The Journal of Supercomputing, 6, 87–98 (1992)

Parallel Heap: An Optimal Parallel Priority Queue

NARSINGH DEO*
Computer Science Department, University of Central Florida, Orlando, FL 32816, deo@eola.cs.ucf.edu

SUSHIL PRASAD*
Mathematics and Computer Science Department, Georgia State University, Atlanta, GA 30303,
matskp@gsusgi1.gsu.edu

(Received November 1990; final version accepted April 1992.)

Abstract. We describe a new parallel data structure, namely *parallel heap*, for exclusive-read exclusive-write parallel random access machines. To our knowledge, it is the first such data structure to efficiently implement a truly parallel priority queue based on a heap structure. Employing p processors, the parallel heap allows deletions of $\Theta(p)$ highest priority items and insertions of $\Theta(p)$ new items, each in $O(\log n)$ time, where n is the size of the parallel heap. Furthermore, it can efficiently utilize processors in the range 1 through n.

Keywords. Parallel data structure, heap, priority queue, optimal parallel algorithm, parallel random access machine.

1. Introduction

Priority queues have traditionally been used for applications such as branch-and-bound algorithms [Mohan 1983; Rao et al. 1987], discrete-event simulation [Fujimoto 1990], shortest path algorithms [Quinn and Deo 1984], multiprocessor scheduling, and sorting. A priority queue is an abstract data structure that allows deletion of the highest priority item and insertion of new items. Typically, priority-queue-based applications consist of repeated cycles: An item of the highest priority is deleted from the priority queue; some processing is done on that item, which leads to the creation of zero or more new items, which in turn are inserted into the priority queue; and this cycle repeats. A *heap*, which is a complete binary tree such that the priority of the item at each node is higher than that of the items at its children, provides an optimal implementation of a priority queue on uniprocessor computers—deletion of the highest priority item and insertion of a new item can each be accomplished on $O(\log n)$ time on an n-item heap.

To effectively use a parallel computer for priority-queue-based applications, a parallel version of the priority queue is required, which would allow multiple inserts and deletes concurrently. There have been various parallelization efforts of the heap for shared-memory parallel computers [Biswas and Brown 1987; Jones 1989; Quinn and Deo 1984; Quinn and Yoo 1984; Rao and Kumar 1988]. Rao and Kumar [1988] proposed a practical scheme that allowed $O(\log n)$ processors to be active simultaneously on a heap. Biswas and Browne [1987] also presented a scheme that could keep $O(\log n)$ processors active on a heap. Jones

*This work was supported by U.S. Army's PM-TRADE contract N61339-88-g-0002, Florida High Technology and Industry grant 11-28-716, and Georgia State University's internal research support during spring and summer quarters, 1991.

[1989] too presented a scheme to keep this many processors active, but the heap access time in his scheme was O(log n) in only an amortized sense. Quinn and Yoo [1984] gave the first pipelined allocation of O(log n) processors to perform deletions from a heap.

We note that in all these implementations, only a maximum of O(log n) processors could be utilized efficiently. Others relied on specific hardware implementations. For example, Leiserson [1981] implemented a parallel priority queue as a linear array of VLSI processing elements. A delete or an insert could be performed every clock period. However, n deletes or inserts required a total of O(n) time. Therefore, using O(n) processing elements to achieve such performance was inefficient. In [Fan and Cheng 1989] a network for a simultaneous access priority queue was presented. This priority queue was also a linear array; however, each array element was w processing elements wide. Additionally, the front of the linear array had a sorting network comprising w^2 processing elements. Thus, Fan and Cheng's priority queue of size n used a total of $n + w^2$ processing elements and was able to perform w accesses in O(log w) time. Therefore, this data structure was inefficient for all values of w, $1 \leq w \leq n$. However, it applied the concept of storing more than one item per node, and that of using sorting and merging for maintaining a priority queue. Wegner and Teuhola [1989] have also described the usage of replacing key comparisons with merge operations on pages of a heap in the context of an external heapsort. Similar concepts were developed and implemented independently for the parallel heap data structure by Prasad [1990].

