Concurrent Access of Priority Queues*

V. Nageshwara Rao and Vipin Kumar†

Department of Computer Sciences,

University of Texas at Austin,

Austin, Texas 78712

Abstract

The heap is an important data structure used as a priority queue in a wide variety of parallel algorithms (e.g., multiprocessor scheduling, branch-and-bound). In these algorithms, contention for the shared heap limits the obtainable speedup. This paper presents an approach to allow concurrent insertions and deletions on the heap in a shared-memory multiprocessor. The scheme also retains the strict priority ordering of the serial-access heap algorithms; i.e., a delete operation returns the best key of all keys that have been inserted or are being inserted at the time delete is started. Our experimental results on the BBN Butterfly parallel processor demonstrate that the use of the concurrent-heap algorithms in parallel branch-and-bound improves its performance substantially.

Index Terms: concurrent data structures, priority queues, insertions, deletions, branch-and-bound, speedup.

1 Introduction

The heap is an important data structure used as a priority queue in a wide variety of parallel algorithms (e.g., multiprocessor scheduling, graph search[15], branch-and-bound[18,

*This work was supported by Army Research Office grant # DAAG29-84-K-0060 to the Artificial Intelligence Laboratory, and Office of Naval Research Grant N00014-86-K-0763 to the computer science department at the University of Texas at Austin.

†Arpanet: kumar@cs.utexas.edu

12, 8, 16]) on shared-memory multiprocessors. In these algorithms each processor repeatedly performs an access-think cycle. Every processor executes its current subproblem at hand (thinking), then accesses the shared heap to insert subproblems if it generated any and removes the best available subproblem in the heap to solve next. Since many processors share the heap, the simplest way to provide consistency in updates is to serialize the updates. A lock is associated with the heap and the processors access the heap under mutual exclusion. This serial-access scheme limits the number of processors that can be used to speed up the problem. If T_{think} is the mean think time and T_{access} is the mean access time, then clearly the maximum speedup achievable is $\leq (T_{access} + T_{think})/T_{access}$ (see [8]). T_{think} is a characteristic of the problem being solved. T_{access} depends on the priority structure being used. For the heap, T_{access} is $O(\log M)$, where M is the size of the heap.

One way to alleviate the limitation is to let many processors access the heap simultaneously. Updates on different parts of a heap can proceed concurrently provided they do not interact with each other. Let us view the heap as a binary tree with the root at the top and leaves at the bottom. In the ordinary serial heap algorithms, deletes manipulate the heap level by level going from top to bottom, while inserts manipulate it from bottom to top. Hence many deletions (or many insertions) can be executed in parallel by using a simple window locking scheme[3] or software pipelining[13, 14]. But inserts and deletes cannot be active together, as they proceed in opposite directions and hence can deadlock. Biswas and Browne[3] present a scheme to handle this problem. But their scheme incurs substantial overhead, and performs worse than the serial-access heap unless the heap size M is very large.

This paper presents a new concurrent-heap access scheme that has small overhead, and is able to perform better than the serial-access heap even for small heaps. Two important ingredients of this scheme are (i) a heap insertion algorithm which manipulates the heap from top to bottom; and (ii) a scheme to combine a delete operation with the most recent unfinished insertion operation. Since these new insertions and the deletions move from top to bottom in the heap, they can both be active together without causing deadlocks.

Section 2 reviews conventional insert and delete operations on the heap, and presents a new insert operation that traverses the heap from top to bottom. Section 3 presents concurrent-heap algorithms developed using this insert operation, and provides a proof of its correctness. Section 4 analyzes the expected performance improvement due to the concurrent-heap scheme. Section 5 presents experimental results on the BBN Butterfly parallel processor evaluating the improvement in performance due to the concurrent-heap algorithms . Section 6 discusses ways to improve the basic scheme to further reduce the overheads. A comparison with related work is presented in Section 7. Section 8 contains concluding remarks.

2 Serial Access Heap algorithms

2.1 Preliminaries

A heap [1, 7] is a complete binary tree of depth d, with the property that the value of the key at any node is less than the value of the keys at its children (if they exist).[1] Before presenting the concurrent update scheme, we briefly describe the sequential implementation of the heap to establish the terminology. Throughout the paper we present algorithms in a machine-independent, high-level pseudo-code.

It is efficient to implement the heap using an array. The root occupies location 1 and the node i occupies location i. The children of node i occupy locations 2i and 2i+1. The parent of node i is at $\lfloor \frac{i}{2} \rfloor$. We assume that each node in the heap has a key pointing to a field of data. Key(i) denotes the key located at node i. VALUE(i) denotes the value or the priority order of the key at node i. Empty nodes in the heap are assumed to have a special key called MAXINT whose value is ∞.

We denote the left son and the right son of node i by LSON(i) and RSON(i) respectively. The parent of node i is denoted by PARENT(i). Associated with the heap are the data fields lastelem and fulllevel[2]. lastelem is the index of the last non-empty node of the heap. fulllevel is the index of the first node in the deepest level of the heap that contains at least one non-empty node. For an empty heap, lastelem = fulllevel = 0. Fig. 1 shows a sample heap of twelve keys, and the value of lastelem and fulllevel.

**** Figure 1 comes here *****

[1] The discussion in this paper is applicable even if the heap is represented as a k-ary tree.

[2] The conventional delete and insert_b operations described below do not need to maintain fulllevel, but it is needed for the insert_t operation which traverses the heap from top to bottom.

2.2 Insertion and Deletion operations on a Serial Heap

The operations supported on a heap are insertion and deletion. The insert operation inserts a new key, nkey, in the heap and the delete operation returns the smallest key in the heap.

The insert operation grows the heap by adding a key to the first empty node in the heap. Let us call this location **target**. When target has this new key, then the heap property may be violated at the nodes on the path from target to the root. The following insertion algorithm performs reheapification on this path by pushing the new key upwards. This type of insertion is called **insert_b** because it proceeds from the bottom of the heap towards the top.

```
insert_b(nkey, heap)
Lock(heap);
lastelem ←lastelem + 1 ;
if (lastelem ≥ fulllevel⋆2) then fulllevel ←lastelem endif
i ←lastelem ;
key[i] ←nkey ;

/* Reheapification Loop */
while ((i ≠ 1) ⋁ (VALUE(i) < VALUE(PARENT(i)))) do
        Exchange(key(i),key(PARENT(i)) );
        i←PARENT(i);
endwhile
Unlock(heap) ;
end_insert_b
```

