ISSN:0975-9646

Ranjeet Kaur et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 6 (1) , 2015, 851-857

MSL Based Concurrent and Efficient Priority Queue

Ranjeet Kaur#1, Dr. Pushpa Rani Suri#2

1Student, 2Professor

1, 2Department of Computer Science and Application.

Kurukshetra University, Kurukshetra

Haryana, India

Abstract— **Priority queues are fundamental in the design of modern multiprocessor algorithms. Priority queues with parallel access are an attractive data structure for applications like prioritized online scheduling, discrete event simulation, or branch-and-bound. This paper proposes an alternative approach: to base the design of concurrent priority queues on the Modified Skip List data structure. To this end, we show that a concurrent modified Skip List structure, following a simple set of modifications, provides a concurrent priority queue with a higher level of parallelism. Many algorithms for concurrent priority queues are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the system's overall performance. Non-blocking algorithms avoid blocking, and are either lock-free or wait-free. Previously known non-blocking algorithms of priority queues did not perform well in practice because of their complexity, and they are often based on non-available atomic synchronization primitives.**

Keywords— **TMSL, threaded chain , Put your keywords here, keywords are separated by comma.**

I. INTRODUCTION

In recent years there is mismatch between the construct of scalable software and the availability of larger computing platforms. We have seen rapidly increase in the number of processors available on commercial multiprocessors. Priority queues are of fundamental importance in the design of modern multiprocessor algorithms. Priority queues are useful in scheduling, discrete event simulation, networking (e.g., routing and realtime bandwidth management), graph algorithms (e.g.,Dijkstra's algorithm), and artificial intelligence (e.g., A*search). In these and other applications, not only is it crucial for priority queues to have low latency, but they must also offer good scalability and guarantee progress. Though there is a wide range of literature addressing the design of concurrent priority queue algorithms

This paper begins to confrontation the issue of designing an efficient concurrent priority queue based on skip list data structure of Pugh et. al[1] and other popular heap structures found throughout the literature. [3; 4; 5; 6;7; 8;9 ;10;11;12;13;14;15;16;17]. Here we proposed an an alternative approach: for the design of concurrent priority queue on the modified skip list data structures of sartaj et. al[2]. This concurrent priority queue is designed with a change in the structure of modified skip list, it is presented in the simple form and produced significant performance gains.

The next three subsections in the introduction summarize the focal points of the paper.

II. PRIORITY QUEUE

Priority queues are a fundamental data structure with many applications. Priority queues manage a set of elements and support the operations an Insert of an item with a given priority, and a delete-min operation that returns the item of highest priority in the queue. Traditionally, priority queues have been implemented on the basis of heap[3; 4; 5; 6; 7; 8;9 ;10;11;12;13;14;15;16;17].or search trees[18; 19] data structures. Empirical evidence collected in recent years [6; 8; 20] shows that heap-based structures tend to outperform search tree structures. This is probably due to a collection of factors, among them that for rebalancing the heap there is no need to lock the heap, and that Insert operations on a heap can proceed from bottom to root, thus minimizing contention along their concurrent traversal paths. The concurrent priority structure based on heap given by Hunt et. al [8] is known to be the best effective structure. Its good performance is the result of several techniques for minimizing locking and contention: inserts traverse bottom up, only a single counter location is locked for a short duration by all operations, and a bit reversal scheme distributes delete requests that traverse top-down. There is one common problem with most of the algorithms for concurrent priority queues is the lack of precise defined semantics of the operations. The empirical evidence shows, that the algorithm balanced search trees and heaps suffer from the typical scalability impediments of centralized structures: sequential bottlenecks and increased contention.lotan et. al[21].

Haken et. al[22] presented a lock free algorithm of a concurrent priority queue that is for both pre-emptive as well as for fully concurrent environments. It was implemented using common synchronization primitives that are available in modern systems. In a skip list data structure the min element can be identified in O(1) time and deleted in O(log n) probalistic time,this was the one of the drawback of skip list data structure pointed by sartaj et.al[2]. The author introduced the concept of priority queue based on modified skip list data structured MSL[2]. The concurrent access of priority queue based on modified skip list is the initial effords, in this direction .