Although not directly relevant to this paper, we note the parallelizations of nonheap data structures such as binary search trees, height-balanced trees, and 2-3 trees for database and related applications [Ellis 1980; Manber 1984; Paul et al. 1983; Quinn and Yoo 1984]. While Paul et al. [1983] could optimally use O(n) processors to search, delete, and insert on a 2-3 tree for dictionary applications, others could use only up to O(log n) processors.

The existing priority queue schemes achieve very limited parallelism. We present a new data structure, *parallel heap*, which achieves a much greater degree of parallelism. It efficiently utilizes p processors, for $1 \leq p \leq n$, and is able to perform $\Theta(p)$ inserts and $\Theta(p)$ deletes in O(log n) time. This, in turn, implies up to an O(p) speedup for priority-queue-based applications.

Our work builds upon and generalizes the existing results on heaps, specifically those obtained by Rao and Kumar [1988] and Quinn and Yoo [1984]. Readers are also referred to earlier versions of this paper for details of the data structure and an elaborately worked-out example [Deo and Prasad 1990; Prasad 1990; Prasad and Deo 1992]. Recently, Pinotti and Pucci [1991] have developed a data structure, n-Bandwidth heap, that is similar to a parallel heap, and can employ O(n) processors on a concurrent-read exclusive-write PRAM. Das and Horng [1991] have provided some analysis on space requirements and processor management on parallel heaps. Our work has been independent of these recent developments.

We implement the parallel heap on an exclusive-read exclusive-write parallel random access machine (EREW PRAM) [Karp and Ramachandran 1990]. This machine model consists of p identical processors, P_0 through P_{p-1}, and a shared memory. Each processor has its own program. There is global synchronization among the processors via a central clock. A processor can execute a simple instruction or read from or write into a shared-memory cell in one clock period. Furthermore, during any clock period, a shared-memory cell is accessed by no more than one processor for reading or writing.

All the logarithms in this paper are assumed to be in base 2. We use $\log^* n$ to denote the smallest integer i such that $\log^{(i)} n \leq 1$, where $\log^{(i)} n = \log(\log^{(i-1)} n)$ and $\log^{(0)} n = n$. For order notation, as usual, we will use Θ to denote "order equal to" and O to denote "order no greater than." The rest of this paper is organized as follows: Section 2 describes the parallel heap and its basic operations, deletion and insertion. Section 3 describes a pipelined implementation of the parallel heap and discusses its time complexity. Section 4 discusses some variants of parallel heaps. Section 5 contains some concluding remarks.

2. Data Structure and Basic Operations

A parallel heap with node capacity $r \geq 1$ is a complete binary tree such that

- each node contains r items (except for the last node, which may contain fewer items), and
- all r items at a node have values less than or equal to the values of the items at its children.

When the second constraint holds at a node, the node is said to satisfy the *parallel heap property*, which is an extension of the "heap property" for conventional heaps. The parallel heap property, when satisfied at all the nodes, ensures that the root node of size r of a parallel heap contains the smallest r items (the r highest priority items). A parallel heap with node capacity r employs r processors. In the discussions that follow, we will assume that the node capacity of a parallel heap is r, that r processor are available, and that the size of the parallel heap—the total number of items in the heap—is n. Furthermore, although the parallel heap property does not mandate it, in the following discussion we will assume that the r items at each node are kept sorted in ascending order. Later, we will also discuss a different version of the parallel heap without an ordering among the items of its individual nodes.

A parallel heap is implemented as an array of records, each record being a node. Each node is simply an array of size r. Thus, a parallel heap is just a linear array in which the first r cells are collectively called node 1, the next r cells, node 2, and so on. Therefore, the i-th item of a parallel heap is actually item $(i - (\lceil i/r \rceil - 1) * r)$ of node $\lceil i/r \rceil$. An array representation, as for conventional heaps, allows an implicit binary tree representation of a parallel heap. We will often refer to the bottom levels of a parallel heap (seen as a binary tree) as its rear nodes (seen as an array).

We will first describe how the two basic operations, deletion and insertion, are implemented on a parallel heap. Thereafter, we will describe their pipelined implementation, which ensures the optimality of a parallel heap.

