

Concurrent Data Structures for Hypercube Machine

M. R. Meybodi

Computer Science Department, Ohio University

Athens, Ohio 47501

Abstract

To efficiently implement parallel algorithms on parallel computers, concurrent data structures (data structures which are simultaneously updatable) are needed. In this paper, three implementations of a priority queue on a distributed-memory message passing multiprocessor with a hypercube topology are presented. In the first implementation, a linear chain of processors is mapped onto the hypercube, and then a heap data structure is mapped onto the chain, where each processor stores one level in the heap. A similar approach is taken for the second implementation, but in this case, a banyan heap data structure is mapped onto the linear chain of processors. Again, each processor in the chain becomes responsible for one level of the data structure. For the third implementation, the banyan heap data structure is again used, but the mapping is not onto linear chain of processors. Instead, the banyan heap is mapped onto processors column by column, so that the algorithm can make better use of the concurrent processing capabilities of the hypercube topology in order to reduce bottlenecks in the first processor, an effect noted in the use of the linear chain employed by the first two implementations. The key advantage of banyan heap over the heap is that with banyan heap it is possible to retrieve elements at different percentile levels.

Keywords and Phrases: Concurrent Data Structure, Hypercube, Banyan Heap, Parallel Algorithm

1 Introduction

Priority queue data structure is a very important data structure which has found application in varieties of situations such as: discrete event simulation systems, timed-shared computing system, finding shortest paths in a graph [17], finding the minimum spanning tree of a graph [17], and iteration in numerical schemes based on the idea of repeated selection of an item with smallest test criteria [16], to mention a few.

Such a data structure is a set of elements each of which has an associated number, its priority for it. For each element x , $p(x)$, the priority of x is a number from some linearly ordered set. Standard operations on a priority queue are *INSERT*, which inserts an element and its associated priority into the priority queue, and *XMAX*, which deletes the element with the highest priority from the queue. Let P denote the set of all element-priority pairs. Define

$$P(s) = \{(x, p(x)) | p(x) = s \text{ and } (x, p(x)) \in P\};$$

The effect of priority queue operations are as follows:

INSERT($x, p(x)$) :

$$P \leftarrow P \cup \{(x, p(x))\}.$$

Response is null.

XMAX:

$$P \leftarrow P - P(p_{max}) \text{ where } P(p_{max}) \text{ is the pair with the highest priority.}$$

Response is $P(p_{max})$.

Since most of the applications of priority queues are computationally intensive, it is important to have an efficient implementation. Three kinds of implementations of priority queue have been reported in the literatures: sequential algorithms for implementation on uniprocessor, parallel algorithmic architectures for realization in hardware, and parallel algorithms for implementation on parallel computers. In the next three paragraphs we review the literature for these approaches.

There are number of implementations for priority queue on a uniprocessor. Priority queue can be maintained as a sorted list, in which case, deleting the element with the highest priority takes $O(1)$ time but insertion takes $O(N)$ time, or it can be maintained as an unsorted list, in which case insertion takes constant time but $XMAX$ takes $O(N)$ time, where N is the number of element-priority pairs in the priority queue. Other implementations which lend themselves to logarithmic time are height or weight balanced tree, partially ordered tree(heap), leftist trees suggested by C. A. Crane [30], pagodas [37], skew heaps [38], implicit heaps [30], and binomial queues [39]. The most widely used implementation of priority queue is with a heap data structure. A heap is a full k -array tree such that the priority of the element at any node in the tree is greater than priority of the element at each of its children. Thus the element with the highest priority is always at the root of the heap. Operation $INSERT$ is performed by adding the pair to the last level of the heap and then converting the resulting complete tree into a heap by pushing that pair up the tree recursively. $XMAX$ works by replacing the pair at root with the pair residing in the rightmost node of the last level of of heap and then converting the resulting complete tree into a heap by pushing the pair at the root down the tree recursively. If $k = 2$ then we have binary heap which is the most wildly studied special case.

