priority queue), an interconnection pattern of bounded degree, and achieves delay $O(\log p)$, which is optimal for any bounded degree system.

1 Introduction

One advantage that multiprocessor computers have over uniprocessors is the ability to speed up computation by having the processors compute in parallel. The archetypal model studied is the PRAM, in which it is assumed that concurrent access to a shared memory is possible at instruction-cycle speeds. The PRAM is attractive since it is relatively easy to program and analyze. Although there is no obvious direct implementation with current technology, it can easily be implemented by a bounded-degree network of processors each with a local memory. Such an implementation requires an increase in running time for a p processor system by a factor of $\Theta(\log p)$, which can be achieved using Columnsort [15] and techniques described in Parberry [19]. However, many of the fastest parallel algorithms (for example, those in $\mathcal{NC}$ [8, 22]) require a massive amount of communication. It is not unusual to require interprocessor communication from distant processors for almost every local instruction executed. It is unreasonable to expect even Ranade's algorithm [26] to provide the interprocessor communication bandwidth that is required by these algorithms.

The world faced by programmers of today's parallel computers is vastly different from that of the theoretician. Parallel computers are attractive because they provide a large amount of processing power at a reasonable cost. However, interprocessor communication is too slow to exhibit the surprisingly large (often exponential, see Parberry [18, Chapters 5–9]) theoretical parallel speedups that have been discovered. It is more efficient in practice to use algorithms that do more local computation than communication. Although it is reasonable to expect the communication speed to increase as technology is improved, it is also reasonable to expect the number of processors to increase at the same time. It would perhaps be naive to conjecture the availability in the immediate future of computers with tens or hundreds of thousands of general-purpose processors and an instruction-cycle time of 10^{-8} seconds or less in which general interprocessor communication can take place as fast as a local instruction.

*Research supported by NSF Grant CCR-8801659. Author's address: Department of Computer Sciences, University of North Texas, P.O. Box 13886, Denton, TX 76203-3886, U.S.A. Electronic mail: `ian@ponder.csci.unt.edu`.

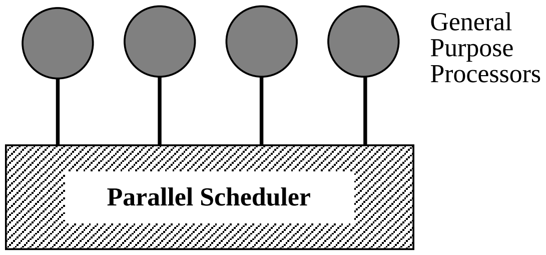

Figure 1: The architecture with $p = 4$ general-purpose processors.

One solution to the problem of slow communication is to increase the granularity of the parallelism: instead of taking the basic unit of the computation to be a single instruction, take the basic unit to be a sequential task at the operating system level. The granularity of this type of parallelism is large enough to allow complete interprocessor communication while the tasks are being executed. For maximum efficiency in a multiprocessor system, the load should be shared evenly over all processors, that is, there should be no idle processors when tasks are available.

We will assume that:

1. The general-purpose processors are both the producers and consumers of the tasks.
2. The duration of the tasks is unknown.
3. Any task may be executed on any general-purpose processor.
4. Each task is to be executed in its entirety on a single general-purpose processor.
5. Each task is assigned a static priority at the time of creation.
6. A processor that creates a task must remain idle until it is relieved of that task.
7. No processor may commence a task if there exist unassigned tasks of higher priority.
8. Interprocessor communication amongst p processors takes time $\Omega(\log p)$.

One measure of the efficiency of a load sharing algorithm is its *throughput*, that is, its ability to schedule an arbitrary set of jobs within the smallest amount of elapsed time. An efficient algorithm to find the optimal schedule under these conditions is highly unlikely since the problem is $\mathcal{NP}$-complete (Ullman [29]). There are many papers on the load sharing problem that assume some probability distribution on the arrival of the tasks, and measure a performance metric designed to quantify the average throughput of the system (for a survey, see Wang and Morris [30]). We will instead measure the maximum amount of time that a processor must remain idle at a stretch, which because of assumption 8 above must be $\Omega(\log p)$. Our algorithm falls into the *global, dynamic, physically distributed, cooperative, suboptimal, one-time assignment* category of Casavant and Kuhl [7].

The *delay* in a load sharing algorithm is the larger of the maximum time that any processor can be idle before a task is assigned to it and the maximum time that it must wait to be relieved of a newly created task. We present a multiprocessor architecture that uses a centralized scheduler, as shown in Figure 1. This scheduler will consist of a tightly-coupled parallel computer with $O(p \log(n/p))$ simple special-purpose processors (where n is the maximal size of the priority queue) connected together in a bounded degree network to give $O(\log p)$ delay in the worst case (which is asymptotically optimal for any bounded degree network).

We show that multiprocessor load sharing is reducible to parallel priority queue operations. A *parallel priority queue* is a priority queue in which the `insert` operation inserts p values, and the `deletemin` operation deletes the p smallest values, for some fixed value of p. Load sharing is achieved by breaking the computation into rounds of time $O(\log p)$. At the end of each round each processor may choose to insert a task into the parallel priority queue, or delete one of the tasks of highest priority, or continue local processing.

Many PRAM algorithms for standard priority queue operations (i.e. $p = 1$) can be found in the literature (for example, Biswas and Brown [6], Munro and Robertson [16], Rao and Kumar [27], and Jones [13]). Quinn and Yoo [25] describe parallel algorithms for filling and emptying a heap on a bounded degree network, but their algorithm deadlocks if `insert` and `deletemin` operations are interleaved. Fan and Cheng [11] have parallel priority queue algorithms for a bounded degree network with $O(\log p)$ delay on $O(n + p^2)$ processors. Das and Horng [9], and Deo and Prasad [10] have PRAM algorithms with delay $O(\log n)$ on p processors.

We present a new data structure for the priority queue called a *ragged heap*. Priority queue operations for the ragged heap can be pipelined in such a manner that O(1) delay can be achieved on a constant degree network of $O(\log n)$ processors. The algorithm is simple to implement and uses only $O(1)$ memory per processor in addition to the priority queue elements. The spirit of the algorithm is similar to that of Munro and Robertson [16], but the details are much more complicated since their algorithm is for a PRAM, which allows instantaneous communication between processors that hold different parts of the heap.