The delete operation shrinks the heap by removing the key at the root of the heap and by placing the key of the last non-empty node of the heap at the root. The heap property may now be violated at the root of the heap. Reheapification is performed by pushing this key downward until the heap property is satisfied at the node where this key is held. Note that, after placing the key of the last node at the root, the heap structure is changed only internally. It does not shrink or grow.

```
delete(heap)
Lock(heap);
if (lastelem = 0) then {Unlock(heap); Return(NULL)} endif
least ←key(1) ;
i ←1 ;
j ←lastelem ;
lastelem ←lastelem - 1 ;
if (lastelem < fulllevel) then fulllevel ←fulllevel/ 2 endif
if (j = 1) then {key(1) ←MAXINT; Unlock(heap); Return(least)} endif
key(1) ←key(j) ;
key(j) ←MAXINT ;

/* Reheapification Loop */
```

```
/* Let MIN(i) be the index of the son of i which has smaller VALUE*/
while (VALUE(i) > VALUE(MIN(i))) do
        Exchange(key(i),key(MIN(i))) ;
        i ←MIN(i) ;
endwhile
Unlock(heap) ;
Return(least) ;
end_delete
```

2.3 Inserting from Top

It is possible to perform insertions from the top by using the following (informally stated) algorithm:

```
k ←1;
if VALUE(k) > VALUE(nkey) then Exchange(key(k),nkey)) ;
while (k has both successors)
        k ←any successor of k;
        if VALUE(k) > VALUE(nkey) then Exchange(key(k),nkey)) endif
endwhile
Put nkey at one of the empty leaves of k.
```

This naive insertion algorithm is not guaranteed to grow the heap level-by-level, which is crucial for the efficiency of insertions and deletions.[3] Our new insertion algorithm, which we call insert_t, performs reheapification in such a way that each insertion adds a key to the first empty node in the heap (just as in the conventional insert_b operation).

Recall that target is the first empty node in the heap. is **insertion path** is the path between the root and target. This path is unique because the heap has a tree structure. This path can be easily traversed starting from target (integer division by 2 gives the parent of any node in the binary tree). This path can also be traversed starting at the root as follows. Let I be the displacement of target at the last level (i.e., I = lastelem - fulllevel), and p be the length of the insertion path. If we view I as a p bit binary number, then the bits of the binary representation of I (from the most significant to the least significant) tell us whether to go right (if 1) or left (if 0) when we go from the root downward. For example, the first

[3] If the heap becomes unbalanced, then inserts and deletes can take up to $O(M)$ operations rather than O($\log M$) operations.

node at the last level (given by fulllevel) has displacement 0 and its path is left,left,left$\cdots$. Fig. 2 shows the twelve node heap of Fig. 1 to which a thirteenth node is being added. It also shows the values of fulllevel, lastelem, and I. In binary representation, I = (101), which means that we can go from the root to target by following right, left and right branches at successive nodes.

****** Figure 2 comes here ********

Values of the nodes on the insertion path (from root to target) are nondecreasing. To insert a new key in the heap, we need to put the new key at a proper node on the insertion path, and move all the keys at and below this node one level down (filling the target node). The insert_b algorithm does this by visiting the nodes on the insertion path from bottom to top. The **insert_t** algorithm given below does it in the opposite order.

```
insert_t(nkey,heap)
Lock(heap)
lastelem ←lastelem + 1 ;
target ←lastelem ;
if (lastelem ≥ fulllevel⋆ 2) then fulllevel = lastelem endif
i ←target - fulllevel ; /* i is the displacement of target */
j ←fulllevel/2 ; /* j = 2^(length of insertion path −1) */
k ←1 ; /* k is the current position in the insertion path */
/* Reheapification Loop */
while (j ≠ 0)
        if(VALUE(k) > VALUE(nkey)) then Exchange(nkey,key(k)) endif
        if (i ≥ j)
          then {k ←RSON(k); i ←i - j;} /* Go Right */
          else {k ←LSON(k)}; /* Go Left */
        endif
        j ←j/2 ;
endwhile
key(k) ←nkey ;
Unlock(heap) ;
end_insert_t
```

In the above procedure, the insertion path is being computed on the fly. In the nth iteration of the re-heapification loop of insert_t, nth bit (from the left) of the binary representation of I is tested. If it is 1, then the right son is traversed next; otherwise the left son is traversed next. At the beginning of the nth iteration of insert_t, the value of i is the same as the value of I if its $n-1$ left most bits are set to zero. See [17] for a proof of correctness of the insert_t algorithm.

Status code	*Meaning*
PRESENT	A key exists at the node.
PENDING	An insertion is in progress which will ultimately insert a key at the node
WANTED	A deleter is waiting for the key.
ABSENT	No key is present at the node.

Table 1: Meaning of various status codes.

3 Concurrent-access heap algorithms

A simple locking strategy is embedded into delete and insert_t routines to achieve concurrency in access maintaining consistency in updates and avoiding deadlocks. Instead of locking the whole heap (as done in the serial-access scheme), we lock only a small portion of the heap at a time. This portion is called a **window**. It consists of 3 nodes for the delete algorithm and 1 node for the insert operation. In order to allow window locking, we associate a lock with every node. Each processor accesses the contents of a node only after locking it to ensure mutual exclusion. The two other data fields of the heap, full level and lastelem, are modified only in the initialization phase of the insert_t and delete routines. Hence we conveniently associate the lock of node 1, the root, with these fields also; i.e., a processor can access these locations only when the root has been locked.

Although insert_t and delete both manipulate the heap from top to bottom, there is one problem in letting them work together. Recall that the delete operation deletes the key at the root and replaces it with the most recently inserted leaf key (and starts reheapification). If the most recent insert_t operation is still in progress, then this last leaf node does not have a key. If delete picks up the key of any other leaf node, then the resulting heap may become unbalanced. If the delete operation waits for the last insertion to finish, then we loose concurrency.