A number of multiprocessor design for maintaining priority queue have been proposed in the literatures. These designs can be classified into two main groups. 1) designs with small number of processors each having an small amount of memory, and 2) designs with small number of processors each having a large amount of memory. Group 1 designs [5,13,22-27,33] which in turn can be classified into systolic tree designs and systolic array designs are suitable for implementation by VLSI hardware whereas group two designs [2,4,11,29,42] can be implemented either in VLSI or any general purpose tightly or loosely coupled multiprocessor architecture.

Existing algorithms for multiprocessors are divided into two groups: those for shared memory multiprocessors and those for distributed memory mutiprocessors. When designing algorithms for shared memory multiprocessors the focus is on reducing the interference between concurrent processes accessing the data structure whereas in the case of distributed memory multiprocessor the focus is on exploiting the parallelism in the data structure. Concurrent algorithms for manipulating binary tree due to Kung and Lehman [18], concurrent algorithms for B-tree due to Lehman and Yao [19], algorithms for concurrent search and insertion of data in AVL-tree and 2-3 trees due to Ellis [20,21], concurrent algorithm for insertion and deletion on the heap due to Rao and Kumar [8], and Biswas and Browne [1] are examples of algorithms for shared-memory multiprocessor. Examples of algorithms for distributed memory multiprocessor are: balanced cube due to Dally [9], sorted chain due to Omondi and Brock [13], and binary search mesh due to Meybodi [41].

It should be noted that most of the machines or algorithms cited in this paper offer capabilities which go beyond priority queue operations. Other operations supported by most of the cited algorithms or machines are operations $SEARCH$, $NEAR$, $UPDATE$, and $DELETE$ on set of key-record pairs. In the context of priority queue a record is the element and the corresponding key is the priority associated to that element.

In this paper we present three implementations of priority queue on distributed-memory

message passing multiprocessor with hypercube topology. The first implementation is based on heap data structure. The heap is mapped into a linear chain of processors which is then embedded into hypercube. The other two implementations are based on banyan heap data structure. They differ in the way that the banyan is mapped into the hypercube. The key advantage of banyan heap over the heap is that with banyan heap it is possible to retrieve elements at different percentile levels. A hypercube computer is briefly described below.

An d dimensional hypercube machine consists of 2^d processor nodes interconnected to one another to form a d dimensional cube. In an d dimensional cube two nodes are connected if and only if the binary representation of their numbers differ by one and only one bit. Each processor node in the hypercube has a processor and a local memory for that processor. The processors work independently and asynchronously and communicate by passing messages [3].

The rest of this paper is organized as follows. Section 2 gives a heap based implementation of priority queue on a distributed-memory message-passing. Section 3 defines banyan graphs and banyan heaps. In section 4 we discuss an implementation of priority queue based on banyan heap data structure. Section 5 derives analytical formula for the percentile level of a retrieved element. In section 6, the second banyan heap based implementation of priority queue is presented. The last section is the conclusion.

2 First Implementation

This implementation is based on heap data structure. The heap is first mapped into a linear chain of n processors. This chain is then embedded into the hypercube in such a way as to preserve the proximity property, i.e., so that any two adjacent processors in the chain are mapped into neighbor nodes in the hypercube [3]. The use of Gray codes is one known technique to obtain a mapping which preserves the proximity property. This technique can be explained as follows: If the binary representation of the node number in the chain is $b_{d-1}b_{d-2}\dots b_0$, then it is mapped into the node with number $c_{d-1}c_{d-2}\dots c_0$, where $c_i = b_i + b_{i+1}$, if $(i < d - 1)$, and $c_i = b_i$, if $i = d - 1$. For example, node number 26 (011010) in the chain is mapped into node 23(010111) in the hypercube. If $2^d < \log n$ then more than one level of the heap may be assigned to a single processor in the chain. This extension will not be discussed here.