We describe how to extend our algorithm to the parallel priority queue, giving $O(\log p)$ delay with $O(p \log(n/p))$ processors using the AKS sorting network (Ajtai, Komlós and Szemerédi [1, 2]). The use of the AKS sorting network unfortunately results in an extremely large constant multiple (more than 6000, see Paterson [21]) in the delay and processor bounds. If a weaker form of load sharing is desired, then it is sufficient to use *halving networks*, which separate the inputs smaller than the median from those larger than it, instead of sorting networks. It is obvious that the depth of an n input halving network must be $\Omega(\log n)$. Alekseyev [4] has shown that the size must be $\Omega(n \log n)$ (see also Knuth [14, pp. 234–235]). We demonstrate that halving networks of depth less than $327 \log n$ exist. This results in a parallel priority queue algorithm with the same asymptotic time and processor bounds as before, and constant multiples that are smaller by a factor of 18. Ajtai, Komlós and Szemerédi [3] also have an improved bound on the depth of halving networks.

The main body of this paper is divided into four sections. The first section describes the ragged heap data structure and the priority queue algorithms. The second section describes the extension to parallel priority queues. The third section describes the application of parallel priority queues to load sharing in multiprocessor systems. The fourth section describes a technique for constructing efficient halving networks. A preliminary version of this paper appears in Parberry [20].

2 Ragged Heaps

We assume that the reader is familiar with the heap implementation of the ADT priority queue (Williams [31]). The root of a heap is said to be at *level* 1. The child of a node at level i is said to be at *level* $i + 1$.

A *ragged heap* is a heap with the following modifications. There are at most $O(\log n)$ empty nodes, termed *holes*, in the heap. The root cannot be a hole. Each nonroot node can have up to five different types of *messages*. As time proceeds, the holes move down the heap (and may accumulate at the bottom of the heap, giving it a "ragged" appearance), some of the messages move down the heap, and some move up. Some messages contain heap values, and some are empty. The five different types of message are shown in Table 1, together with their respective direction of motion and whether they carry a heap value.

We use $\log n$ processors with processor i connected to processor $i + 1$, for $1 \leq i < \log n$ (see

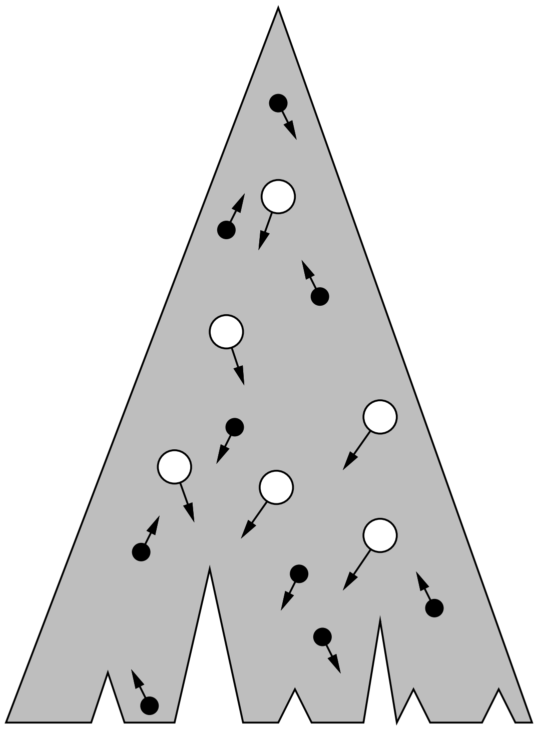

Figure 2: A ragged heap. The white circles are holes, and the black dots are messages.

Message	**Direction**	**Value**	**Purpose**
`promote`	up	yes	Promote last leaf value to replace hole
`bounce`	up	no	A failed `promote` (last leaf was hole)
`replace`	down	yes	Send inserted value to chase hole
`fetch`	down	no	Send for last leaf to replace hole
`purge`	down	no	Purge top bit of hole routing stacks

Table 1: Ragged heap messages.

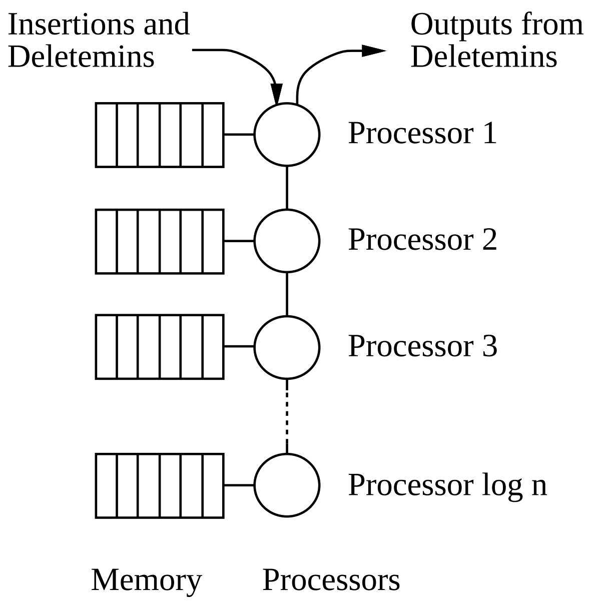

Figure 3: The interconnection pattern for a parallel heap.

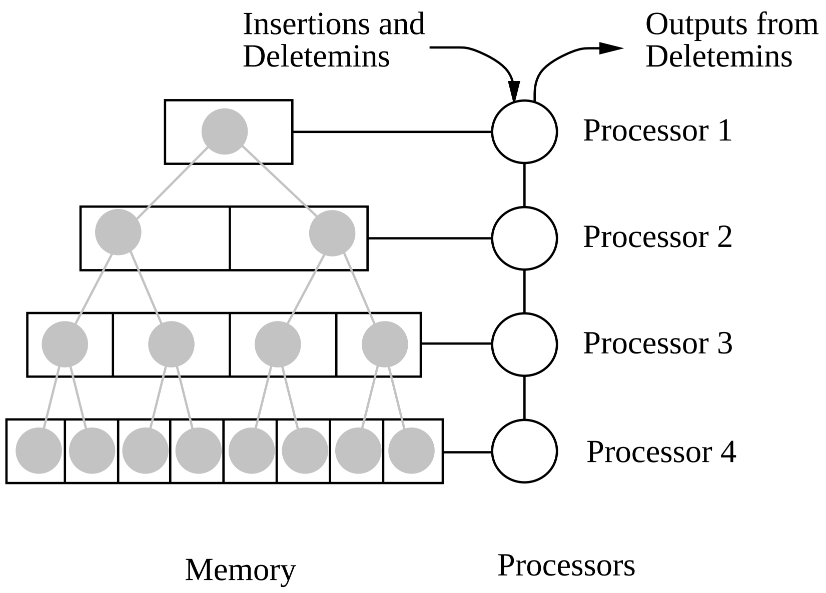

Figure 4: Embedding a 15-node priority queue into the memory of a 4-processor system.

