Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors

Sushil K. Prasad* Sagar I. Sawant†

Mathematics and Computer Science
Georgia State University
Atlanta, GA 30303
sprasad@cs.gsu.edu

Abstract

We present an efficient implementation of the parallel heap data structure on a bus-based Silicon Graphics multiprocessor GTX/4D. Parallel heap is theoretically the first heap-based data structure to have implemented an optimally scalable parallel priority queue on an exclusive-read exclusive-write parallel random access machine. We compared it with Rao-and-Kumar's concurrent heap and with the conventional serial heap accessed via a lock. The parallel heap outperformed others for fine-to-medium grains achieving speedups of two to four using six processors relative to the best sequential execution times. The concurrent heap, however, exhibited performance comparable only to the serial heap. As expected for coarser grain, the serial heap performed at par with or better than others.

1 Introduction

A priority queue – an abstract data structure which allows deletion of the highest-priority item and insertion of new items – is one of the basic data structures for many important non-numeric computations such as discrete event simulation [5], and branch-and-bound algorithms [8]. Implementing an efficient parallel priority queue, therefore, is at the core of the effort in parallelizing these problems. In this paper, we describe implementation of a fast parallel priority queue targeted specifically to fine-to-medium grained applications on a small, bus-based, shared-memory multiprocessor. The shared priority queue is not a serial bottleneck for coarse-grained applications – a serial heap accessed via a lock suffices.

We have previously developed a data structure, called parallel heap, which is the first heap-based data structure to have efficiently implemented a scalable parallel priority queue on an exclusive-read and exclusive-write PRAM [4]. Employing p processors, a parallel heap allows deletions of $\Theta(p)$ highest-priority items and insertion of $\Theta(p)$ new items each in $O(\log n)$ time, where n is the size of the parallel heap. The number of processors, p, is optimally scalable up to n. A simplified and practical version of parallel heap has been reported in [1] which forms the theoretical basis for much of the current work. However, significant practical improvements were needed before the parallel heap could be implemented efficiently on a commercial parallel computer.

The main problem in employing a shared priority queue is the penalty for its high access time, which usually involves several locks and barriers for synchronization and mutual exclusion. Therefore, while it is acceptable to perform just one or two operations per queue access for coarse-grained applications, one must perform several operations per access for fine- and medium-grained applications in order to spread out

*This research was partially supported by GSU's RIA grant #96-048.

†Currently with American Mangement Systems, Fairfax, VA.

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE

the queue access overhead over several delete-mins and inserts (the *grain-packing* principle [3]). Accordingly, the following *delete-think-insert* scenario emerges: repeatedly, a processor deletes several top priority items, processes them all one by one, and then, inserts all the newly produced items.

The conventional data structures, however, do not have any mechanism to support multiple delete-mins and inserts per processor using a single queue access. Parallel heap, on the other hand, is built on this concept of grain-packing. Each node of a parallel heap can accommodate up to r items, for any chosen $r \geq 1$, and its root node always has r top priority items available. It allows synchronized deletions of these r or fewer items from the root node, and insertions of up to $2r$ new items in each access cycle (integer 2 can be substituted with any larger integer; 2 suffice for the purpose of this presentation). The parameter r represents the inherent *concurrency factor*, and thus determines the size of the grain-pack.

Rao-and-Kumar's concurrent heap [7] is extracted from the serial heap, and the parallel heap has evolved from concurrent heap, to a certain extent. Also, all three heaps use the array data structure with its implicit binary tree representation without using any pointers. Thus, they constitute an appropriate collection to be experimented with in search for a fast and practical priority queue.

We implemented both synchronous and asynchronous versions of concurrent heap and that of serial heap. In the synchronous version, processors periodically synchronize after executing a total of r items. In asynchronous version, processors continually execute delete-think-insert cycles and terminate after completing a predetermined cycle count. To vary granularity, we modeled the thinking phase by executing through an empty "for loop" for each item processed. The number of iterations is randomly determined up to an upper bound of $maxThink$.