Processor p_i , ($0 \leq i \leq n - 1$), of the chain stores the i^{th} level of the heap. Each node has 6 fields: DATA, PRIORITY, LCHILD, RCHILD, LEMPTYNODES, and REMPTYNODES. For a node, the DATA field holds an element and the PRIORITY field holds the priority associated with that element, LCHILD and RCHILD hold respectively pointers to the leftchild and rightchild of that node, and LEMPTYNODES and REMPTYNODES hold the number of null nodes (nodes with no information) in the left and right subtrees of that node. Initially, the DATA field of all the nodes are set to null and the PRIORITY field of all the nodes are set to -1. The LEMPTYNODES and REMPTYNODES fields of all the nodes at level i are initialized to $2^{n-i} - 1$. These two fields are updated as data elements are inserted into or deleted from the heap. Information about the number of empty nodes is used by *INSERT* operation to decide which path in the heap should be followed during the insertion process. Lack of such information may lead to an overflow situation in the last processor. This happens if the *INSERT* operation moves along a path in which all the nodes are non-empty.

The process running on each processor in the hypercube consists of two distinct parts. The first part belongs to the application that runs on all the processors of the hypercube. The second part, called *executive*, is responsible for the execution of the priority queue operations. An executive can receive and process many messages simultaneously. Priority queue operations are initiated by the application part of the processes running on the processors

4 Second Implementation

The effective implementation of banyan heap on hypercube requires efficient mapping of the banyan structure among the processors of the hypercube. We examine two such mappings. The first mapping is obtained by first mapping the banyan into a linear chain and then embedding the chain into the hypercube (See figure 3). The second mapping is obtained by collapsing columns of the banyan into single processors [34]. That is each base-apex path in the banyan with identically labeled vertices is mapped into one processor in the hypercube with the same label. According to this mapping, if two nodes are adjacent in the banyan then they are mapped into two adjacent processors in the hypercube, those having labels that differ exactly in the i^{th} digit. See figure 6 for an example. In this section we consider the first mapping.

Before we give a detailed description of the operations *XMAX* and *INSERT* we show how to compute the addresses of the children of a given node. The nodes at a given level are stored sequentially in an array of size M in the local memory of the corresponding processor. Let n_{li} to denote the i^{th} node on the l^{th} level of the banyan where $0 \leq i \leq M, 0 \leq (\log M + 1)$. Then n_{li} is connected to two nodes $n_{(l+1)i}$ and $n_{(l+1)m}$, where m is the integer found by inverting the i^{th} most significant bit in the binary representation of i . We call nodes $n_{(l+1)i}$ and $n_{(l+1)m}$ the right and left children of node n_{li} , respectively. Therefore, the left child and right child of node i at level k are stored respectively in the i^{th} location and m^{th} location of the array residing in the local memory of processor $k + 1$.

All the *XMAX* and *INSERT* operations initiated at different processors are sent to the head of the embedded chain and are executed in a pipelined manner. When operation *INSERT* is received by processor p_0 , it first finds the leftmost partition which has at least one empty node. This can be done using information stored in the *REMPYTHONES* and *LEMPYTHONES* fields of the apexes in $O(M)$ time. It then pushes the element requested to be inserted down the banyan heap using operation *insert-adjust*. The operation *insert-adjust* pushes the element down (along the paths from the root of the partition to the bases) until it finds its correct position. This requires $O(\log M)$ time. Thus *INSERT* is an $O(M)$ operation.

In the codes given below we use the following syntax and semantic for send and receive instructions. The instruction *Send(<processor>, <instruction>)* sends instruction *<instruction>* to processor *<processor>* for execution. The execution of *receive(<processor>, (information))* causes the information specified by the second argument be obtained from processor *<processor>* and forwarded to the requesting processor (the processor executing the receive instruction). A receive instruction executed by processor p_i is not complete until a message is received from processor p_{i+1} .