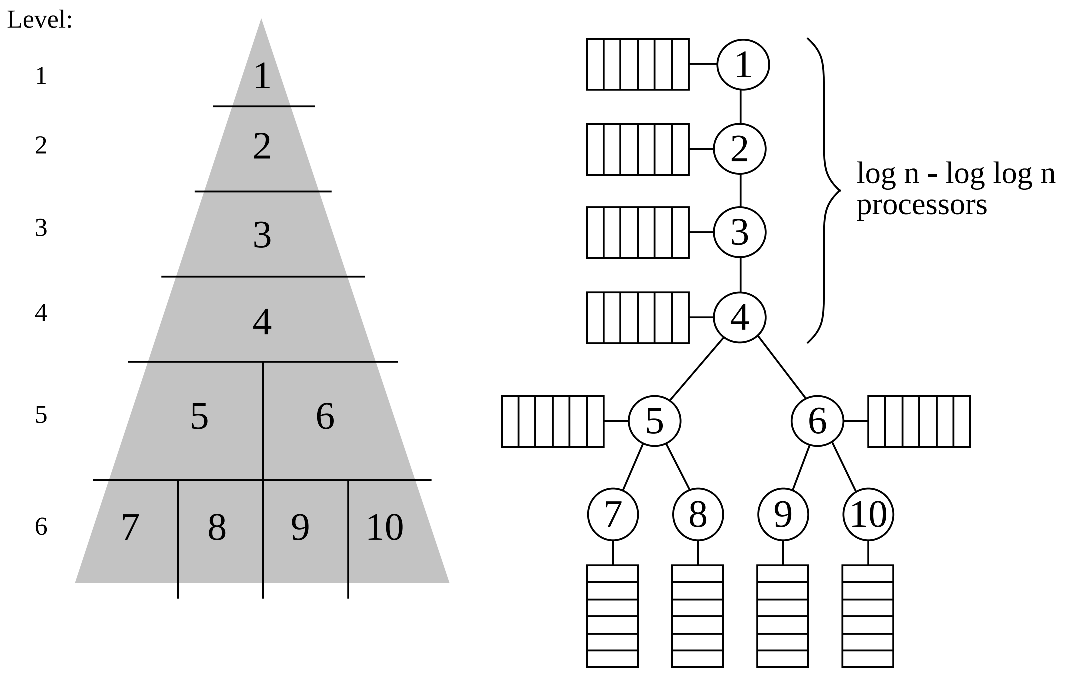

Figure 5: A more practical embedding. Each numbered area in the heap on the left is stored in the memory of the processor on the right that bears the corresponding number.

`push` b	insert the bit b onto the top of the stack
`pop`	delete and return the bit on the top of the stack
`size`	return the number of bits in the stack

Table 2: Stack operations.

Figure 3). Processor 1 receives the operations. The value removed during each `deletemin` operation is to be output from processor 1 before the processing of the next operation begins. We use one processor for each level of the heap: processor i stores all of the nodes at level i, for $1 \leq i \leq \log n$ (see Figure 4). In practice one should seek to distribute the priority queue values more equitably amongst the processors since the scheme we have described stores up to half of the values in a single processor. However, this can be easily rectified by replacing the last $\log\log n$ processors with a complete binary tree of depth $\log\log n$, resulting in a constant degree network of less than $2\log n$ processors, each of which carry $O(n/\log n)$ priority queue values (see Figure 5). We leave the necessary modifications to our priority queue algorithms as an exercise for the reader.

Each processor has a *stack* of bits. The operations permitted on this stack are shown in Table 2. We will suppose that all of the stack operations can be implemented in constant time. This can be achieved by packing the stack into a constant number of words. We will for the remainder of this paper assume that each stack occupies a single word of memory. We also assume that the processors can perform the elementary operations shown in Table 3 in constant time. This instruction-set, which is quite common on microprocessors, is slightly unusual in the theory community since it

`inc`	b	increment the integer in b by 1
`dec`	b	decrement the integer in b by 1
`xor`	a, b	bitwise exclusive-or a into b
`lsh`	a, b	left shift a by the amount in b, filling with zeros
`rsh`	a, b	right shift a by the amount in b
`copy`	a, b	copy b into a

Table 3: Elementary operations.

```
procedure push($b$)
  inc  $c$
  lsh  $s, 1$
  xor  $s, b$

procedure pop
  dec  $c$
  copy $t, s$
  rsh  $s, 1$
  lsh  $s, 1$
  xor  $t, s$
  rsh  $s, 1$
  return($t$)

procedure size
  return($c$)
```

Figure 6: Implementation of the stack operations from Table 2 using the elementary operations from Table 3

includes arbitrary-length shifts. The drawbacks and advantages of using shifts are discussed at greater length in Parberry [18] (see the *restricted arithmetic instruction set*). The stack operations can then be implemented in constant time as shown in Figure 6, where s is the stack and c is a counter (both initially zero).

We will describe our algorithm on a *synchronous* network, that is, a network with some form of synchronization between the processors. For descriptive purposes, we assume that this is achieved explicitly with a global clock. In practice, synchrony can be achieved implicitly using a message-passing protocol that has a mechanism which allows processors to wait for the arrival of a message from a specific neighbour (for example, using standard locking protocols such as test-and-set or compare-and-swap). We also assume that the operations arrive at every odd-numbered clock tick. In practice this can be implemented by giving processor 1 a clock that runs at least twice as fast as the maximum data input rate. The other processors need have no concept of global time other than that provided from processor 1 via the message-passing protocol.

At each clock tick, two phases of computation are performed consecutively. The first phase is the *input phase*, in which processor 1 processes the current operation (if one is present). A `deletemin` operation is processed as follows. Processor 1 has access to the root of the ragged heap. It deletes the value at the root, replaces it with a hole, and outputs it immediately. It also creates a `fetch` message at the root. An `insert` operation is processed as follows. The inserted value becomes a `replace` message at the root.