2.1. Deletion

To delete the smallest r items from a parallel heap, one simply deletes the r items at the root. This, however, leaves a root empty. Therefore, unless the parallel heap has become completely empty, a *delete-update process* follows a deletion operation to restore the parallel heap.

Delete-Update Process

1. Fetch the last r items from the bottom of the parallel heap as *substitute* items, and place them at the root, sorted in an ascending order: If the last node has r items, then those are the items fetched as substitute items. They are already sorted. However, if the last node has fewer than r items, say k items, then these k items together with the largest $r - k$ items of the second-last node are fetched as substitute items. Before being placed at the root, these two sets of items are merged.
2. Sink the substitute items down level by level until the parallel heap property is satisfied: Start with the root as the current node. Repeat the following steps until either the parallel heap property is satisfied at the current node or the current node is a leaf node:
   a) Let the value of the largest item of the left child be x and that of the right child be y. Merge the items at the current node and those at its two children into a single list, and place the smallest r items in the current node. This satisfies the parallel heap property at the current node.
   b) If $x \geq y$, then place the next smallest r items in the left child and the rest of the items in the right child. Observe that the largest item in the left child now—the $(2r)$-th item of the merged list—is no greater than x. Thus the parallel heap property is satisfied at the left child. The right child becomes the current node (for adjusting the heap). Otherwise, if $x < y$, the roles of the left and the right child are switched; that is, the second set of the smallest r items is placed in the right child, and the left child becomes the current node.

Remarks

1. A delete-update process initiated at the root of a parallel heap remains contained in exactly one node at each level. This crucial fact was also found useful in the context of external heapsorts by Wegner and Teuhola [1989].
2. Step 1 of the delete-update process requires O(log r) time, assuming that Bilardi and Nicolau's parallel bitonic merging [1989], which requires $(r/\log r)$ processors and O(log r) time, is employed. Similarly, O(log r)-time steps 2a and 2b are repeated for $\lceil \log (\lceil n/r \rceil + 1) \rceil$ levels of the parallel heap for a total of O(log n log r) time.
3. To delete fewer than r items from a parallel heap, say k items, only the first k items are deleted from the root, and only k substitute items are fetched from below.

2.2. Insertion

An insertion operation is also implemented top to bottom to facilitate pipelining. The idea of insertion at the root has been used earlier by Gonnet and Munro [1986] and Rao and Kumar [1988].

Assume that there are r new items to be inserted. These are first sorted. If the last node is full (it has r items), then the r new items to be inserted are targeted for a new node following the last node. The new items arrive at their *target node* after filtering through the parallel heap level by level, starting at the root and moving along the unique *insertion*

path from the root down to the target node. This task is accomplished by initiating an *insert-update process* at the root.

When the last node has $k < r$ items, two insert-update processes are initiated at the root—the first carrying the smallest $(r - k)$ new items targeted for the last node itself, and the second carrying the rest of the new items targeted for a new node succeeding the last node.

Insert-Update Process

An insert-update process initiated at a node carrying $k \leq r$ new items is processed as follows.

1. Repeat until the target node is reached:
   a) Merge the k items being carried with the insert-update process with the r items of the current node to obtain a single sorted list.
   b) Place the smallest r items in the current node, and carry over the larger k items with the insert-update process down to the child lying on the insertion path leading to the target node. That child becomes the current node in the next iteration.
2. Merge the k items being carried with the insert-update process with the items at the target node.