Results: For fine-to-medium grains on Silicon Graphics GTX/4D (*SGI*), the parallel heap yields absolute speedups of two to four using six processors with increasing granularity ($maxThink$ varying from 1 to 4000) on a heap of size $n = 1024K$ items while processing $r = 8K$ items in each delete-think-insert cycle (a concurrency factor of less that 1%). These speedups are significantly more than those obtained from the other two data structures. Almost similar speedups are maintained for smaller concurrency factors (for $r \geq 128$). Even for smaller priority queues, the parallel heap is either better than or at par with the others. For very small n, or for coarser grain, the serial heap outperforms others. The concurrent heap did not show any superiority over serial heap under these test conditions.

The paper's organization is as follows: Section 2 briefly describes each of the three data structures and their operations. Section 3 describes the implementation details of various heaps and their performance tune-ups. Section 4 contains experimental configurations and the timing plots comparing the three data structures. Section 5 contains some concluding remarks.

2 Data Structures and Their Operational Details

2.1 Serial Heap

A serial-heap of a maximum size n can be represented by an array $HEAP[0..(n-1)]$, where nodes $2i+1$ and $2i+2$ are, respectively, the left and the right children of node i. For heap to satisfy the (min) *heap property*, an item at a node should have a value no greater than those of the items at either of the children. A heap supports two operations – *delete-min* and *insert*. A delete-min entails deleting the item from the root. Since this action destroys the heap property at the root, a deletion is followed by a delete-update process, which is transferring the item from the last non-empty node (at the bottom of the heap) to the root, and "sinking" that item down level by level un-

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE

til the heap property is satisfied. This item will be referred to as the "substitute item." An insert operation is traditionally implemented by placing the newly inserted item at the first empty node at the bottom of the heap, and letting it float up toward the root until the heap property is satisfied.

2.2 Concurrent Heap

A concurrent heap is simply a conventional heap structure with some flags associated with each node to support concurrent deletions and insertions. Insertion proceeds from root down to the target node along its unique "insertion path." The insert item keeps sinking down until the heap property is satisfied. Deletion naturally flows top-to-bottom. After deletion, a substitute item is brought in at the root from the last node, and then this substitute is sunk to satisfy the heap property. This substitute needed from the last node may not be immediately available because it could be enroute to the last node while being inserted. In that case, the delete process sets a 'wanted' flag at the last node. The insert process keeps checking the status of its target node and directly places its insert items at the root as the substitute item, once its item becomes 'wanted.'

Since inserts are immediately followed by a delete, both are combined and processed together: If the insert item is smaller than the item at the root, the processor returns with the insert item itself for the subsequent think phase leaving the heap intact. Else, the root item is deleted, and the insert item is used as the substitute item to update the heap. Same strategy can also be employed for serial heap. Readers are referred to [7] for further details.

2.3 Parallel Heap

We provide sketch of the operations of a parallel heap here. Readers are referred to [1] for complete algorithmic details and time complexity analysis. 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*. 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 of node capacity r keeps all the r items at each of its individual nodes sorted. The parallel heap sorts the new items to be inserted before starting its insert-update process. These new items start at the root node, and 'sink' toward their target node at the bottom of the parallel heap after being repeatedly merged with the items at the intervening nodes, and by carrying down the larger items each time. Likewise, after a deletion of some $k \leq r$ items, k items are brought in at the root as substitute items from the bottom of the parallel heap, and merged with the remaining $(r - k)$ items of the root. Since the parallel heap property would be destroyed at the root because of this, a delete-update process begins at the root. This involves merging the items at the root, and its two children, keeping the smallest r at the root, placing the next smallest r items at the left child if its largest item was bigger than that of the right child, else placing them at the right child. Finally, the largest r items are placed at the other child, and a delete-update process is initiated at that child. This process repeats until the parallel heap property gets satisfied. These insert- and delete-update processes are carried out in a pipelined fashion for overall optimality.