The second phase is the *maintenance phase*, during which all processors perform computation to update and maintain the ragged heap. The maintenance associated with the holes and messages is as follows:

The Holes: Every hole in the ragged heap is swapped with its smallest child, unless it is in a leaf or both of its children are holes. (For the purposes of this comparison, a hole is interpreted as having a value larger than any element of the heap.) A record is maintained of the path taken by the hole (so that a value inserted later can catch up with it), as follows. Suppose the hole is in node i, which is stored in processor $p_i = \lfloor \log i \rfloor + 1$. Processor p_i requests the values v_ℓ and v_r in nodes $2i$ and $2i + 1$ (respectively) from processor $p_i + 1$, and determines which is the smaller. If $v_\ell < v_r$, then it passes the hole to node $2i$, places v_ℓ into node i, and `push`es 0 onto its stack. If $v_r < v_\ell$, then it passes the hole to node $2i + 1$, places v_r into node i, and `push`es 1 onto its stack. Each hole carries along with it the position of the bit pushed on the stack (which is the same for all levels). This is called the *address* of the hole.

The `fetch` Messages: Each `fetch` message is passed to the child whose descendants include the last leaf (which may contain a hole). When it reaches the last leaf, the processor in which the last leaf is stored does the following. If the last leaf contains a heap element, then it removes that value from the heap, and places it in an `promote` message. If the last leaf contains a hole, then the hole is removed from the heap, and a `bounce` message is created at that node. The `bounce` message is said to *represent* the hole, and carries with it the hole's address.

The `promote` Messages: Each `promote` message moves one level up the tree. When it reaches the root, it turns into an `insert` operation which is processed in the next even-numbered tick.

The `bounce` Messages: Each `bounce` message moves one level up the tree. When it meets a `replace` message that is heading for the hole that it represents (which condition it checks by checking whether the address of its hole is equal to the `size` of the current stack), the

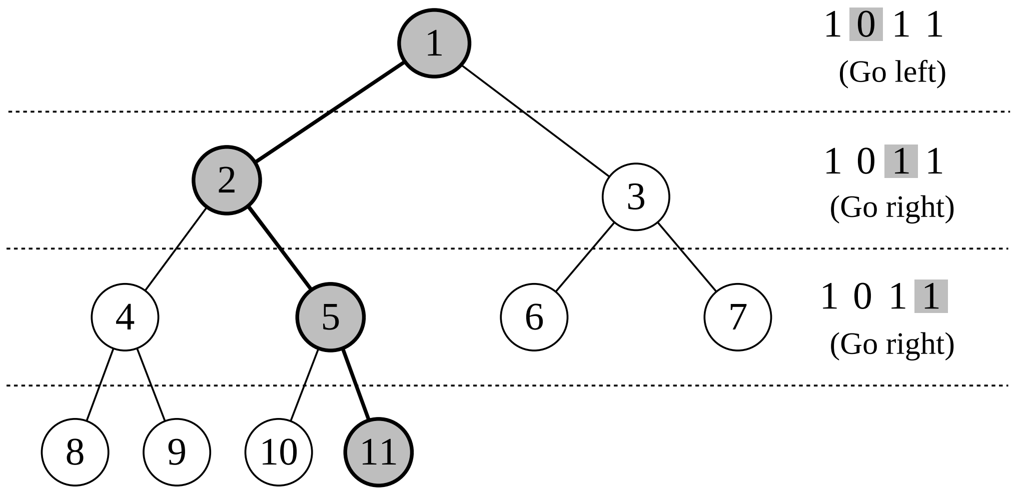

Figure 7: Determination of the last leaf (11 in this diagram). The binary representation of 11 is 1011. The last 3 bits, 011, encode the path from the root to a leaf, 0 meaning "left" and 1 meaning "right". The processor at a given level can determine the correct bit in $O(1)$ time using shifts.

`bounce` message takes on the value carried by the `replace` message, and becomes a a `promote` message. The `replace` message becomes a `purge` message.

The `replace` Messages: Each `replace` message is moved one level down in the tree in pursuit of the most recent previous hole, if there is one, and towards the next unused node otherwise. Suppose the `replace` message v is in node i, which is resident in processor $p_i = \lfloor \log i \rfloor + 1$. Processor p_i compares v to the value v_i in node i. If $v < v_i$, then it swaps v with v_i. If the stack in processor p_i is not empty, then it sends the `replace` message after the most recent previous hole by `pop`ping a bit b from its stack, and passing it to node $2i + b$. Otherwise, if the stack in processor p_i is empty (in which case the stacks of all its predecessors were empty when the `replace` message was there), then it passes the `replace` message to the child whose descendants include the next unused node. If the `replace` message v reaches a node with a hole in it, then the hole is replaced with the value v, and the `replace` message is destroyed. If the `replace` message v reaches the next unused node, then the value v is placed into that node (which then becomes the last used node), and the `replace` message is destroyed.

The `purge` Messages: Each purge message behaves like the `replace` message that spawned it upon collision with a `bounce` message, except that it carries no value and is destroyed when it reaches a leaf.

Routing of `replace` message to the next unused node, and routing of `fetch` messages to the last used node is performed using the technique of Gonnet and Munro [12] and Rao and Kumar [27], by having each processor keep a record of the next unused node, which it updates every time it sees a `replace` message when it has an empty stack. The updating can be carried out in constant time if the processors have arbitrary-length shifts in their instruction sets (as discussed in Section 2). Suppose the last node is number k. Note that k has $\lfloor \log k \rfloor + 1$ bits. The least-significant $\lfloor \log k \rfloor$ bits of k, in most-to-least significant order, encode a path from the root to node number k, with 0 denoting a move to the left child, and 1 a move to the right child (see Figure 7 for an example).

The formal properties of a ragged heap are as follows:

1. The value stored in each node is no larger than the values in its descendants.
2. The root contains neither a hole, nor a message of any type.
3. A hole is called *mobile* if at least one of the children of the node that it is stored in does not contain a hole. There is at most one mobile hole per level.
4. At most $2 \log n$ clock ticks after a hole is created, there will be an `insert` operation.
5. There is at most one of each type of message per level.
6. Each `replace` message is no smaller than the heap value that is stored in the same node.

Lemma 2.1 *The ragged heap properties 1–6 are preserved by an input phase followed by a maintenance phase.*

Proof: Suppose we have a ragged heap that has properties 1–6, and perform an input phase followed by a maintenance phase. We will check the status of each of the properties in turn:

Property 1. The input phase does not affect the property that the value in each node is no larger than the values in its children. The motion of a hole causes its smallest child to be promoted, which (as in the standard heap `deletemin` algorithm) does not affect Property 1. As a `replace` message moves down the tree in pursuit of a hole, it will be found in a situation where it is in a node x, and (by Property 6), it carries a value that is no smaller than the value in the parent of x. In the maintenance phase, its value is compared to the value of x, and exchanged if it is smaller. Thus the value in x is replaced by a value that is smaller (and thus is smaller than its children), yet is larger than the value in its parent. Hence, Property 1 is maintained.