Remarks

1. An insert-update process does not require additional space for storing its items: Empty locations in target nodes can be used as temporary storage space. Thus, when an insert-update process carrying k items is initiated at the root of a parallel heap of size n, the insert-update process uses cells $n + 1$ through $n + k$ of the parallel heap to "carry" these k items. To access these items, the insert-update process needs only two variables: the starting cell, $n + 1$, and the number of items being carried, k. It should be understood, however, that these k items do not become a part of the parallel heap until their insert-update process terminates after reaching its target node. Thus, neither a delete- nor an insert-update process requires any additional space to store its items.
2. An insert-update process does not need the insertion path explicitly. From the current node, to reach the target node i levels below the current node, move left if the i-th bit from right to left in the binary representation of the target node number is a 0; else move right.
3. When as a result of an insertion operation two insert-update processes are initiated at the root, they are carried down together level by level, but are processed one after another in the order of increasing target nodes at each level. However, if both insert-update processes are currently at the same node, both can be processed together, although they may go to different children.
4. Sorting the r new items to be inserted can be done in $O(\log r)$ time with r processors by using Cole's parallel sorting [1988]. Carrying out one or more insert-update processes requires $O(\log r)$ time steps for each of the $O(\log (n/r))$ levels of a parallel heap for a total of $O(\log n \log r)$ time.
5. The insertion of $k < r$ new items follows the same procedure with obvious changes.

2.3. *Combined Insertion and Deletion*

A parallel heap will normally be used as a priority queue in which r processors will synchronously delete the smallest r items from the parallel heap and process those items, thereby producing some new items as a result. We will assume that the processing of a single item creates at most two new items (the constant 2 is chosen only to aid our presentation, which can be extended to larger constants easily). Therefore, up to $2r$ new items would be produced that must be reinserted into the parallel heap. Thereafter, the smallest r items would again be deleted from the parallel heap for the subsequent cycle. Thus the processors repeatedly go through two phases: the *think phase*, which is the processing of deleted items, and the *insert-delete phase*, which is the insertion of new items into the parallel heap, and the deletion of the smallest r items. Instead of first inserting up to $2r$ new items and then deleting the smallest r items, the two operations are combined as follows.

1. Sort the new items, and then merge them with the items at the root to obtain a single sorted list of up to $3r$ items.
2. Delete the smallest r items for the subsequent think phase. These are clearly the smallest r items of all the existing items.
3. The remaining items number at most $2r$. Since the root has become empty, a delete-update process needs to be initiated at the root. If the remaining items number at least r, the smallest r of them is placed at the root as the substitute items and a delete-update process is initiated. Additionally, up to r largest leftover items are inserted into the parallel heap by initiating up to two insert-update processes at the root (as explained in the previous subsection). All these three update processes are carried down the parallel heap together while executing the delete-update process followed by the insert-update processes at each level. However, if the remaining items number $k < r$, the last $(r - k)$ items of the parallel heap are fetched. These r items are then sorted and placed at the root as substitute items. Finally, a delete-update process is initiated at the root.

The total time required for an insert-delete phase is still $O(\log n \log r)$ using r processors. We use pipelining to accomplish an insert-delete phase in $O(\log n)$ time.

3. Pipelined Implementation of the Parallel Heap

The idea of lazy evaluation carries over well to parallel heaps. After an insert-delete phase, all one really needs is to have the root of the parallel heap updated—ready for the next cycle of think-insert-delete. This task is accomplished during an insert-delete phase by executing the newly initiated delete- and insert-update processes at the root and sinking them down by just one level.

To correctly implement this idea, observe that the execution of the delete-update process at the root involves the root and its children, so one must update the root's children before updating the root itself. Following this reasoning, all the past update processes from previous think-insert-delete cycles at the lower levels of the parallel heap must also be executed and sunk down by a level each. To avoid the apparent sequentiality in this procedure, a parallel

heap is constrained to have update processes only at its odd levels at the beginning of each insert-delete phase (assuming the root is at level 0). Then, during an insert-delete phase, these update processes are first moved down to the even levels, in parallel. Next, the update processes at the even levels, which include the ones newly initiated at the root, are moved down to the odd levels, thus leaving the root updated for the subsequent cycle.

3.1. Additional Data Structures

Due to pipelining, particularly that of insert-update processes, at no instance are the bottom levels of a parallel heap completely updated. That is, there will usually be insert-items moving down along their insertion paths that will reach their target nodes only after a certain number of think-insert-delete cycles, depending on the length of their insertion paths. The insert-update processes carry these insert items by temporarily placing them at the rear of the parallel heap beyond its updated items (Section 2.2). To distinguish the updated items that actually belong to the parallel heap from those insert items being carried by insert-update processes, two variables are maintained: (1) n', the number of items actually in the parallel heap (only the insert items lie beyond cell n') and (2) n, the total number of existing items (the insert items placed after the n'-th item of the parallel heap number exactly $n - n'$, occupying cells $n' + 1$ through n).