Combined Insertion and Deletion: We will assume that the processing of a single item creates at most two new items. Therefore, up to $2r$ new items would be produced that must be reinserted into the

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE

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. We 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. Then, we delete the smallest r items for the subsequent think phase. These are clearly the smallest r items of all the existing items. The remaining items number at most $2r$. Since the root becomes 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 (in case the insert items span two consecutive nodes). 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.

Pipelined Implementation: 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.

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.

Insert-Delete Phase: The basic insert-delete cycle of a parallel heap, which takes care of insertion of up to $2r$ new items and deletion of up to r smallest items, is as follows:

1. Process the update processes at the odd levels of the heap, in parallel, and move them down to the even levels.
2. Sort the newly created items and merge them with the items at the root. Delete the smallest r for the subsequent think phase.
3. Get substitute items and initiate update processes at the root.
4. Move the update processes at the even levels down to the odd levels.

Therefore, after the execution of steps 1 and 4, the root is left updated for the subsequent cycle.

3 Implementation Details

3.1 Serial and Concurrent Heaps

The 'C' language declaration for the serial heap implemented as an array was "int * SerialHeap" along with a single lock for the entire structure. An insertion followed by a delete-min was combined into just one operation as described for the concurrent heap in the Subsection 2.2.

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE

The concurrent heap was also implemented as a linear array with the following declaration:

```
typedef enum {present, pending, wanted,
              absent}       statuscode;
typedef struct { int        key;
                 statuscode status;
               } node;
node * ConHeap;
```

Thus, each node has an additional field, one of the causes for the inefficiency of ConHeap. Another cause is a lock for each level. The implementation of concurrent heap followed the detailed pseudo-code given by [7].

3.2 Implementation of Parallel Heap

Although a parallel heap is simply a linear array of its nodes (which themselves are linear arrays), it was found helpful to implement a parallel heap as a multi-dimensional dynamic structure: int *** ParHeap; Thus, ParHeap[level][node][item] access pattern could be employed. In addition, we have delete and insert flags for each level.

Our first implementation used the algorithm from Subsection 2.3 and employed three barrier synchronizations (no locks are used): one after Step 1, second after Step 3, and the third after Step 4. The think phase was placed at several places: before and after Step 4, and between Step 1 and 3. All the three placements for think phase were found to have similar results. Updating the levels of the parallel heap (in Step 1 and 4) was done concurrently by assigning a processor to update roughly $\frac{l/2}{p}$ levels one by one, where l is the number of levels. This is because the update processes are present either at odd levels or at even levels, not both during any step. Each level was updated sequentially by first processing the delete-update process followed by up to two insert-update processes. Step 2 and 3, sorting newly-produced items, getting substitutes, and initiating update processes at the root, was done by processor P_0 alone.

This version was competitive with serial heap only for coarse grain. For finer grain, although it had a self-relative speedup of two (for $maxThink = 1$, $n = 2^{19}$, and $r = 32$), its actual parallel execution time was no better than one-processor time of SerialHeap.

3.3 Performance Tuning

The parallel heap uses sequential merging in each of its delete- and insert-update processes. Earlier, our delete-update process was implemented as merging the parent subarray z with one of the child subarray x, and getting the sorted output in a temporary array T' of size $2r$. This was followed by merging T' with the second child y of z getting the $3r$ sorted items into another temporary array T. Next, the smallest r items from T were copied into z, the next smallest r into either x or y whichever had the larger last item, and the remaining r into the other child. Thus, the total number of comparisons in the worst-case was $5r$, and the number of moves was $8r$. The following change speeded up this process. Assuming that y had the larger last item, we merged x and y (scanning right to left) outputting the largest r directly into y and the smaller r into T'. Next, we merged z with T' scanning left to right directly placing the smallest r into z and the largest r into x. Note that now both z and y satisfy the parallel heap property. Thus, the total number of comparisons became $4r$, a saving of 20%. Likewise, the number of moves came down by 50%. Similar improvement was done for insert-update process as well.