To solve this problem, we associate a field called status with every node in the heap. The status of a node can have four values, each associated with the semantics given in Table 1. When an insertion starts, the status of its target is set to PENDING. If a deleter starts working when an insertion is still in progress, it changes the status of the target of the last inserter to WANTED, and waits. At the beginning of each iteration of the reheapification loop, the inserter checks to see if the status of target has become WANTED. If this is the case, then nkey is placed at the root and the inserter quits. Once the key is placed at the root, deleter starts working. The concurrent deletion and insertion algorithms are presented below.

```
Concurrent Delete(heap)
Lock(1);
/* Lock the root of the heap */
if (lastelem = 0) then {Unlock(1); Return(NULL)} endif
least ←key(1) ;
i ←1 ;
j ←lastelem ;
lastelem ←lastelem - 1 ;
if (lastelem < fulllevel) then fulllevel ←fulllevel/2 endif
if (j=1) then{ key(1) ←MAXINT; status(1) ←ABSENT; Unlock(1); Return(least)} endif

Lock(j) ;
if (status(j) = PRESENT)
   then {key(1) ←key(j); status(j) ←ABSENT; key(j) ←MAXINT;}
   else {status(1) ←ABSENT ; status(j) ←WANTED};
endif
Unlock(j) ;
while (status(i) = ABSENT) do {Wait()} endwhile /* i = 1 at this point */

Lock(LSON(i)) ; Lock(RSON(i)) ;
/* Reheapification Loop */
/* Let MIN(i) give index of the son of i which has lower VALUE*/
/* Let MAX(i) give index of the son of i which has higher VALUE*/
while (VALUE(i) > VALUE(MIN(i))) do
        Exchange(key(i),key(MIN(i))) ;
        Unlock(i) ; Unlock(MAX(i)) ;
        i ←MIN(i) ;
        Lock(LSON(i)) ; Lock(RSON(i)) ;
endwhile
Unlock(i) ; Unlock(LSON(i)) ; Unlock(RSON(i)) ;
Return(least) ;
end_Concurrent_Delete
```

```
Concurrent Insert(nkey,heap)
Lock(1) /* Lock root of the heap */
lastelem ←lastelem + 1 ;
target ←lastelem ;
if (lastelem ≥ fulllevel⋆ 2) then fulllevel ←lastelem endif
i ←target - fulllevel ; /* i is the displacement of target */
j ←fulllevel/2 ; /* j = 2^(length of insertion path −1) */
k ←1 ; /* k is the current position in the insertion path */
status(target) ←PENDING ;

/* Reheapification Loop */
while (j ≠ 0)
        if (status(target) = WANTED) then break endif
        if (VALUE(k) > VALUE(nkey)) then Exchange(nkey,k); endif
        if (i ≥ j)
           then /* Go Right */
           {Lock(RSON(k));Unlock(k); k ←RSON(k); i ←i - j}
           else /* Go Left */
           {Lock(LSON(k)); Unlock(k); k ←LSON(k)};
        endif
        j ←j/2 ;
endwhile
```

```
if (status(target) = WANTED)
        then /* Some deleter is waiting at the root to pick the key at target */
        {key(1) ←nkey; status(target) ←ABSENT; status(1) ←PRESENT}
        else
        {key(target) ←nkey; status(target) ←PRESENT};
endif
Unlock(k) ;
end_Concurrent Insert
```

Whenever an inserter or a deleter moves down 1 level by incrementing k or i, it first locks the next node and then releases the current lock. This ensures that concurrent deletes or inserts proceeding in the same path progress in strict queue order without any interference. Since the locking sequence is in the strict increasing order of node indices, there are no deadlocks. See [17] for a proof of correctness of the concurrent-heap scheme.

4 Theoretical Analysis of Performance

In this section we present a discussion of expected improvement in speedup due to the new concurrent-heap algorithms. First we define some terms and state some assumptions that are made to simplify the analysis.

4.1 Definitions and Assumptions

1. M is the total number of keys present in the heap.

2. Think time T_{think} is the time spent by each processor in computation in between successive accesses to the heap.

3. L is the number of levels traversed in the heap during heap access operations. For example, for insert_t,

$$L = \lfloor \log(M+1) \rfloor$$

4. Access time T_{access} is the time required to perform an operation on the heap in the absence of contention for the heap.

$$T_{access} = L * T_w + T_r$$

   Here, T_r is the time for executing the initialization part of the code (i.e., the code before the reheapification loop), and T_w is the time for executing one iteration of the reheapification loop. For simplicity, the above definition of T_{access} assumes that, for any given operation, the execution time of different iterations of the reheapification loop are the same.

Clearly, T_w ,T_r and hence T_{access} are higher for concurrent heap because of the overheads of locking, etc. We use the superscript c to denote that the term refers to the concurrent-access operation and the superscript s to denote that the term refers to the corresponding serial access operation, whenever a distinction needs to be made. For example, T^c_{access} refers to access time using the concurrent-access scheme.

Speedup S is the ratio of time taken by one processor to execute N think-access cycles, and the time taken by P processors to execute the same number of cycles. In the parallel case, each processor performs $\frac{N}{P}$ think-access cycles.

Speedup improvement factor due to concurrent heap

$$= \frac{\text{speedup due to concurrent heap}}{\text{speedup due to serial-access heap}}$$

4.2 Analysis

Here we study the performance of serial and concurrent-heap algorithms for the task of performing N think-access cycles using P processors ($N \gg P$). To simplify the analysis, we assume that all N access operations are identical, and that all processors have the same processing speed.

1. The Base Case

   The time needed by one processor to execute N think-access cycles

   $$= N * (T_{think} + T^s_{access})$$

2. Serial Heap

   Now P processors perform the same number of operations using the serial heap algorithms. Each processor performs $\frac{N}{P}$ think-access cycles, and locks the heap for the duration of each access operation. If there is no contention for the heap, then each processor takes $T_{think} + T^s_{access}$ for executing one cycle. Even if P is very large, the maximum rate at which the operations can be done is $\frac{1}{T^s_{access}}$ per unit time.

$$Time\ for\ N\ operations = \begin{cases} \frac{N}{P}(T_{think} + T^s_{access}) & if\ P \leq \frac{T_{think}+T^s_{access}}{T^s_{access}} \\ \\ N * T^s_{access} & if\ P \geq \frac{T_{think}+T^s_{access}}{T^s_{access}} \end{cases}$$

$$Speedup = \begin{cases} P & if\ P \leq \frac{T_{think}+T^s_{access}}{T^s_{access}} \\ \\ \frac{T_{think}+T^s_{access}}{T^s_{access}} & if\ P \geq \frac{T_{think}+T^s_{access}}{T^s_{access}} \end{cases}$$

The maximum speedup achievable with the serial heap algorithms is $\frac{T_{think}+T^s_{access}}{T^s_{access}}$.

3. Concurrent Heap

   Now P processors perform N think-access cycles using the concurrent-heap algorithms. Note that the next operation can start after the current operation has unlocked the root. The time for which the root is locked $= T^c_w + T^c_r$. Therefore, the maximum rate at which the operations can be performed is $\frac{1}{T^c_w+T^c_r}$ per unit time.

$$Time\ for\ N\ operations = \begin{cases} \frac{N}{P}(T_{think} + T^c_{access}) & if\ P \leq \frac{T_{think}+T^c_{access}}{T^c_w+T^c_r} \\ \\ N(T^c_w + T^c_r) & if\ P \geq \frac{T_{think}+T^c_{access}}{T^c_w+T^c_r} \end{cases}$$

$$Speedup = \begin{cases} P * \frac{T_{think}+T^s_{access}}{T_{think}+T^c_{access}} & if\ P \leq \frac{T_{think}+T^c_{access}}{T^c_w+T^c_r} \\ \\ \frac{T_{think}+T^s_{access}}{T^c_w+T^c_r} & if\ P \geq \frac{T_{think}+T^c_{access}}{T^c_w+T^c_r} \end{cases}$$

   The maximum speedup obtainable with the concurrent-heap algorithms is $\frac{T_{think}+T^s_{access}}{T^c_w+T^c_r}$.

From the above analysis, we conclude the following.