Upon receiving *INSERT(p, (item, priority))* by processor p_0 , it executes the following codes. The letter p is the address of the leftmost apex, and $(item, priority)$ is the pair requested to be inserted.

```

found ← false
While (not found) do
    if DATA(p) ≠ null then
        if priority > PRIORITY(p)
            begin
                if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p) ≠ 0 then
                    begin
                        if LEMPTYNODES(p) > REMPTYNODES(p) then
                            begin
                                p' ← RCHILD(p);

```

```

if LEMPTYNODES(p) > REMPTYNODES(p) then
begin
    p' ← LCHILD(p) ;
    LEMPTYNODES(p) ← LEMPTYNODES(p)-1
end
else
begin
    p' ← RCHILD(p);
    REMPTYNODES(p) ← LEMPTYNODES(p) -1
end
end
send( $p_{i+1}$ , 'insert-adjust(p',(DATA(p),PRIORITY(p)))');
DATA(p) ← item; PRIORITY(p) ← priority
end
else
begin
    if LEMPTYNODES(p) ≠ 0 or REMPTYNODES(p) ≠ 0 then
begin
    if LEMPTYNODES(p) > REMPTYNODES(p) then
begin
        p' ← RCHILD(p);
        REMPTYNODES(p) ← REMPTYNODES(p) -1
    end
    else
begin
        p' ← LCHILD(p);

        LEMPTYNODES(p) ← LEMPTYNODES(p) -1
    end
    send( $p_{i+1}$ , 'insert-adjust(p',(item,priority))');
end
else
begin
    DATA(p) ← item; PRIORITY(p) ← priority
end
end
else
begin
    DATA(p) ← item;
    PRIORITY(p) ← priority
end;

```

XMAX operation first locates the apex which contains the element with the highest priority, reports that element to the outside world, and then fills up that apex with the element in one of its children. *xmax-adjust* is responsible for restructuring the banyan as it moves down the heap. When *XMAX(p)* is received by processor p_1 , it executes the following codes, where p is the address of the leftmost apex. This address is known to the outside world (front end computer).

```

p' ← p
while NEXT(p) ≠ nil and DATA(NEXT(p)) ≠ null do
begin
    if PRIORITY(p) > PRIORITY(NEXT(p)) then p' ← p
    p ← NEXT(p)
end
send('outside world', DATA(p'));
receive (p2, ((PRIORITY(RCHILD(p')),DATA(RCHILD(p'))),
                (PRIORITY(LCHILD(p')),DATA(LCHILD(p'))))) )
if DATA(RCHILD(p')) ≠ null or DATA(LCHILD(p'))≠ null then
    if DATA(RCHILD(p')) > DATA(LCHILD(p')) then
        begin
            DATA(p') ← DATA(RCHILD(p'));
            PRIORITY(p') ← PRIORITY(RCHILD(p'));
            REMPTYNODES(p') ← REMPTYNODES(p') + 1;
            send(p2, 'xmax-adjust(RCHILD(p'))')
        end
    else
        begin
            DATA(p') ← DATA(LCHILD(p'));
            PRIORITY(p') ← PRIORITY(LCHILD(p'));
            LEMPTYNODES(p') ← LEMPTYNODES(p') + 1;
            send (p2, 'xmax-adjust(LCHILD(p'))')
        end
    else
        begin
            DATA(p') ← null;
            PRIORITY(p') ← -1
        end

```

Processor p_i , ($1 \leq i \leq L$), upon receiving $xmax-adjust(p)$) executes the following codes.

```

receive(pi+1, ((PRIORITY(RCHILD(p)),DATA(RCHILD(p))),
                (PRIORITY(LCHILD(p)),DATA(LCHILD(p)))) );
if DATA(RCHILD(p)) ≠ null or DATA(LCHILD(p)) ≠ null then
    if DATA(RCHILD(p)) > DATA(LCHILD(p)) then
        begin
            DATA(p) ← DATA(RCHILD(p));
            PRIORITY(p) ← PRIORITY(RCHILD(p));
            REMPTYNODES(p) ← REMPTYNODES(p) + 1;
            send(pi+1, 'xmax-adjust(RCHILD(p))')
        end
    else
        begin
            DATA(p) ← DATA(LCHILD(p));
            PRIORITY(p) ← PRIORITY(LCHILD(p));
            LEMPTYNODES(p) ← PRIORITY(p) + 1;
            send(pi+1, 'xmax-adjust(LCHILD(p))')
        end