When a delete-update process needs $k \leq r$ items as substitute items, it fetches the last k items of the parallel heap, those lying at cells $(n - k + 1)$ through n. Variable n is thus reduced; n' is reduced only if $n - n' < k$, that is, when there are not enough insert items. Those insert items thus fetched away as substitute items are removed from their respective insert-update processes. Initiation of an insert-update process increases n, and termination of a insert-update process after it reaches its target node increases n'. Observe that, because of the way the insert-update processes are initiated and the order in which they are executed at each level, the insert-update processes reach the bottom of the parallel heap in ascending order of their target nodes.

Next, to access the three update processes at each level during an insert-delete phase, three flags are used per level: (1) *delete-flag*, which, if not -1, points to the node at that level at which a delete-update process is to be carried out, and (2) *insert-flag1* and *insert-flag2*, which, if not -1, point to the nodes for the first and the second insert-update processes, respectively. Additionally, each insert flag is associated with two variables: the starting cell number of the insert items stored at the bottom (rear) of the parallel heap, and the number of those items being carried.

3.2. Insert-Delete Phase

Here are the essential details of the pipelined manipulation of a parallel heap. These details are enough to deduce an $O(\log n)$ time bound for an insert-delete phase. For a more detailed exposition, readers are referred to [Prasad 1990].

1. Move update processes at odd levels down by a level to even levels: Form $m = \lceil \log r \rceil$ groups of processors, $g_0, g_1, g_2, \ldots, g_{m-1}$, each comprising $\lfloor r/\log r \rfloor$ processors, for $r > 1$ (if $r = 1$, then let $m = 1$). Out of $l = \lceil \log (\lceil n/r \rceil + 1) \rceil$ levels of the parallel heap, there are $l_o = \lfloor l/2 \rfloor$ odd levels containing update processes.

   For each $i = 0$ through $\lceil l_o/m \rceil - 1$, carry out the following:
   a) Assign group g_j, for all j, 0 through $m - 1$, to level $2(j + i * m) + 1$.
   b) Each group executes the update processes at its assigned level in parallel; thereafter, the flags at the assigned levels are cleared (set to -1):
      (1) If the delete-flag is not -1, execute the delete-update process at the node pointed to by the delete-flag (steps 2a and 2b of the delete-update process in Section 2.1). If not terminated, the delete-update process moves down to one of the children. The delete-flag of the lower level now points to that child node.
      (2) Next, if insert-flag1 is not -1, execute an insert-update process at the node specified by insert-flag1 (steps 1a and 1b of the insert-update process of Section 2.2). Thus, unless the target node is reached, the insert-update process is moved down to the child along the insertion path, and insert-flag1 of the lower level points to this child. If the target node is reached, increase n' by the number of items inserted (only the group assigned to the last level of the parallel heap will update n'; therefore, there is no write conflict). Similarly, if insert-flag 2 is not -1, execute the corresponding insert-update process.
2. Sort up to $2r$ newly created items. Merge the new items with the r items at the root into a single sorted list. Delete the smallest r of this list for the following think phase.
3. Initiate update processes at the root:
   a) If the number of remaining items is $k \geq r$, then place the smallest r of the leftovers into the root as substitute items. Place the remaining $k - r$ largest items as insert items at the rear of the parallel heap at cells $n + 1$ through $(n + k - r)$, and construct up to two insert-update processes at the root by setting, for each update process, the two variables at level 0 indicating the starting cell location of the insert items and their number (two insert-update processes are needed only if the insert items span two nodes). Increase n by $(k - r)$.
   b) Else, $(r - k)$ substitute items are needed from the last nodes. Fetch this many items from cells $(n - (r - k) + 1)$ through n. Reduce n by $(r - k)$. Reduce n' by the integer $(r - k - (n - n'))$ if it is positive. Sort the k remaining items and $(r - k)$ substitute items, and place these r items at the root.
   c) Set the delete-flag of level 0 to point to the root, indicating the presence of a new delete-update process. Similarly, set insert-flag1 and insert-flag2 to point to the root, depending on how many insert-update processes were constructed.
4. Process the update processes at the even levels of the parallel heap and move them down one level as was done for the odd levels in step 1. This updates the root also. Before executing an insert-update process in this step, check if some of its insert items have been fetched away by the newly initiated delete-update process as substitute items in the previous step; that is, check if its insert items lie at cells beyond the current value of n.