1. The concurrent-heap scheme allows $O(\log M)$ operations to proceed in parallel.

2. Unless T^s_{access} is greater than $T^c_w + T^c_r$, concurrent heap cannot perform better than the serial heap.

3. If T_{think}= 0 (and $T^s_{access} > T^c_w + T^c_r$), then the concurrent heap performs better than the serial heap as long as $P > \frac{T^c_{access}}{T^s_{access}}$. Otherwise (for $P < \frac{T^c_{access}}{T^s_{access}}$), the concurrent heap performs worse than the serial heap due to the overheads.

4. If $T_{think} \gg T_{access}$ (and $T^s_{access} > T^c_w + T^c_r$), then the concurrent heap does not perform worse than the serial heap even for small P. It performs better than the serial heap if $P > T_{think}/T^s_{access}$.

5. For large P ($\geq \frac{T_{think}+T^c_{access}}{T^c_w+T^c_r}$), the speedup improvement factor due to concurrent heap is $\frac{T^s_{access}}{T^c_w+T^c_r}$, which is independent of T_{think}. The actual value of the improvement factor varies for different operations as follows.

- Case I: Deletes

  For the serial delete, $T^s_{access} = T^s_r + L * T^s_w$, where $L \approx \log M$. Thus, T^s_{access} increases logarithmically with the heap size, and $T^c_w + T^c_r$ remains (an implementation dependent) constant. Hence there exists K such that for $M > K, T^s_{access} > T^c_w + T^c_r$. Thus for big enough heaps, the concurrent-heap scheme performs better than the serial-access heap, and the speedup improvement factor grows as $O(\log M)$.

- Case II: Random Inserts

  If the keys of randomly distributed values are inserted into a heap, then the number of iterations of the heapification loop executed in insert_b is very small [7]. Hence, L in $T^s_{access} = T^s_r + L * T^s_w$ is very small (<2) even for very large heaps. Hence, for this case, the concurrent heap does not perform better than the serial heap, as in most implementations, $T^c_w + T^c_r$ is not much smaller than T^s_{access} . [4] Note that if concurrent (random) inserts and deletes are performed simultaneously, then the concurrent heap could still perform better overall if the average access time for serial inserts and deletes is larger than the average root locking time in concurrent heap operations.

- Case III: Biased Inserts

  In many parallel algorithms (e.g., branch-and-bound [18]), each newly inserted key tends to be nearly as good as the best key already available in the heap. In this case, $T^s_{access} = T^s_r + L * T^s_w$, where $L \approx \log M$. For this case, just as in case I, the speedup improvement factor grows as $O(\log M)$.

[4]it may even be larger for some implementations.

In our analysis, we assumed that all access operations take exactly the same amount of time. In practice, T_{access}would be different for different operations. In this case, for larger number of processors, the speedup improvement factor would be $\frac{mean(T^s_{access})}{mean(T^c_w+T^c_r)}$.

5 Experimental Evaluation of the Concurrent-access heap algorithms

We have implemented the concurrent-heap algorithms and the serial-access heap algorithms on the BBN Butterfly multiprocessor to test their performance. Using each scheme, P processors performed a total of 1000 delete or insert operations (each processor performed $1000/P$ operations. P was varied between 1 and 30). The speedup was computed according to the definition given in Section 4.1. Relative performance of the concurrent heap was studied for the following cases.

Case I: Deletes

In this case, each processor performed one delete operation in each access-think cycle. A total of 1000 delete operations were performed on a heap that initially had 2048 keys. Thus, the depth of the heap remained 10 for all the deletions. Fig. 3 shows speedup results for the case in which T_{think} is set to 10 ms ($\approx 5 * T^s_{access}$). For the serial-access scheme, the speedup was fairly linear up to 5 processors, but saturated after that. For the concurrent heap, the speedup saturated at 11.6. For less than 5 processors, concurrent delete performs slightly worse than the serial delete. When we decreased (or increased) T_{think}, the speedup dropped (or went up) for both sequential and concurrent deletes. But, as predicted by the analysis of the previous section, the speedup improvement factor for large number of processors remained roughly the same (≈ 2.3).

***** Figure 3 come here ******

Case II: Inserts

In this case, each processor performed one insert operation in each access-think cycle. A total of 1000 insert operations were performed on a heap that initially had 1024 keys. Thus, the depth of the heap remained 10 for all the insertions. As discussed in the previous section, for inserting keys with random key values, our concurrent-heap scheme does not perform better than the serial-access heap insert. (The speedup figures are roughly the same for both concurrent heap and serial heap; hence they are not shown.)

To test the performance for biased inserts, we generated keys whose values were in the decreasing order. In this case, the relative performance of the concurrent-heap scheme is similar to that obtained for deletes. Fig. 4 shows the speedup curves for T_{think}= 10ms.

****** Figure 4 comes here ****

Case III: Parallel Branch-and-Bound

To test the performance of the concurrent-heap scheme in a more realistic situation, we incorporated it in a parallel branch-and-bound algorithm for solving the traveling salesman problem[18, 12, 8]. In this parallel algorithm, in each access-think cycle, each processor removes a least cost node from the heap, generates two successors, computes their costs, and inserts them both on the heap. The think time is the time to create two successors and compute their costs (for the algorithm we used, it is $O(n^2)$, where n is the number of cities). We implemented two versions of the parallel algorithm on BBN Butterfly - one using the serial-access heap, and the other using the concurrent-access heap. For each version, the speedup was computed with respect to sequential branch-and-bound using the conventional heap algorithm. As shown in Fig. 5, the concurrent-access scheme delivers significantly higher speedups than the serial-access scheme. In the problem instances we used for experiments, the heap size grew up to 8000 elements. The larger heap size explains the larger speedup improvement factor (≈ 3).

***** Figure 5 comes here *****

6 Possible Improvements

A number of modifications can be made to our scheme to further improve its performance. Some of these are outlined below.

1. Combining Insertions and Deletions

   If a processor needs to do one or more inserts and a delete, then it can merge the last insert operation and the delete operation into a slightly modified delete operation as follows. If the root of the heap is worse than the key to be inserted, then the (merged) operation finishes right away, and processor treats the key to be inserted as the one deleted. Otherwise, it replaces the root with the key to be inserted, and proceeds (without having to wait for some other active inserter to give it the key) with the deletion routine. This improvement can be incorporated in the serial-access scheme as well.

2. FIFO access at the bottom of the heap.

   Notice that insertions and deletions extend the current last level of the heap in a LIFO manner; i.e., a delete operation removes the most recently inserted key. We can change this to FIFO; i.e., as long as the current level is not completely empty, the deleter deletes the oldest inserted key from this level. Now inserters put nodes at one end of the last level while deleters take nodes form the other end. (Clearly, this implicit FIFO queue is to be implemented at each level with a wrap around. We also need extra synchronization when a level is completely filled up or completely empty.) This scheme makes it unnecessary for the deleter to wait for the inserter to put its key at the root, and thus removes one checking operation from the loop of insert (reducing T_w^c) and reduces the initial processing time (T_r^c) in delete.