Property 2. The input phase may create a hole and/or downward moving message at the root, but in each case the maintenance phase either moves it down one level in the tree or deletes it. Any message moved upward into the root in the maintenance phase is also destroyed during that phase.

Property 3: Initially, there is at most one mobile hole per level (by Property 3). After the input phase there is still at most one mobile hole per level (by Property 2). Since in the maintenance phase each mobile hole moves one level down, the property that there is at most one mobile hole per level is preserved.

Property 4: After the creation of a hole, it takes at most $\log n$ clock ticks for the `fetch` message to reach the last level of the ragged heap. It takes at most $\log n$ clock ticks for the resulting upward moving message to reach the root. If it is a `bounce` message, then it is guaranteed (by Property 4) to meet a `replace` message that is bound for the hole that it represents. Thus it is guaranteed to be a `promote` message by the time it reaches the root and becomes an `insert` operation. Therefore, a new hole is followed by an `insert` operation at most $2 \log n$ clock ticks later.

Property 5: The argument for Property 5 is similar to that for Property 3. The only difficulty is with the upward moving messages, since the removal of the last node in the last level may cause difficulties. No upward moving messages are created during the input phase. During the maintenance phase, a single upward moving message may be created at the last node. There will not be another upward moving messages at this level since they are created in response to `deletemin` operations, which arrive only at every alternate clock tick. Hence, since upward moving messages are moved one level up in each maintenance phase, Property 5 is maintained.

Property 6: During the maintenance phase, if the value in a newly-arrived `replace` message is smaller than the heap value that is stored in its node, then the two are swapped. Hence, Property 6 is maintained. □

Theorem 2.2 *A ragged heap has properties 1–6 throughout its existence.*

Proof: The proof is by induction on the number of clock ticks. The claim is vacuously true for the empty ragged heap. At each clock tick, Lemma 2.1 ensures that the properties are maintained. □

Corollary 2.3 *A* `deletemin` *operation returns the smallest value in the ragged heap.*

Proof: The values in the ragged heap are distributed amongst the nodes, the `promote` messages, and the `replace` messages. By Property 1, the root contains a value no larger than any of the nodes. By Property 6, the `replace` messages are no smaller than the root. By Property 1, every `promote` message carries a value no smaller than the value in its node. Hence, a `deletemin` operation outputs the smallest value in the ragged heap. □

Corollary 2.4 *The ragged heap algorithm has delay $O(1)$ and uses only $O(1)$ words of word-size $O(\log n)$ per processor in addition to the heap elements.*

Proof: The constant delay-bound follows immediately from Properties 3 and 5. The number of words per processor is $O(1)$, by inspection. The word-size is clearly $O(\log n)$ except for the stack. By Property 4, at most $3 \log n$ clock ticks after a hole is created, a hole will be destroyed (the extra $\log n$ clock ticks is the time it takes the `replace` message created by the insertion to reach a hole). Therefore, since a hole is created at most on every alternate clock tick, there are at most $1.5 \log n$ holes in the ragged heap at any one time, and the stacks therefore require only a single word of $1.5 \log n$ bits each. □

3 Parallel Priority Queues

The purpose of this section is to apply the ragged heap to support a parallel priority queue. It begins by extending the traditional heap to support multiple operations. A *parallel priority queue* is a priority queue in which the `insert` operation inserts p values, and the `deletemin` operation deletes the p smallest values in the priority queue and returns them in no particular order. Parallel priority queues have applications in multiprocessor scheduling, and in parallel graph algorithms (see, for example, Quinn and Deo [24]). A *parallel heap* is a data structure with the same tree structure as a heap, but with p values per node. It also has the property that all of the values in a node are smaller than the values in its children, and all of the values in one sibling are smaller than the values in the other.

Sequential algorithms for parallel heaps are fairly straightforward: An `insert` operation is processed by first placing the p new items in the next unused node d, which we will call *dirty*. Combine the values in d and its parent r and place the smallest p values into r, and the largest p values into d. Combine the values in d and its sibling (if it has one), and place the largest p values into the sibling that had the largest value between them, and the smallest p values into the other sibling. Node r then becomes the dirty node, which is processed in a similar fashion. For example, see Figure 9 which shows an insertion into a 4-node parallel heap with $p = 4$.

The correctness of the parallel heap `insert` procedure is easily verified. Each non-dirty node contains values which are all smaller than the values in its parent, all larger than the values in its children, and either all smaller or all larger than the values in its sibling (if one exists). The

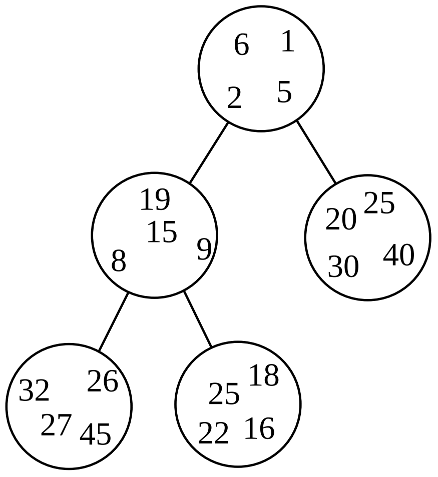

Figure 8: A parallel heap with $p = 4$ values per node. Note that the values in each node are not ordered.

dirty node has values which are smaller than its children. Combining the dirty node d with its parent r may spoil the relationship that r has with its parents and sibling; thus it becomes dirty. Node d may obtain new values from its parent r, which are necessarily smaller than the values of its children, and so d still has values smaller than its children at this point. Combining d with its sibling s has the following effect. Let c_2 be whichever one of d and s has the largest element of the two, and c_1 be the other sibling. Note that before the combination, both c_1 and c_2 have values smaller than their respective children. Then, c_1 will get the p smallest elements, and c_2 the p largest elements of the combined siblings. This establishes the sibling relationship, and does not spoil the relationship of the siblings with their parent, but may spoil the relationships of the siblings with their children. Since c_1 has possibly obtained smaller values from its sibling, these values are still smaller than the values in its children. Now, c_2 may have obtained larger values from its sibling, but its largest value is still the same. Therefore, its values are all still smaller than its children.

A `deletemin` operation is processed by removing the values from the root, and replacing them with the values from the last used node. Call the root *dirty*. Let d be the dirty node. Combine the values in d with those of its smallest child s, and place the smallest p values into d and the largest p values into s. Combine the values in d and its sibling (if it has one), and place the largest p values into the sibling that had the largest value between them, and the smallest p values into the other sibling. Node s then becomes the dirty node, which is processed in a similar fashion. The correctness of this procedure is easily established in a manner similar to the way the correctness of the `insert` procedure was established above.