```

```

    end
else
begin
    DATA(p) ← null;
    PRIORITY(p) ← -1
end

```

Remark 3 The elements stored in the apex nodes are not ranked in any particular order. This speeds up the insertion process, but leads to $O(M)$ time for deletion. It is possible to insert the elements in such a way that the element with the highest priority is always available at the leftmost apex in which case locating the correct apex to initiate the insertion takes $O(M)$ time. This method seems to be more efficient due to the fact a portion of the time spent to find the correct position can be overlapped with the time spent to locate an apex with zero REMPTYNODES or zero LEMPTYNODES. In the algorithms presented above we have used the first approach. The second approach will be reported in another paper.

From the properties of *SW*-banyan graphs and the above algorithms, we can state the following results.

Theorem 1 *Operation XMAX requires $O(M)$ time to complete.*

Theorem 2 *Operation INSERT requires $O(M)$ time to complete.*

Theorem 3 *The second implementation offers $O(M/\log M)$ throughput.*

Lemma 1 a) *The insert-adjust operation never encounters a node which is non-null and has zero LEMPTYNODES and zero REMPTYNODES.* b) *The insert-adjust operation always finds a null node to insert its element.*

Proof a) If operation *zmax-adjust* encountered a non- null node with LEMPTYNODES and REMPTYNODES fields equal to zero then it must have been initiated from an apex with both REMPTYNODES and LEMPTYNODES equal to zero. This is impossible according to the algorithm for *INSERT*. b) Proof is immediate from the proof of part a and the definitions of LEMPTYNODES and REMPTYNODES.

Remark 4 Deletion of an element from partition i may cause one of the elements in other partitions whose nodes are reachable from apex i to become null. This happens if a delete operation causes the *zmax-adjust*, on its way down the heap, to move up the content of one of the leaf nodes of partition i to fill up its parent which has been emptied by *zmax-adjust* operation at the previous step. The emptiness of this node now will be reflected in the REMPTYNODES or LEMPTYNODES of apex i . This node is now reachable by partition from apex i and will be filled by an insertion initiated at apex i . The maximum number of nodes that may become reachable by partition from apex i as a result of a deletion is equal to $(L + 1) - D$, where D is the depth of partition i .

Lemma 2 *Zero REMPTYNODES and zero LEMPTYNODES for an apex does not imply that all the nodes in the corresponding partition are non-null.*

Proof From remark 4.

Proof. We prove this lemma by induction on d .

Basis: For $d = 1$, $NP_j = 0$ for all $1 \leq j \leq \log N + 1$.

Induction: Assume that

$$NP_k = \left\lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \right\rfloor.$$

Consider $d+1$. Either d is even or it is odd. If d is even then apex $d+1$ is the root of a partition with depth 1. Therefore,

$$NP_k = \left\lfloor \frac{d+1 - \sum_{j=1}^{k-1} NP_j - (NP_1 + 1)}{2} \right\rfloor,$$

that is,

$$NP_k = \left\lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \right\rfloor.$$

For the case that d is odd we consider two subcases: Apex $d+1$ is either the root of a partition of depth k or it is the root of a partition with depth h , ($h < k$). If $d+1$ is the root of a partition of depth d then we will have

$$NP_k + 1 = \left\lfloor \frac{d+1 - \sum_{j=1}^{k-1} NP_j}{2} \right\rfloor.$$

Since d is even then we have

$$NP_k + 1 = 1 + \left\lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \right\rfloor,$$

or

$$NP_k = \left\lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \right\rfloor.$$

If apex $d+1$ is the root of a partition with depth h where $1 \leq h < \log N + 1$ and $h \neq k$ then we have

$$NP_k = \left\lfloor \frac{d+1 - \sum_{1 \leq j \leq k-1, j \neq h} NP_j - (NP_h + 1)}{2} \right\rfloor,$$

or

$$NP_k = \left\lfloor \frac{d - \sum_{j=1}^{k-1} NP_j}{2} \right\rfloor.$$

Lemma 10 *The total number of non-null nodes in a full $M \times M$ partitioned rectangular SW-banyan up to apex d , written $\text{size}(M, d)$, is given by:*

$$\text{size}(M, d) = \sum_{k=0}^{M \log M} 2^k + \sum_{j=1}^d NP_j 2^j$$

Proof From lemma 9.