3. Level Locking

   This modification is useful if processors execute at uniform speed and the locking operation is expensive. We can associate one lock with each level of the heap, and instead of locking individual nodes, we can lock the entire level as needed. This reduces the number of lock and unlock operations in each cycle of concurrent delete from four to two (reducing T_w^c). The number of lock and unlock operations in concurrent insert remain unchanged.

4. Service Processors

   It is possible to incorporate service processors in our scheme as follows. Whenever a processor needs to insert or delete, it communicates with the next available service processor (the available service processors may be maintained in a queue), which performs the actual operation on behalf of the user processor. For example if the user processor likes to delete a key, then a service processor locks the root, gives the root key to the user processor, and continues with the delete operation. The user processor can continue with its processing without having to wait for the entire delete operation to finish. If $O(\log M)$ processors are available, then one processors (or a small number of processors) can perform delete or insert in constant time irrespective of the size of the heap. This arrangement may be useful if T_{think} and the number of user processors are small. If T_{think} is large, then service processors do not provide any performance improvement.

5. Splitting T_r^c

   One way to reduce the bottleneck at the root (given by $T_w^c + T_r^c$) is to split the root update operation into two steps - the first step updating the values of lastelem, etc., and the second part performing the reheapification at the root. We can associate two different locks for these two steps.

7 Related Research

In [3], Biswas and Browne present a scheme, called CHEAP, that allows insertions and deletions to proceed in parallel. In their scheme, an insert or delete operation is decomposed into a sequence of update steps at different levels of a heap. An auxiliary task queue stores the steps of insertions and deletions currently in progress. By appropriately scheduling these update steps, a set of service processes concurrently perform insertions and deletions without causing deadlocks. If enough service processors are available, then this scheme allows many insertions and deletions to proceed in parallel. This approach is not able to perform better than the serial access scheme except for very large heaps due to the overheads associated with scheduling window updates through the server queue.

Unlike the scheme in [3], our scheme does not require special server processors to update the heap. Also the number of locks needed for each operation are much smaller in our scheme. Unlike their scheme, our scheme also retains the strict priority ordering of the serial-access heap algorithms; i.e., a delete operation returns the best key of all keys that have been inserted or are being inserted at the time the delete operation is started. The scheme presented in this paper was motivated by the work of Biswas and Browne. Initially, we wanted to incorporate CHEAP in our parallel branch-and-bound algorithms to improve their performance. But experiments conducted by Biswas [5] showed that CHEAP was not able to perform better than the serial-access scheme even for heaps with 10,000 keys.

Ellis and Gaffar[6] have developed a scheme that also does not require the use of separate special service processors. In this scheme, inserts and deletes proceed in opposite directions,

[5]Private communication

[6]Private communication with Carla Ellis

but avoid deadlock using a "sliding-lock" scheme. Performance results of this scheme are not yet available.

A number of concurrent-access schemes have been developed for manipulating dictionaries that are represented as balanced trees[6, 11], B-trees [5], and the balanced cube[4]. Most of these concurrent schemes allow $O(\log M)$ operations (delete the smallest key, delete a key, insert a key, search for key, etc.) to be done simultaneously. A major exception is the balanced cube which permits $O(M)$ search, insert and delete operations to done concurrently. However, even the balanced cube permits only $O(\log M)$ operations "delete-the-smallest-key" operations at a time. In a priority queue, the only operations of interest are "delete-the-smallest-key" and "insert-a-key". For these operations, on a sequential processor, the heap is clearly a more efficient data structure than B-tree, balanced trees and the balanced cube. Since our concurrent-access heap scheme has the same degree of concurrency as others and has smaller overhead, it is better than other concurrent schemes for manipulating a strict priority queue.

A number of VLSI dictionary machines [**?**, 19, 2] use $O(M)$ processors and allow $O(\log M)$ operations to proceed in parallel. Leiserson's hardware priority queue[10] provides $O(1)$ access time at the expense of $O(M)$ processors. As discussed in the previous section, with the help of $O(\log M)$ service processors, our concurrent-access scheme can also provide $O(1)$ access time. However, the constant factor in $O(1)$ is expected to be smaller for the hardware priority queue.

8 Conclusions

We have presented a new scheme that allows concurrent insertions and deletions in a priority queue. The insert and delete operations of this scheme keep the heap balanced; hence each operation still takes $O(\log M)$ steps, where M is the size of the heap. The scheme also retains the strict priority ordering of the serial-access heap algorithms; i.e., a delete operation returns the best key of all keys that have been inserted or are being inserted at the time delete is started. The scheme allows $O(\log M)$ processors to manipulate the heap simultaneously. For a large number of processors, the speedup improvement factor due to our scheme grows as $O(\log M)$. We have incorporated the concurrent-heap scheme in a parallel branch-and-bound algorithm for solving the traveling salesman problem, and have obtained significantly higher speedups than with the serial-access schemes.

Note that even in the concurrent-access heap scheme, at most $O(\log M)$ processors can manipulate the heap concurrently. To allow greater concurrency, it seems necessary to relax the strictness of the priority queue. In [9], we present several "distributed" formulations of priority queue that permit $O(M)$ concurrency, and test their effectiveness in parallel branch-and-bound.

Acknowledgements: We would like to thank Jit Biswas for many useful discussions concerning CHEAP, and Dan Miranker for comments on an earlier draft of this paper. Center for Automation Research, University of Maryland, provided access to a large BBN Butterfly parallel processor.

References

[1] A. V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] M. J. Atallah and S. R. Kosaraju. A generalized dictionary machine for VLSI. *IEEE Transactions on Computers*, C-34, No. 2:151–155, 1985.

[3] Jit Biswas and James C. Browne. Simultaneous update of priority structures. In *Proceedings of International Conference on Parallel Processing*, pages 124–131, 1987.

[4] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, Boston, MA, 1987.

[5] Carla S. Ellis. Concurrent search and insertion in 2–3 trees. *Acta Informatica*, 14:63–86, 1980.

[6] Carla S. Ellis. Concurrent search and insertion in avl trees. *IEEE Transactions on Computers*, C-29 No 9:811–817, Sept 1980.

[7] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, MD, 1978.

[8] S.-R. Huang and Larry S. Davis. A tight upper bound for the speedup of parallel best-first branch-and-bound algorithms. Technical report, Center for Automation Research, University of Maryland, College Park, MD, 1987.

[9] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, pages 122–126, August 1988.

[10] Charles E. Leiserson. *Area-Efficient VLSI Computation*. MIT Press, Cambridge, MA, 1983.

[11] U. Manber and R.E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9, 3:439–455, 1984.

[12] Joseph Mohan. Experience with two parallel programs solving the traveling salesman problem. In *Proceedings of International Conference on Parallel Processing*, pages 191–193, 1983.

[13] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, NY, 1987.