If the values in each node are sorted, then nodes can be combined by merging them. This gives rise to a sequential algorithm for parallel priority queues with delay $O(p \log(n/p))$, which is best when $p = 1$, and a PRAM algorithm with delay $O(\log p \log(n/p))$ using p processors. Our ragged heap algorithms can be extended to parallel priority queue algorithms in a similar fashion. Processor 1 is replaced by a constant degree network of $O(p)$ processors that sort using the AKS sorting network and Leighton's Columnsort [15]. Every other processor is replaced by a constant degree network of p processors (such as the shuffle-exchange [28] or cube-connected cycles [23]) which implement Batcher's odd-even merging algorithm (Batcher [5]). This gives a delay of $O(\log p)$ on

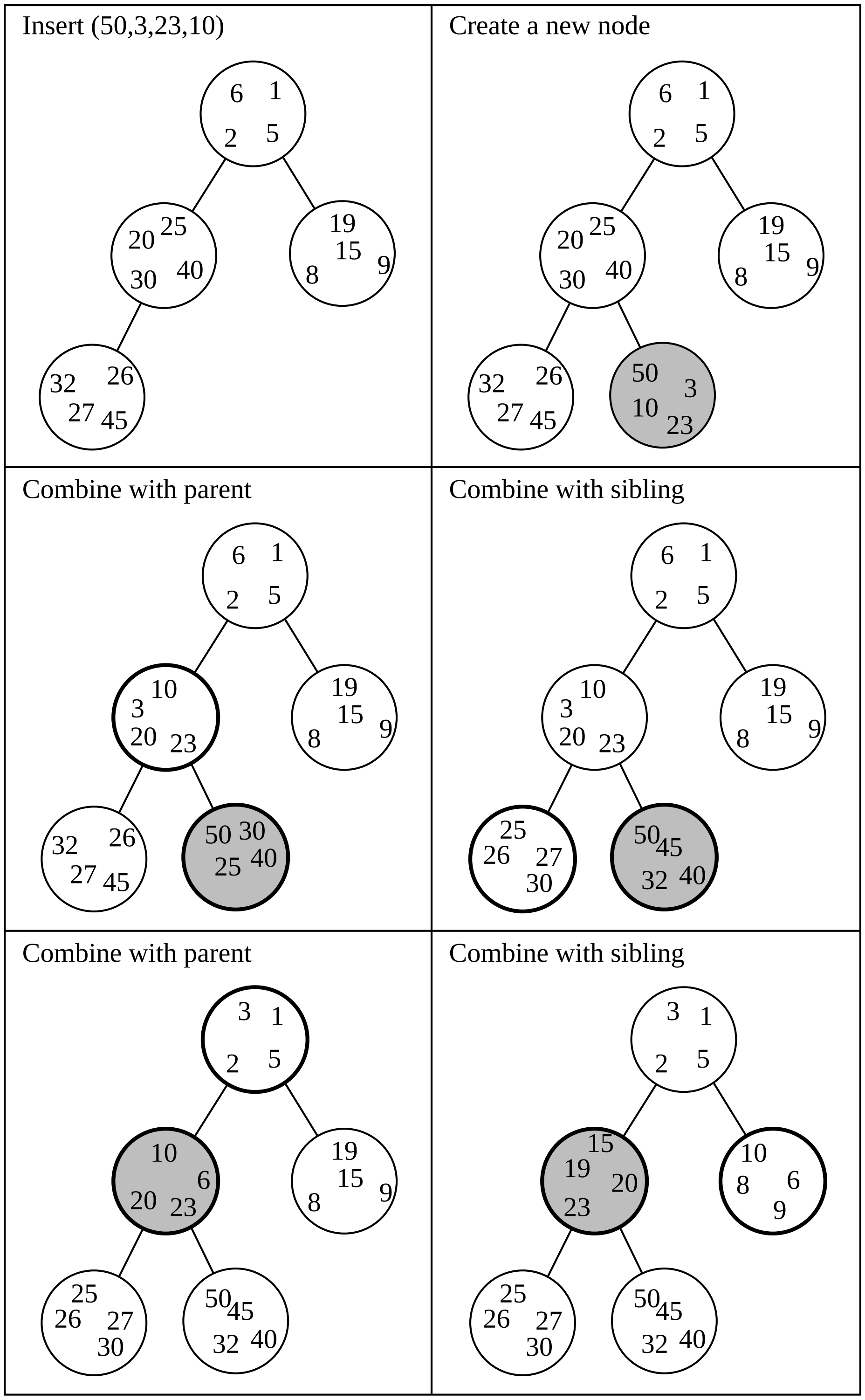

Figure 9: Example of insertion into a parallel heap with $p = 4$. Heavily outlined nodes have been combined. The dirty node is shaded.

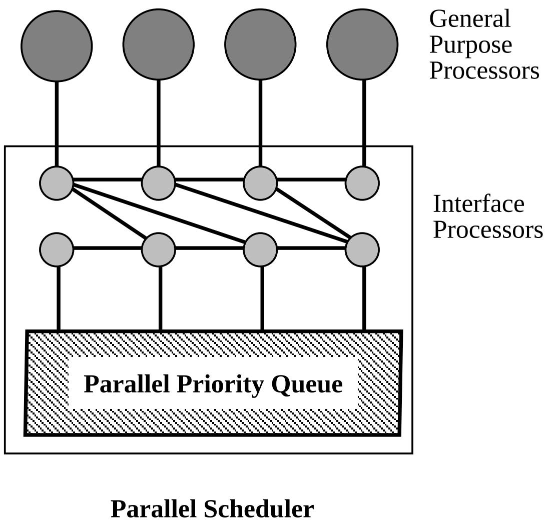

Figure 10: Detail of the scheduler with $p = 4$ general-purpose processors.

a constant degree network of $O(p \log(n/p))$ processors.

4 Load Sharing

Parallel priority queues can be used for load sharing as follows. Tasks are stored in the parallel priority queue with their priority values as the key field, with lower value meaning higher priority. Every $O(\log p)$ steps the general-purpose processors have an opportunity to access the parallel priority queue. Each of these periods is called a *round*. At each round, a processor may choose to insert a task, delete a task or do nothing.