3.3. Time Complexity of an Insert-Delete Phase

Steps 2 and 3 are clearly O(log r)-time tasks using r processors. In step 1 each group of O(r/log r) processors can execute the update processes at their assigned levels in O(log r) time by employing Bilardi and Nicolau's bitonic merging [1989]. Since each such group has to update O (log(n/r)/log r) levels, the total time required by step 1 is O(log (n/r) + log r). The log r factor is added because it is a lower bound on the time taken by step 1—even if just one level needs updating, at least O(log r) time will be required. Similarly, step 4 requires O(log (n/r) + log r) time. Thus an insert-delete phase requires a total of O(log n) time.

Therefore, a parallel heap of node capacity r containing n items can perform O(r) deletions and insertions in O(log n) time by employing r processors. Such a performance yields linear speedup when compared to a sequential heap performing one deletion or insertion every O(log n)-time steps. In addition to the optimality of the parallel heap data structure, we note its scalability also: For any number of processors r between 1 and n, there is an optimal parallel heap with node capacity r.

4. Variants of Parallel Heaps

Two important versions of parallel heaps, differing mainly in the algorithms employed for heap manipulation, are as follows.

4.1. Parallel Heap without Cole's Sorting Algorithm

The large constant of proportionality of Cole's mergesort algorithm [1988] could be a cause for concern. Suppose one used a slower parallel sorting algorithm, one that sorts r items in O($\log^2 r$) time using r/log r processors (similar to those described by Bilardi and Nicolau [1989] or Hagerup and Rub [1989]). Observe that although the node capacity of the parallel heap is r, only r/log r processors are available to maintain it. Sorting r items in step 2 of an insert-delete phase would thus require O($\log^2 r$) time. In step 1, r/log r processors update O(log (n/r))levels of the parallel heap in O(log (n/r) log r) time. Steps 3 and 4 have the same time bounds as steps 2 and 1, respectively. Thus, the total time required for an insert-delete phase is O (log(n/r) log r + $\log^2 r$). Therefore, since the sequential time to perform r equivalent deletions or insertions on a serial access heap is O(r log n), a linear speedup is attained. Furthermore, since r can vary up to n, this version of a parallel heap is optimally scalable up to n/log n processors. If given p as the number of processors available, the appropriate choice for the node capacity is $r = p \log p$.

4.2. Parallel Heap with Items at Each Node Unsorted

There are some applications, such as branch-and-bound algorithms, that do not need the smallest r items deleted from the parallel heap in each insert-delete phase to be sorted.

For such applications we no longer need to keep the items at individual nodes of a parallel heap sorted. Therefore, Cole's sorting and Bilardi and Nicolau's merging algorithms are replaceable with any optimal parallel selection algorithm to carry out all the steps of an insert-delete phase.

Suppose one uses Cole's selection algorithm [1986], which can perform selection on $O(r)$ items in $O(\log r \log^* r)$ time employing $r/\log r \log^* r$ processors. With p available processors, set the node capacity to $r = p \log^* p$. Then steps 2 and 3 of the insert-delete phase are bounded by $O(\log r \log^* r)$ time. For step 1, form $p \log r \log^* r/r$ groups of $r/(\log r \log^* r)$ processors each. These groups can update $\log (n/r)$ levels of the parallel heap in $O((r/p) \log(n/r) + \log r \log^* r)$ time. The same time bound applies for step 4 as well. Therefore, this is also the overall time bound for an insert-delete phase. Compared to the sequential time of $O(r \log n)$ for $O(r)$ operations, the speedup attained is linear as long as $p \leq r/\log^* r$. The number of processors has indeed been chosen to satisfy this optimality condition. This version of parallel heap is optimally scalable up to $n/\log^* n$ processors.