[14] Michael J. Quinn and Narsingh Deo. Data structures for the efficient solution of graph theoretic problems on tightly-coupled MIMD computers. In *Proceedings of International Conference on Parallel Processing*, pages 431–438, 1984.

[15] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys*, 16,No 3:319–348, September 1984.

[16] Michael J. Quinn and Narsingh Deo. An upper bound for the speedup of parallel branch-and-bound algorithms. Technical report, Purdue University, Pullman, Washington, 1984.

[17] V. Nageshwara Rao and V. Kumar. Concurrent access of priority queues. Technical Report TR88-06, Computer Science Department, University of Texas at Austin, 1988.

[18] V. Nageshwara Rao, V. Kumar, and K. Ramesh. Parallel heuristic search on a shared memory multiprocessor. Technical Report AI TR87-45, University of Texas at Austin, 1987.

[19] A. Somani and V. Agarwal. An unsorted dictionary machine for VLSI. In *Proceedings of 1984 Symposium on Computer Architecture*, pages 142–150, 1984.

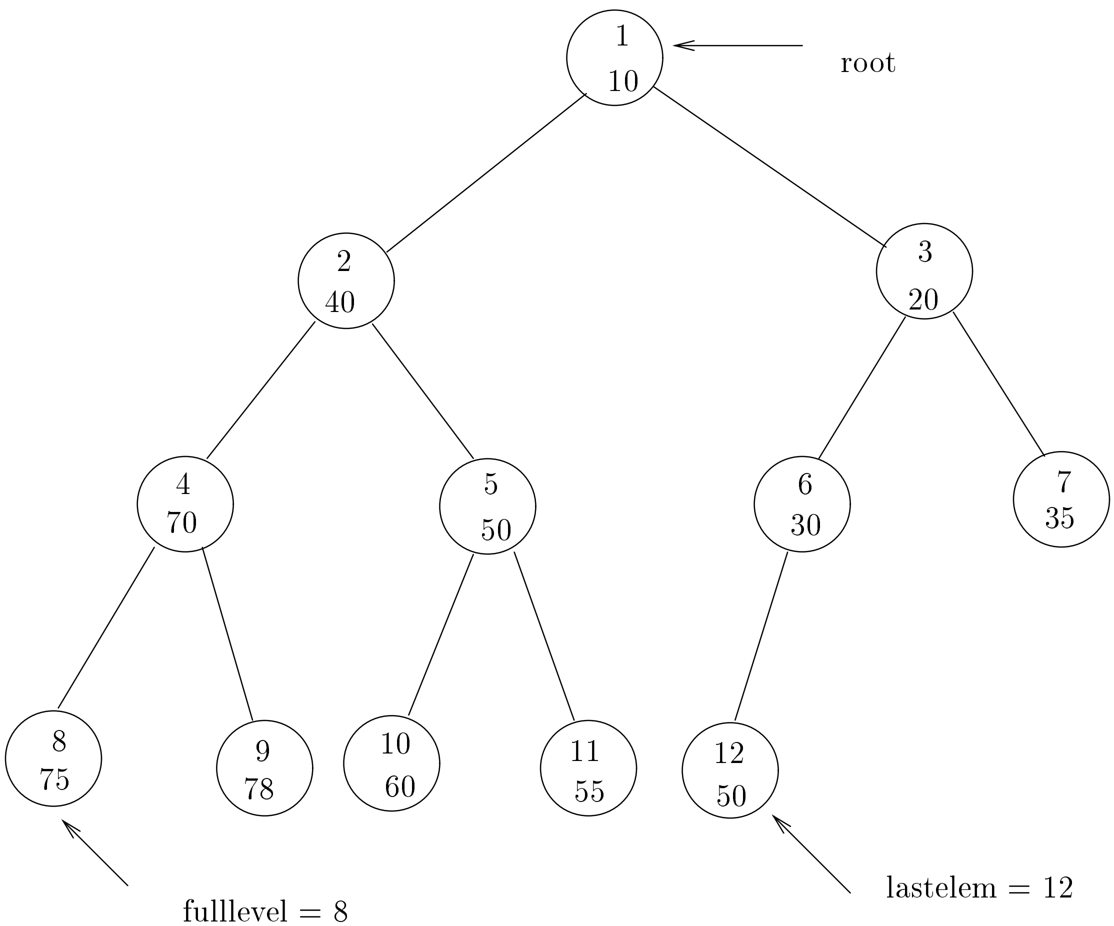

Figure 1: A heap of 12 keys. Upper half of the circle contains the node number, and the lower half contains the value of the key.