The multiprocessor architecture is as follows (see Figure 10). We use p general-purpose processors that may create and execute tasks of varying priority. These tasks are stored in a parallel priority queue as described in Section 3. Access to the parallel priority is via $2p$ simple special-purpose processors called *interface processors*, which handle the communication between the general-purpose processors and the parallel priority queue, and ensure that the parallel priority queue operations used in each round are legal despite the fact that some processors may insert, some delete, and others not participate at all in the current round. The interface processors are connected using an interconnection pattern capable of optimally executing ASCEND and DESCEND algorithms (see Preparata and Vuillemin [23]) such as the cube-connected cycles or shuffle-exchange. We use $2p$ interface processors for the purposes of description only; p special-purpose processors will suffice (using standard techniques described in Parberry [18, Section 7.4]).

Each round is broken up into an `insert` phase and a `deletemin` phase. In the former, those general-purpose processors that have a job to insert into the parallel priority queue send it to its interface processor. In the latter phase, those general-purpose processors that are idle send a request to their interface processors. In the `insert` phase the interface processors then:

1. Concentrate the new tasks and those that were left over from previous rounds, that is, route them to the leftmost interface processors in Figure 10.

2. If there are at least p tasks, initiate an `insert` operation in the parallel priority queue using the leftmost p tasks.
3. Concentrate the leftover tasks at the rightmost processors.

In the `deletemin` phase the interface processors then:

1. Concentrate the task requests (suppose that there are $r \leq p$ of them).
2. Initiate a `deletemin` operation in the parallel priority queue.
3. Merge the tasks deleted from the parallel priority queue with the leftover tasks from the `insert` phase.
4. The first r interface processors (those that have a task request) route their tasks back to the initiating interface processor by reversing the Concentrate operation from step 1, and pass them to the general-purpose processors that requested them.
5. Concentrate the remaining tasks to the rightmost interface processors.
6. Reinsert the remaining tasks with another `insert` phase.

It can be proved that if g general-purpose processors require tasks, then they get the g available tasks of highest priority. Steps 1 and 3 of the `insert` phase, and steps 1, 4, and 5 of the `deletemin` phase can be implemented in $O(\log p)$ time using the Concentrate procedure from Nassimi and Sahni [17] (using standard techniques to implement it on a bounded degree network described in Parberry [18, Section 7.1]). Step 2 of the `insert` phase can be implemented in time $O(\log p)$ using standard techniques (see, for example, Parberry [18, Section 7.3]) and the parallel priority queue `insert` algorithm from Section 3. Step 2 of the `deletemin` phase takes $O(\log p)$ time using the parallel priority queue `deletemin` algorithm from Section 3. Step 3 of the `deletemin` phase takes $O(\log p)$ time using Batcher's odd-even merging algorithm (Batcher [5]). Therefore, the delay of our algorithm is $O(\log p)$.

5 Halving Networks

One problem with the approach that we have used so far is that the constant multiple in the depth of the AKS sorting network is very large (currently more than 6000, according to Paterson [21]). It is widely conjectured that the size of the constant is due to the extreme difficulty of analyzing the AKS sorting network, rather than being an intrinsic property of the algorithm. Regardless of whether this is so, it is apparent that we can save time by not sorting the values in the nodes. Nodes can be combined directly by a *halving network*, which may be constructed using AKS-type techniques, but at much smaller cost in depth. Since a halver must be placed at every level of the ragged parallel heap, the processor bound rises to $O(p \log p \log n)$. However, since a comparator can be easily pipelined, it is sufficient to use only one halver for every $O(\log p)$ levels, reducing the processor requirement to $O(p \log n)$. This is sufficient if we weaken our load sharing requirements (Assumption 7 of Section 1) to allow an idle processor to be assigned one of the p tasks of highest priority, instead of requiring that in each round, if there are g idle processors, then they must collectively receive the g available tasks of highest priority.

A *halving network* is a comparator network with $2n$ inputs which has the property that the smallest n inputs appear on the n leftmost outputs (and symmetrically, the largest n inputs appear on the rightmost n outputs), in no particular order. It is obvious that the depth of an n input halving network must be $\Omega(\log n)$ (consider an input which consists of the values 1 through $2n$, in which the ith value is i for $1 \leq i \leq n-1$, and the median value n is the jth value for some $n+1 \leq j \leq 2n$). Alekseyev [4] has shown that the size must be $\Omega(n \log n)$, which is interesting in the light of the observation that the decision tree lower bound is only $2n - o(n)$.

It is clear that halving networks of asymptotically optimal depth and size exist, since a sorting network is also a halving network (see also Ajtai, Komlós and Szemerédi [3]), Paterson [21] has an improved version of the AKS sorting network with depth $6070 \log n$. His algorithm can be used to construct shallower halving networks by stopping as soon as the root bag becomes empty. This, with Paterson's best choice of parameters, leads to halving networks of depth less than $1869 \log n$. A significant improvement can be gained by further careful modifications, as follows. We assume that the reader is familiar with the terminology, notation, and techniques of Paterson [21]. Since a full description and analysis of the algorithm would take many pages, and much of it would be very similar to [21], we will limit ourselves to describing the principal changes that need to be made.

There is no longer any need for the complete binary tree of bags used by Paterson. All nonroot bags need only have a single child. As before, the root bag γ-halves its values and passes the "smaller" half to its left child, and "larger" half to its right child. Each nonroot bag ϵ-nearsorts its values, and passes a λ fraction of them to its parent, the remainder to its child. Descendants of the left child of the root pass their "largest" values up, whilst descendants of the right child pass their "smallest" values up.

Let $V(k,t) = A^{k-1}\nu^t n$ denote the capacity of a level k bag at time t. At time t, the root receives $\lambda V(2, t-1)$ values from each child. Therefore, we require that

$$2\lambda V(2, t-1) \leq V(1,t),$$

that is,

$$2\lambda A \leq \nu. \tag{1}$$

A nonroot bag at level k receives $\lambda V(k+1, t-1)$ values from its child, and at most $(1-\lambda)V(k-1,t)$ values from its parent. Therefore, we require that

$$(1-\lambda)V(k-1,t-1) + \lambda V(k+1,t-1) \leq V(k,t),$$

that is,

$$(1-\lambda)/A + \lambda A \leq \nu. \tag{2}$$

Equations (1) and (2) take the place of Paterson's inequality (1).

Since all of the values are passed up from one side of the bag, we can take $\lambda = 1/2^p$, instead of Paterson's $1/2^{p-1}$, and we need only insist that

$$\mu \leq \lambda \tag{3}$$

instead of Paterson's inequality (3): $\mu \leq \lambda/2$.