Next, we moved the sorting task of new items from Step 2, and implemented a simple parallel sort after the think phase. After each process thinks and produces new items, it sorts them individually. Then, those individual lists are repeatedly merged together along a virtual binary tree (synchronizing at each level using a barrier). These enhancement worked well yielding reasonable absolute speedups. It allowed us to efficiently employ large concurrency factors r for better grain-packing.

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE

The next crucial improvement was directed by the profiling data which showed that the barrier synchronization gets much costlier as the number of processors synchronizing increases. Furthermore, for large values of r, there are only about $\log_2(n/r)$ levels and only half of those are active during Step 1 or 4. Thus, not all the processors could be fruitfully employed. This led us to employ only $M < p$ processors for maintenance jobs (Step 1, 2, 3, and 4), and the rest $(p - M)$ processors for general processing (thinking, producing new items, and sorting them in parallel). ParHeap falls back to using all the processors for maintenance and general tasks for $M = p$. The performance improved further to a reasonable level, our current status.

We have found that $M = 1$, 2, and p are usually the best values for fine grain ($maxThink \leq 1000$), and $M = p$ is the best for larger grains. ParHeap yields a self-relative speedup of 2.5 to 5 using 6 processors for $maxThink = 1$ through 4000. This translates into an absolute speedup of 2 to 4 with respect to SerialHeap's one-processor time (the fastest sequential time).

4 Experiments

Experimental Platform: All the experiments have been conducted on an 8-processor Silicon Graphics 4D/280GTX (SGI). This is a bus-based shared-memory computer with an unix-like environment and library function calls to support forks, barriers, and locks, similar to Sequent's parallel machines. The programming was done in 'C' language.

4.1 Experimental Configurations

The three data structures have been exhaustively tested in search for their best operational range. All "C" code was compiled with the highest level of optimization (cc with "-O3" flag). The data structure for a node was simply an integer for all the heaps (except ConHeap which needed the "Status" field as well). We did experiment with larger node structures, and found that the parallel heap performed even better. All data reported represent an average of ten runs. We used super-user mode on SGI to restrict the processors to exclusively run only our experiments (using "sudo restrict p" and "sysmp(MP_MUSTRUN,p)"). However, for a lightly-loaded SGI, use of such measures improved the performance only marginally.

We tested the heaps of sizes $n = 2^7$ through 2^{20} in powers of 2, and varied the number of processors p from 1 through 6. SGI has eight processors; however, two are almost always needed for system and public tasks. For $n = 2^{20}$, the concurrency factor r was varied from 2^3 through 2^{17} in powers of 2. For other n values, r was varied as 0.5%, 1%, 2%, 4%, 8%, and 16% of n. The granularity was varied by choosing $maxThink$ as 1, 10, 100, 200, 500, 1000, 2000, 4000, 10000, and 20000. The number of maintenance processors for ParHeap was varied as $1, 2, \cdots, p$.

To compare all these data structures, we made each perform 2^{21} delete-mins, inserts, and think phases. Such a large enough number resulted in each run of at least a minute, thus predicting the performance over a substantial run. The asynchronous data structures (SerialHeapAsync and ConHeapAsync) ran for $c = 2^{21}/p$ delete-think-insert cycles, each processors performing one delete-min, insert, and think per cycle. For synchronous versions (ParHeap and SerialHeapSync), the number of delete-think-insert cycle was kept at $c = 2^{21}/r$. In each cycle, a processor performed $\frac{r}{p}$ deletes, inserts, and thinks, before synchronizing with others at a barrier.

We tried both the hold model [2], wherein each delete-min results in exactly one insert, and a *relaxed hold model*, which randomly generated zero, one, or two items for each item deleted. The later tests the variability in queue-access pattern as well. The timings for the two were quite similar, and hence the relaxed model timings have not been reported.