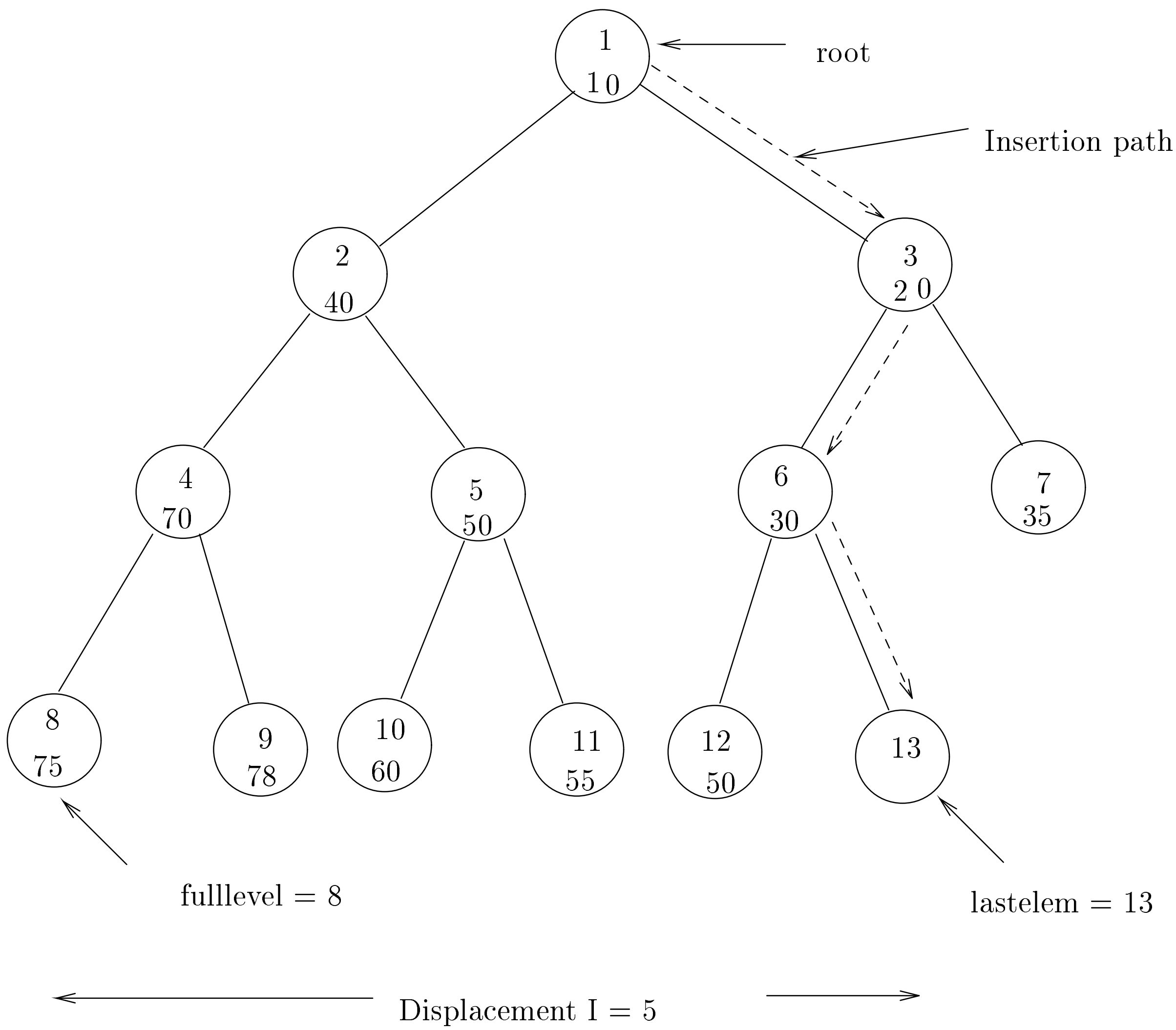

Figure 2: An example of how the insertion path is computed in insert_t. A new key is inserted into the heap at node 13. I = 5 = (101) in the binary representation; length of the insertion path = 3.

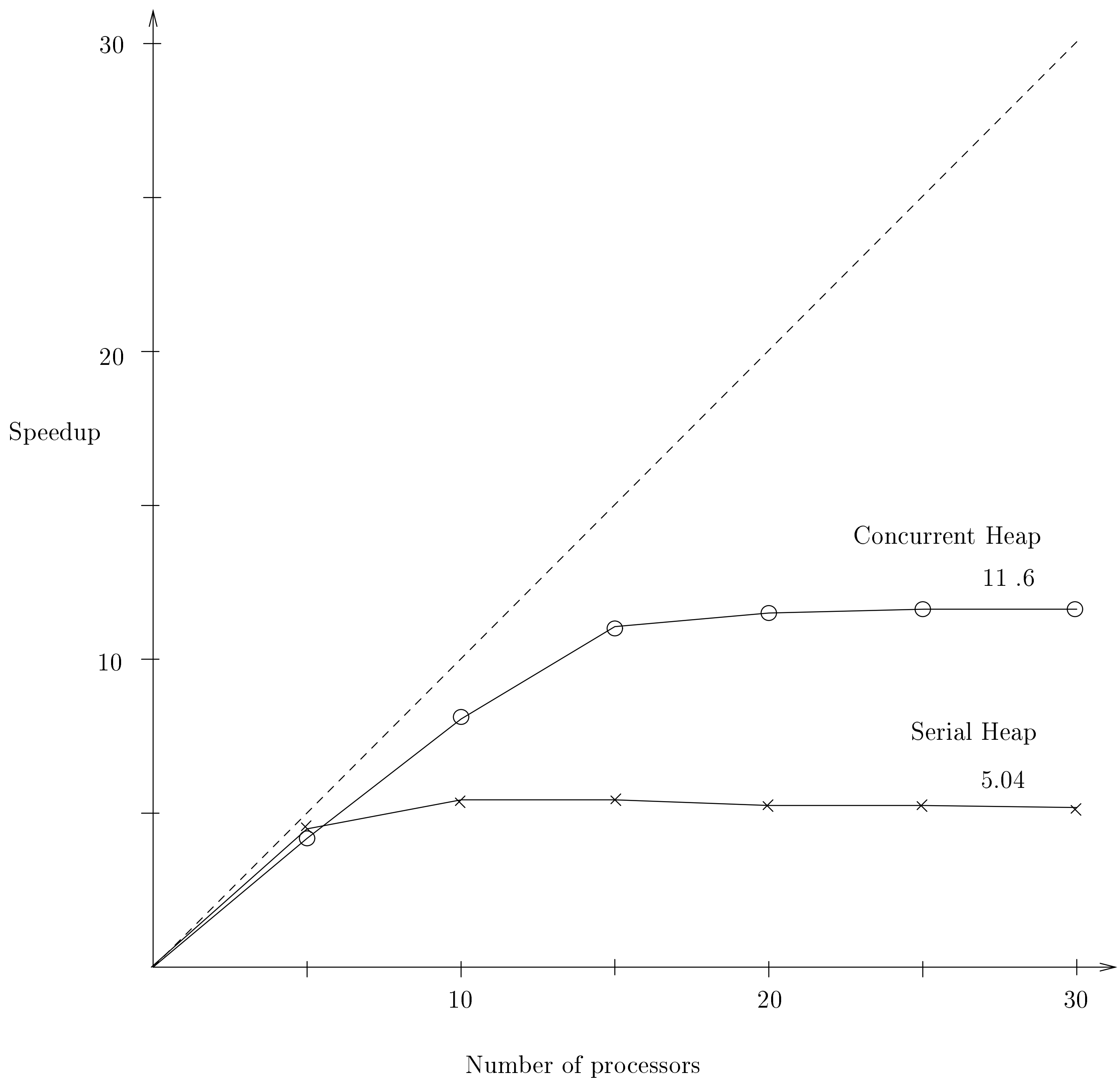

Figure 3: Plot of speedups obtained in execution of access-think cycles for delete operation. Think time is set to 10ms ($\approx 5 * T^s_{access}$).