Lemma 11 *If an $M \times M$ rectangular SW-banyan partitioned heap is full up to apex d then the element stored at apex 1 is at percentile level*

$$\frac{(2M - 1) * 100}{\text{size}(M, d)}.$$

Proof $\text{size}(M, d)$ is the total number of nodes in an $M \times M$ rectangular SW-banyan heap which is full up to apex d and $2M - 1$ is the total number of nodes in partition.1.

Lemma 12 In an $M \times M$ partitioned rectangular SW-banyan heap which is full up to apex d , if operation $XMAX$ investigates $i, (i < d)$, non-null apexes then the percentile of the reported element is

$$\frac{\text{size}(M, i) * 100}{\text{size}(M, d)}.$$

Proof From lemma 10.

Lemma 13 If operation $XMAX$ examines apexes 1 through d in an $M \times M$ rectangular banyan heap then the percentile of the reported element is smaller than or equal to

$$\frac{\text{size}(M, d) * 100}{(\text{size}(M, M) - \text{NULLNODES}(M, M))}.$$

Proof If a banyan heap is full up to apex d then by lemma 12 the percentile of the element reported by operation $XMAX$ is

$$\frac{\text{size}(M, d) * 100}{\text{size}(M, d)}.$$

By lemma 7, this can be written as

$$\frac{\text{size}(M, d) * 100}{(\text{size}(M, M) - \text{NULLNODES}(M, M))}.$$

But the banyan is not full up to apex d and therefore some of the nodes which are reachable from apexes 1 to d are null. Let F be the total number of such nodes. Therefore the percentile of the element reported by $XMAX$ operation will be

$$\frac{(\text{size}(M, d) - F) * 100}{(\text{size}(M, M) - \text{NULLNODES}(M, M))}.$$

This proves the lemma.

Remark 5 A priority queue can also be implemented as banyan heap. A partitioned banyan heap can be converted into a banyan heap by an operation called *adjust*. $M \log M - 2$ *adjust* operations are broadcast by the $XMAX$ operation when it inserts an element into an empty partition. These *adjust* operations cause some of the elements in the nodes of those partitions which are reachable from apex i to move up and fill up the nodes of partition i . As a result, all the nodes whose contents (empty or non-empty) have been moved by the *adjust* operation become reachable by partition from apex i . It should be noted that some of the *adjust* operations initiated at processor p_0 by $XMAX$ operation may not have any effect on the structure of the heap.

The advantage of banyan heap over partitioned banyan heap is that it allows a more uniform distribution of data elements among the partitions in the heap and leads to a more uniform increase in the percentile level of the reported element as the number of examined apexes increases.

Algorithms for $XMAX$, *xmax-adjust* and *insert-adjust* are the same for the banyan heap. The operation *INSERT* and the new operation *adjust* are described in [4].

tree requesting the computation of the minimum.

Operations *XMAX* and *INSERT* for this implementation are described below.

INSERT: An *INSERT* operation initiated at node x first computes the address of the leftmost apex y whose partition has at least one empty node (using *Global Min* operation). The element-priority pair is then sent from node x to the node which contains apex y . The pair will be inserted into the partition rooted at apex y according to the procedure described for the second implementation.

The problem with the *INSERT* operation as given above is that an insertion operation may find a partition (reported to have empty positions) full when it tries to insert a pair into that partition, and therefore blocked and unable to proceed. This is caused by allowing several *INSERT* operations to search the list of apexes concurrently for their point of insertions, which, as a result more than one *INSERT* operation, may receive the same apex whose corresponding partition has only one empty position as the point of insertion. In this situation a new *INSERT* operation may be reissued at the node which blocked the insertion.