The portion of the code timed does not include initial memory allocation, barrier allocation, or the creation of the priority queues with n items (which were randomly generated using rand()%50; other choices of integers had little effects). After the initial con-

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE

struction, the timer was started, p parallel processes were forked out to perform c delete-think-insert cycles, the processes were killed, and then timer was stopped. The timing reported is the real time elapsed. Other detailed timings including system and user times, etc. obtainable from function "times()" were observed but not found to be particularly useful.

An exponentially-distributed priority-increments were generated by the following code:

```
int exp_inc() {return(log((double)random()));}
```

The code used for thinking to introduce random granularity with an upper bound of $maxThink$ for each item was:

```
void think()
  {   int i,j;
      j = random() %  maxThink;
      for (i=1; i<j; i++);
  }
```

Note, therefore, that a plot corresponding to a think time of 1000 actually corresponds to an average of 500 iterations.

4.2 Results

For brevity, we present only a selected set of plots chosen to give a clear idea of the performance of the three data structures, and their limitations. [6] contains more timing figures and additional details.

Figure 1 presents the absolute speedup obtained for ParHeap, ConHeapSync, SerialHeapAsync, and SerialHeapSync as the maximum think time is varied from 1 through 4000 (higher granularity is uninteresting as serial heap performs best). The queue size was $n = 2^{20}$, $r = 8092$, and the maximum number of processors employed was six. ConHeapSync is not reported because it is worse than its asynchronous counterpart. Even for zero granularity, ParHeap has a speedup of over two. And, it quickly reaches three for $maxThink = 1000$. ConHeapAsync is only marginally better than the SerialHeaps for finer grain, and then gets worse for larger

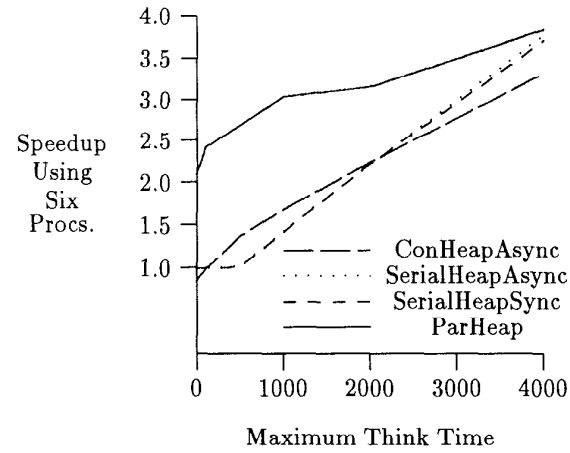

Figure 1: Speedup as a function of granularity ($n = 1024K$, $r = 8K$, $p \leq 6$)

grain. More importantly, for $maxThink \leq 1000$, both ConHeapAsync and SerialHeaps achieve speedups well within two. Even for $maxThink = 4000$, ParHeap continues to dominate. Thus, Figure 1 establishes the practicality of ParHeap for fine-to-medium grained applications which require large priority queues.

Figure 2 plots the execution times for varying r. Thus, even a concurrency factor of 128 allows good speedups.

To see the behavior of these data structures for varying heap sizes, Figure 3 plots the absolute speedups for varying sizes of heaps for a medium-grain of $maxThink = 1000$ and $r = n/32$ (a concurrency factor of about 3%). While the speedups obtainable from serial and concurrent heaps steadily decline as the heap size grows, the speedup increases for the parallel heap. Data for SerialHeapSync and SerialHeapAsync are very similar. Therefore, the ones for the former are not plotted. Thus, the operational range of the current version of the parallel heap is heap size $n \geq 1K$, grain size $maxThink \leq 4000$, and concurrency factor $r \geq 128$.