We can redefine the *strangeness* of a value to be zero if it is on the correct side of the tree, and one less than the level of the bag that it is in, otherwise. In the analysis of the number of values of strangeness greater than 1, since each nonroot bag has only one child to receive strangers from and the root can contain no strangers, Paterson's inequality (4) becomes

$$A^2\delta^2 + \epsilon \leq \delta A \nu. \tag{4}$$

Only the children of the root can contain values of strangeness 1. Each can receive values of strangeness 1 from its child (where they had strangeness 2), from mistakes in the root's γ-halving, and from the existence of an unbalanced number of left-seeking and right-seeking values in the root. There can be at most $\mu\delta A^2\nu^{t-1}$ strangers of the first kind, at most $\gamma\nu^{t-1}n/2$ of the second kind, and, by a similar argument to Paterson's (remembering that we are dealing with the root instead of an arbitrary bag), at most

$$\frac{\mu\delta A^2\nu^{t-1}n}{1-A^2\delta^2}$$

of the third kind. Therefore, Paterson's inequality (5) becomes

$$\delta A + \frac{\gamma}{2A\mu} + \frac{\delta A}{1-A^2\delta^2} \leq \nu. \tag{5}$$

Our separators are also less complicated than Paterson's since they need only be γ-halvers in the root and ϵ-nearsorters elsewhere (and thus no longer need to serve double duty). The depth of our separator is thus

$$\max\{C(1-\eta-2\mu,\gamma), \sum_{i=1}^{p} C(2^i\mu,\epsilon_p)\},$$

where

$$\eta = \frac{\mu\delta A^2}{1-\delta^2 A^2}.$$

The following choice of parameters:

p=3	ϵ_1=0.0325	λ=0.125	μ=0.0419
ϵ=0.11082	ϵ_2=0.03623	A=3.0039	δ=0.0672
γ=0.02988	ϵ_3=0.04209	ν=0.75098	

satisfies all of the above inequalities, and results in approximately $2.42\log n$ iterations, with a γ-halver of depth 135 used in the root, and an ϵ-nearsorter made up of ϵ_i-halvers for $1 \leq i \leq 3$, each of depth 45. Thus the depth of the resulting halving network is less than $327\log n$.

6 Conclusion and Open Problems

We have presented efficient algorithms for processing priority queue operations in parallel and described how these algorithms can be used to design an architecture for load sharing in multiprocessor systems. Our algorithms are simple and practical except for their reliance on AKS or AKS-like constructions. It remains to design algorithms that are free of the excessive constant-multiple overhead that this entails.

We have demonstrated a method for the construction of halving networks that are much shallower than the best currently known sorting networks. Our analysis of halving networks is probably pessimistic. We conjecture that halving networks of depth less than $10\log n$ exist. It is an open question whether there is an analog of Leighton's Columnsort [15] for halving networks, which would give an n processor constant degree network which could halve n values in time $O(\log n)$.

Our parallel priority queue algorithms are not *work optimal* (work is defined to be the product of the time and processor bound), since the insertion of n values into a parallel priority queue takes work $O(n\log n\log p)$, which is asymptotically larger than the decision tree lower bound of $\Omega(n\log n)$ (for $p = o(n)$). It remains to be seen whether work optimal bounded degree algorithms are possible.

For a parallel priority queue to be useful in practice for multiprocessor scheduling, it must support dynamic updating of priorities, and real-time cancellation of tasks by the general-purpose processors. The efficient implementation of these operations remains an open problem.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM Press, 1983.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–48, 1983.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. Halvers and expanders. In *33rd Annual Symposium on Foundations of Computer Science*, pages 686–692. IEEE Computer Society Press, 1992.

[4] V. E. Alekseyev. Sorting algorithms with minimum memory. *Kibernetika*, 5:99–103, 1969.

[5] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, April 1968.

[6] J. Biswas and J. C. Brown. Simultaneous update of priority structures. In *Proc. of the 1987 International Conference on Parallel Processing*, pages 124–131. Penn State Press, 1987.

[7] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computer systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[8] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1–3):2–22, 1985.

[9] S. K. Das and W. B. Horng. Managing a parallel heap efficiently. In *Proc. of the 1991 Conference on Parallel Architectures and Languages, Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 270–287. Springer Verlag, 1991.

[10] N. Deo and S. Prasad. Parallel heap. In *Proc. of the 1990 International Conference on Parallel Processing*. Penn State Press, 1990.

[11] Z. Fan and K. H. Cheng. A simultaneous access priority queue. In *Proc. of the 1989 International Conference on Parallel Processing*, volume 1, pages 95–98. Penn State Press, 1987.

[12] G. H. Gonnet and J. I. Munro. Heaps on heaps. *SIAM Journal on Computing*, 15(4):964–971, 1986.

[13] D. W. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, 1989.

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

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

[16] J. I. Munro and E. L. Robertson. Parallel algorithms and serial data structures. In *Proc. 17th Annual Allerton Conference on Communication, Control, and Computing*, pages 1–6, 1979.

[17] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30(2):101–106, February 1981.

[18] I. Parberry. *Parallel Complexity Theory.* Research Notes in Theoretical Computer Science. Pitman Publishing, London, 1987.

[19] I. Parberry. Some practical simulations of impractical parallel computers. *Parallel Computing*, 4(1):93–101, 1987.

[20] I. Parberry. Load sharing with parallel priority queues (Extended Abstract). Technical Report CRPDC-91-10, Center for Research in Parallel and Distributed Computing, Dept. of Computer Sciences, University of North Texas, October 1991.

[21] M. S. Paterson. Improved sorting networks with $O(\log n)$ depth. *Algorithmica*, 5(4):75–92, 1990.

[22] N. Pippenger. On simultaneous resource bounds. In *20th Annual Symposium on Foundations of Computer Science*, pages 307–311. IEEE Computer Society Press, 1979.

[23] F. P. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, 1981.

[24] M. J. Quinn and N. Deo. Parallel graph algorithms. *ACM Computing Surveys*, 16(3):319–348, September 1984.

[25] M. J. Quinn and Y. B. Yoo. Data structures for the efficient solution of graph theoretic problems on tightly-coupled MIMD computers. In *Proc. of the 1984 International Conference on Parallel Processing*, pages 431–438. IEEE Computer Society Press, 1984.

[26] A. G. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, pages 185–194. IEEE Computer Society Press, 1987.

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

[28] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, 1971.

[29] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[30] Y.-T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C–34(3):204–217, 1985.

[31] J. W. J. Williams. Algorithm 232 (Heapsort). *Communications of the ACM*, 7:347–348, 1964.