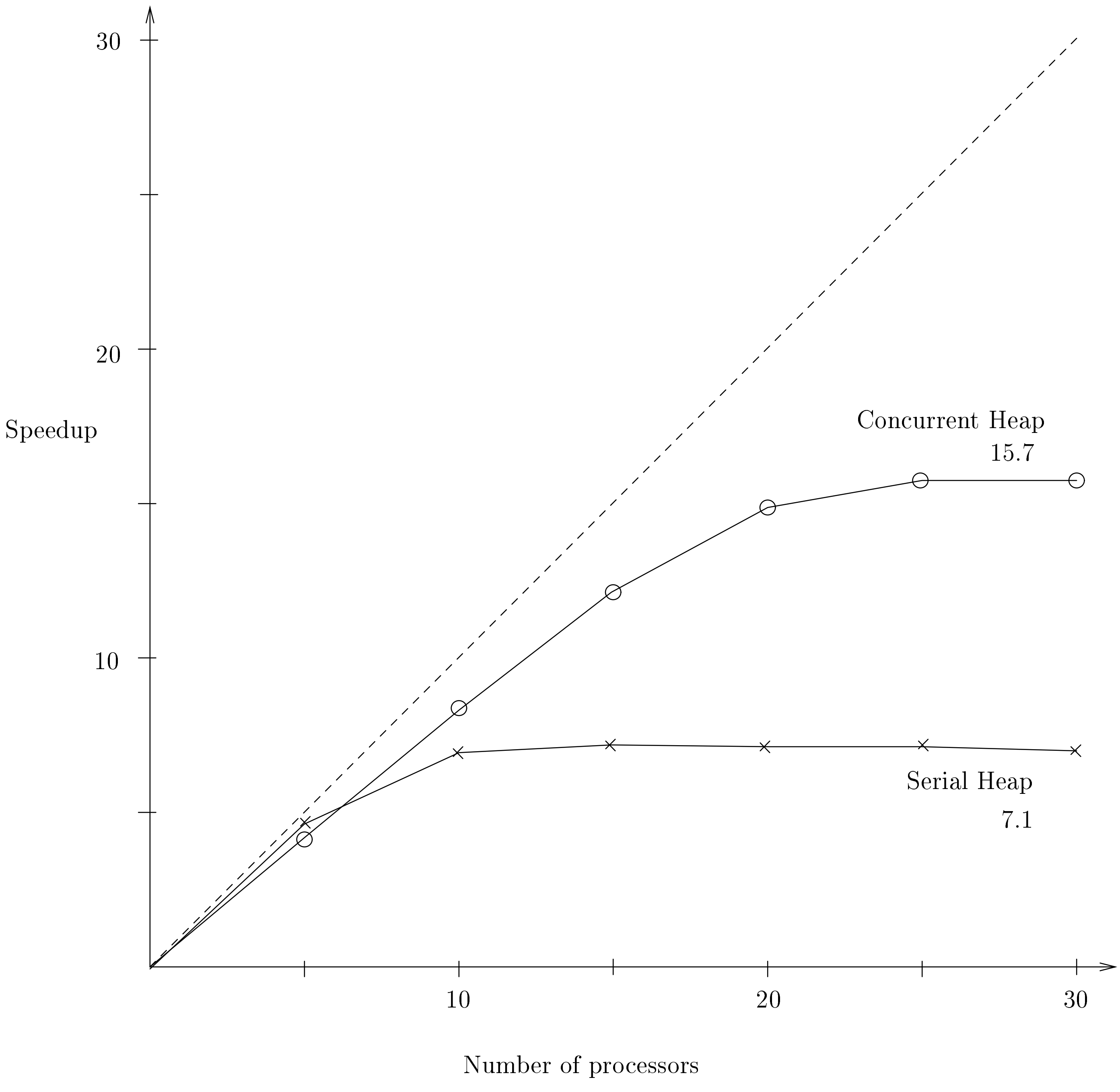

Figure 4: Plot of speedups obtained in execution of access-think cycles for insert operation. The numbers inserted are in a decreasing order. Think time is set to 10ms ($\approx 7 * T^{s}_{access}$).

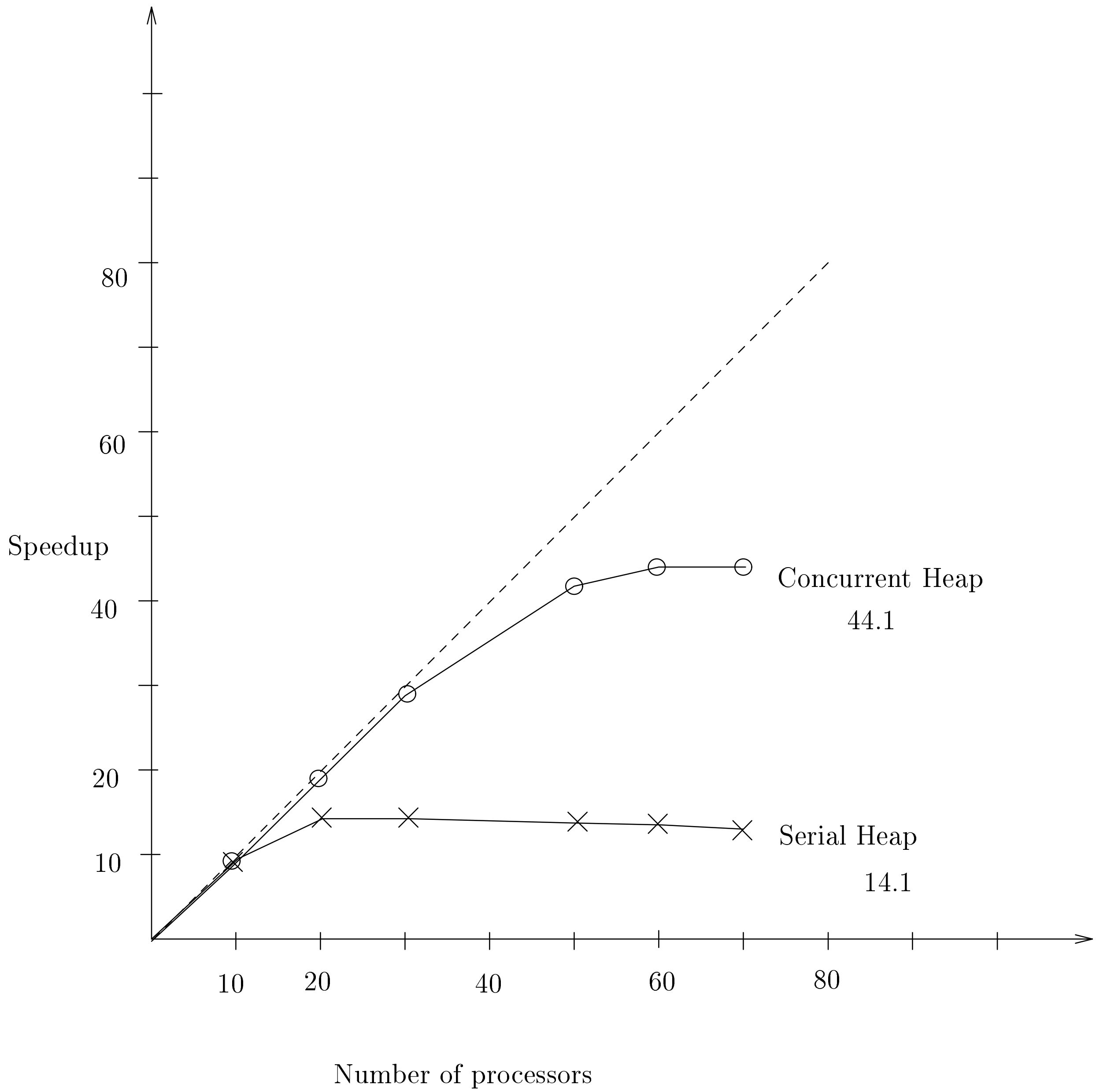

Figure 5: Comparison of performance of concurrent-access heap vs serial-access heap for Parallel Branch and Bound algorithm.