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE

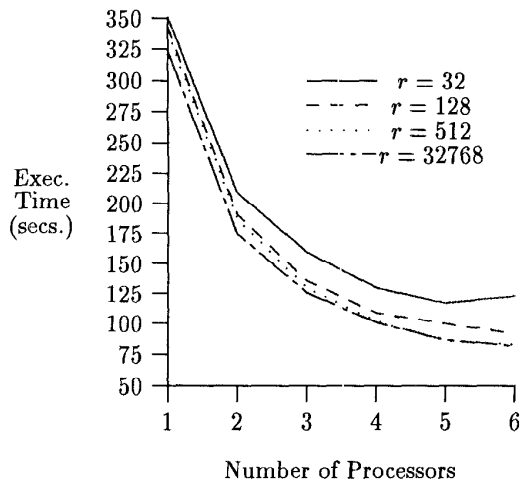

Figure 2: Effect of concurrency factor r ($maxThink$ = 2000, $n = 2^{20}$).

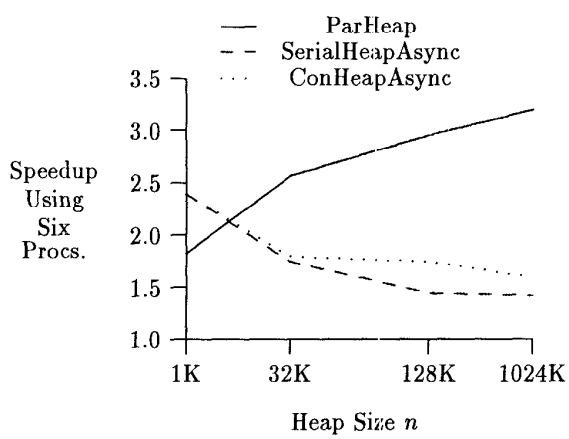

Figure 3: Speedup as a function of heap size ($maxThink$ = 1000 and $r = n/32$).

5 Conclusions

Parallel heap has emerged as a practical priority queue data structure for fine-to-medium-grained large-sized applications on small multiprocessors achieving speedups from two to four using six processors. The serial heap suffices for coarser grains ($maxThink >$ 4000) and for very small queues ($n \leq 1024$). Rao-and-Kumar's concurrent heap, however, only achieved speedups comparable to those of serial heaps. The work on additional improvements in parallel heap continues, as well as porting it to other machines. Other parallel sorting and merging algorithms are being evaluated for our environment, and they are expected to bring forth further speedups.

References

[1] Deo, N., and Prasad, S. 1992. Parallel heap: An optimal parallel priority queue. *J. Supercomputing,* 6: 87-98.

[2] Jones, D. W.. 1986. An empirical comparison of priority-queue and event-set implementations. *Comm. ACM,* 29, 4 (April), 300-311.

[3] Kruatrachue, B. and T. Lewis. 1988. Grain size determination for parallel processing. *IEEE Software,* 5, 1 (Jan), 23-31.

[4] Prasad, S. K. 1990. *Efficient parallel algorithms and data structures for discrete-event simulation.* Ph.D. Diss., Computer Science Dept., Univ. Central Florida, Orlando, (Dec.).

[5] Prasad, S. K. 1993. Efficient and Scalable PRAM algorithms for discrete-event simulation of bounded degree networks. *J. Parallel and Distributed Computing,* 18, 524-530.

[6] Prasad, S. K. and Sawant, S. 1994. Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors. Tech. Rep. Mar-15-95-3, Dept. of Math. and Computer Science, Georgia State University, 20 pages.

[7] Rao, V. N., and Kumar, V. 1988. Concurrent access of priority queues. *IEEE Trans. Comput.,* 37, 12 (Dec.), 1657-1665.

[8] Rao, V. N., Kumar, V., and Ramesh, K. 1987. Parallel heuristic search on a shared-memory multiprocessor. Tech. Rep. AITR87-45, Univ. of Texas at Austin.

Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing (SPDP '95)
1063-6374/95 $10.00 © 1995 IEEE