XMAX: An *XMAX* operation initiated at node x first determines the apex which contains the element with the highest priority (using *Global Min* operation) and then sends an operation, called *adjust*, to the node containing that apex. Operation *adjust* first reports the element to node x , and then adjust the banyan heap as described in the previous section.

The problem with *XMAX* operation as described above is that due to concurrent access to the list of apexes by several processors, the same apex may be reported to several *XMAX* operations as the holder of the element with the highest priority. This may lead to the situation in which elements returned and subsequently deleted by some of the *XMAX* operation do not have the highest priority in the banyan heap at the time of removal. One solution to this problem is to send all the *XMAX* operations (issued at different nodes) to node p_0 and let node p_0 execute them sequentially. This limits the amount of concurrency obtainable in the system. The amount of obtainable concurrency starts to decrease due to this strategy when the number of elements in the banyan heap exceeds $2^{\log M+2} - 1$ for the first time.

Theorem 5 *Operation XMAX requires $O(\log M)$ time to complete.*

Proof: Operation *XMAX* consist of 3 parts:

1. finding the apex containing the element with the highest element.
2. reporting the element to the node initiated the operation.
3. adjusting the banyan heap

Each of these steps requires $O(\log M)$ time and hence the total time of $O(\log M)$ for *XMAX*.

Theorem 6 *Operation INSERT requires $O(\log M)$ time to complete.*

Proof: Similar to theorem 5.

Theorem 7 *An XMAX operation needs no more than $2(M-1)+2*\log M$ communications.*

Theorem 8 *Third implementation offers $O(M/\log M)$ throughput.*

Proof: The third implementation can perform M operations at a time and each operation requires $O(\log M)$ time, and hence $O(M/\log M)$ throughput.

In what follows we first define a few terms and then introduce the concept of consistency for concurrent data structure.

Definition 12 *The timestamp of an operation is the time at which the operation is initiated at a node and the timestamp of a communication is the timestamp of the operation which generated that communication.*

Definition 13 *Operations O_1, O_2, O_3, \dots , and O_q with timestamps t_1, t_2, t_3, \dots , and t_q are said to be a sequence of operations if $t_1 < t_2 < \dots < t_q$ and operations O_i , $1 < i \leq q$ are the only operations issued between t_1 and t_q .*

Definition 14 *A concurrent data structure is said to be consistent if the concurrent execution of any sequence of operations on the data structure gives the same result as executing the operations sequentially.*

Theorem 9 *The third implementation is not consistent*

Proof: Consider a sequence of $XMAX$ operations waiting in node p_0 for execution. Due to the fact that these $XMAX$ operations are executed sequentially in increasing order of their timestamps, it is quite possible for an $INSERT$ operation which have a larger timestamp than any of the $XMAX$ operations to be executed before all the $XMAX$ operations are completed.

Theorem 10 *The third implementation is consistent if an access made to an apex by an operation is not started unless all the operations with the lower timestamps have completed all their $M+1$ accesses to the apices.*

Below we describe a procedure for making the third implementation consistent.

An operation is first recorded by all the processors in the hypercube. The processor which initiated the operation broadcasts that operation together with its timestamp to all the processors using its fan out tree. The operation and its timestamp is added to the list of incomplete operations maintained by resident executive of each processor. After the operation is added to the list of incomplete operations by a processor, that processor sends a notification signal to the processor that initiated the operation. The notification signals issued by the processors are combined according to the fanout tree rooted at the processor which initiated the operation; no processor sends a notification signal to its parent in the fan out tree unless all its children have noted that operation. After the root processor received all the notification signals, the execution of the operation starts. During the execution of the operation no access is made to an apex unless all the operations with lower timestamps have completed their access to that apex. When an operation completes its last access to the apices, it asks all the executives to remove that operation from their list of incomplete operations. The removal operation is performed in the same fashion as the operation of recording an operation. After an operation is removed from the list of incomplete operations of an executive, the next pending operation will be allowed to perform its access to the corresponding apex.