Similarly, if one employed Akl's optimal selection algorithms [1984], which can perform selection on $O(r)$ items in $O(r^\epsilon)$ time using $r^{1-\epsilon}$ processors for $0 < \epsilon < 1$, a linear speedup is achieved by scaling the number of processors p up to $n \log n/n^\epsilon$ and by choosing the node capacity to be $r = (p/\log n)^{\epsilon-1}$.

5. Conclusion

We have presented a new parallel data structure, namely parallel heap, that allows $\Theta(p)$ deletions and $\Theta(p)$ insertions in $O(\log n)$ time, where p is the number of processors and n is the size of the parallel heap. The number of processors can optimally vary from 1 through n. Therefore, the parallel heap is superior to the existing parallelization schemes of a heap, which can use only $O(\log n)$ processors efficiently.

In a practical implementation of the parallel heap, Cole's mergesort algorithm, which has a large constant of proportionality, can be replaced by slower sorting algorithms such as those described in [Bilardi and Nicolau 1989] and [Hagerup and Rub 1989] while optimally using up to $n/\log n$ processors.

We have also described a simpler version of the parallel heap applicable in branch-and-bound and related algorithms that keeps the smallest r items unsorted. This simpler version can optimally use processors in the range 1 through $n/\log^* n$.

Although not a requirement for a priority-queue data structure, we would like to develop a scheme to support an efficient deletion of arbitrary items from the parallel heap, not necessarily the highest priority items. Such an operation is required, for example, for preempting conditional events from a parallel heap when used as a parallel event queue [Fujimoto 1990]. One approach based on garbage collection is as follows: Delete arbitrary items and keep a count of the holes thus created. After $O(n)$ holes have been created, weed out all the holes by packing, and then reinitialize the parallel heap. Even if the reinitialization is done by sorting the entire $O(n)$ items, thus employing $O(n \log n/p)$ time, asymptotic optimality is maintained because $O(n)$ holes can be created only after n/p think-insert-delete cycles of the parallel heap, where each cycle requires $O(\log n)$ time.

We are investigating the usefulness of the parallel heap for applications such as branch-and-bound and heuristic search, multiprocessor scheduling, and shortest path algorithms. Parallel heaps have also been employed for discrete-event simulation [Prasad 1991; Prasad and Deo 1991]. A thorough experimental study of parallel heaps is being conducted on a BBN Butterfly GP-1000.

Acknowledgments

We thank Muralidhar Medidi for discussions on arbitrary deletions, and Robert Brigham and Amit Jain for reading the manuscript and suggesting improvements in presentation. We also thank the anonymous referees for a thorough job in reviewing this paper and Mary Ann Grandjean for making suggestions for improving the presentation of the manuscript.

References

Akl, S.G. 1984. An optimal algorithm for parallel selection. *Inf. Proc. Letters*, 19, 1 (July):47–50.

Bilardi, G., and Nicolau, A. 1989. Adaptive bitonic sorting: An optimal algorithm for shared-memory machines. *SIAM J. Computing*, 18, 2 (Apr.):216–228.

Biswas, J., and Browne, J.C. 1987. Simultaneous update of priority structures. In *Proc., 1987 Internat. Conf. on Parallel Processing* (Aug.), IEEE Comp. Soc. Press, Silver Spring, Md., pp. 124–131.

Cole, R. 1986. An optimal selection algorithm. Ultracomputer Note #97, Courant Institute of Math. Sci., New York Univ., N.Y. (Mar.).

Cole, R. 1988. Parallel merge sort. *SIAM J. Computing*, 17, 4 (Aug.):770–785.

Das, S.K., and Horng, W.-B. 1991. Managing a parallel heap efficiently. In *Proc., Parallel Lang. and Arch. Europe, Lecture Notes in Computer Science*, vol. 505, Springer-Verlag, Berlin, pp. 270–287.

Deo, N., and Prasad, S. 1990. Parallel heap. In *Proc., 1990 Internat. Conf. on Parallel Processing*, vol. 3 (Aug.), IEEE Comp. Soc. Press, Silver Spring, Md., pp. 169–172.

Ellis, C.S. 1980. Concurrent search and insertion in AVL trees. *IEEE Trans. Comps.*, C-29, 9 (Sept.):811–817.

Fan, Z., and Cheng, K.H. 1989. A simultaneous access priority queue. In *Proc.*, 1989 Internat. Conf. on Parallel Processing, vol. 1 (Aug.), IEEE Comp. Soc. Press, Silver Spring, Md., pp. 95–98.

Fujimoto, R.M. 1990. Parallel discrete event simulation. *CACM*, 33 (Oct.):31–53.

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

Hagerup, T., and Rub, C. 1989. Optimal merging and sorting on the EREW PRAM. *Inf. Process. Letters*, 33 (Dec.):181–185.

Jones, D.W. 1989. Concurrent operations on priority queues. *CACM*, 32, 1 (Jan.):132–137.

Karp, R.M., and Ramachandran, V. 1990. Parallel algorithms for shared-memory machines. *Handbook on Theoretical Computer Science*, vol. A: *Algorithms and Complexity* (J. van Leeuwen, ed.), MIT Press, pp. 869–941.

Leiserson, C.E. 1981. Area-efficient computation. Ph.D. diss., Carnegie-Mellon Univ., Penn.

Manber, U. 1984. Concurrent maintenance of binary search trees. *IEEE Trans. Software Engineering*, SE-10, 6 (Nov.):777–784.

Mohan, J. 1983. Experience with two parallel programs solving the traveling salesman problem. In *Proc., 1983 Internat. Conf. on Parallel Processing* (Aug.), IEEE Comp. Soc. Press, Silver Spring, Md., pp. 191–193.

Paul, W., Vishkin, U., and Wagner, H. 1983. Parallel dictionaries on 2-3 trees. In *Proc., ICALP, Lecture Notes in Computer Science*, vol. 154, Springer-Verlag, Berlin, pp. 597–609.

Pinotti, M.C., and Pucci, G. 1991. Parallel priority queues. TR 91-016, ICSI, Berkeley (Mar.) (to appear in *Inf. Proc. Letters*).

Prasad, S. 1990. Efficient parallel algorithms and data structures for discrete-event simulation. Ph.D. diss., Comp. Sci. Dept., Univ. Central Fl., Orlando (Dec.).

Prasad, S. 1991. A scalable and efficient optimistic algorithm for parallel discrete-event simulation. In *Proc., Simulation Technology (SIMTEC)* (Orlando, Oct. 21–23), The Soc. for Comp. Simulation, pp. 350–355.

Prasad, S., and Deo, N. 1991. An efficient and scalable parallel algorithm for discrete-event simulation. In *Proc., Winter Simulation Conf.* (Phoenix, Ariz., Dec. 8–11), The Soc. for Comp. Simulation, pp. 652–658.

Prasad, S., and Deo, N. 1992. Parallel heap: Improved and simplified. In *Proc., Internat. Parallel Processing Symp.* (Beverly Hills, California, Mar. 23–26), IEEE Comp. Soc. Press, Los Alamitos, California, pp. 448–451.

Quinn, M.J., and Deo, N. 1984. Parallel graph algorithms. *Computing Surveys*, 16, 3 (Sept.):319–348.

Quinn, M.J., and Yoo, Y.B. 1984. Data structure for the efficient solution of graph theoretic problems on tightly-coupled MIMD computers. In *Proc., 1984 Internat. Conf. on Parallel Processing* (Aug.), IEEE Comp. Soc. Press, Silver Spring, Md., pp. 431–438.

Rao, V.N., and Kumar, V. 1988. Concurrent access of priority queues. *IEEE Trans. Comps.*, 37, 12 (Dec.):1657–1665.

Rao, V.N., Kumar, V., and Ramesh, K. 1987. Parallel heuristic search on a shared memory multiprocessor. Tech. rept. AI TR87-45, Univ. of Tex. at Austin, Tex. (Jan.).

Wegner, L.M., and Teuhola, J.I. 1989. The external heapsort. *IEEE Trans. Software Engineering*, 15, 7 (July):917–925.