III. THE NEW APPROACH

According to sartaj.et.al [2], at first sight, it might imply that skip lists can be a better choice for priority queue than modified skip list. In case of skip list to search a list of n items, O (log n) level lists are traversed, and a constant number of items is traversed per level, making the expected overall complexity of an Insert or Delete operation on a SkipList O(logN)

The elucidation we put forward in this paper is to design concurrent priority queues based on the highly distributed Threaded Modified SkipList(TMSL) data structures of sartaj et. al [2]. Surprisingly, Modified SkipLists have received little attention in the parallel computing world, in spite of their highly decentralized nature.

Modified skip list(MSL) is a search structure in which each node has one data field and three pointer fields :left, right, and down. Each level l chain worked solely as doubly linked list. The down field of level l node x points to the leftmost node in the level l-1 chain that has key value larger than the key in x. H and T respectively , point to the head and tail of the level lcurrent chain. Underneath figure 1 shows the MSL.

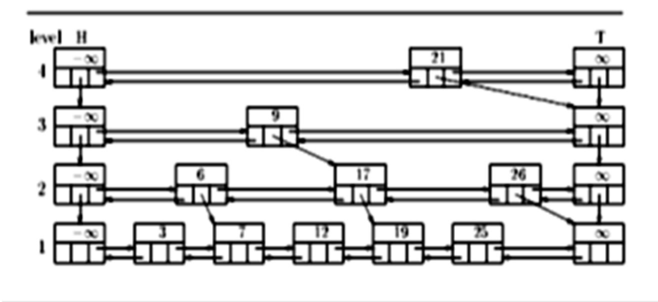

Fig. 1 Four level modified skiplist

In case of MSL the minimum element is the first one in one of the lcurrent chains. By using an additional pointer filed in each node, we can thread the elements in an MSL into a chain. The elements appear in non-decending order on this chain. The subsequent threaded structure is referred to as TMSL (threaded modified skip list). When a TMSL is habituated, the delete min operation can be accomplished in O (1) expected time.

In this paper we familiarize the lock-free access of threaded modified skip list(TMSL) in a concurrent environment. In order to provide concurrent access to MSL, a elementary adaptation is done in the structure of sartaj[2] , there is no insistence of down pointer for connecting one level to another level. There will be a pointer which works as a junction for threaded chain of MSL.

IV. ALGORITHM

By virtue of concurrent traversal of nodes they will be frequently allocated and reclaimed. We consider several aspects of memory management like no node should be reclaimed and then later re-allocated while some other process is traversing this node. This can be done with the help of reference counting. We have selected to use the lock-free memory management scheme invented by Valois [23] and corrected by Michael and Scott [24], which makes use of the FAA,TAS and CAS atomic synchronization primitives. The operations done by these primitives given underneath in figure 2, 3, 4 and 5.

```
 //Global variables
   Node *head, *tail
// Local variables
   Node *node2
```

Fig. 2 Variables Used

```
Structure Node
{
key : integer
value : pointer to word
next,prev : pointer to Node
thread_ptr: pointer to node
           }
```

Fig. 3 Node Structure

```
procedure FAA (address: pointer to
word, number: integer)
atomic do
*address := *address + number;
```

Fig. 4 FAA Atomic primitive

```
function CAS (address: pointer to word, oldvalue: word,
new value: word):boolean
atomic do
if *address = old value then
*address: = new value;
return true;
else
return  false;
```

Fig. 5 CAS Atomic primitive

For doing insertion (or delete min) of a node from the TMSL we need to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. This can be possible if we have additional information on each node about its insertion (or deletion) status. This additional information will guide the concurrent processes that might traverse into one partial deleted or inserted node. After changing all necessary next pointers, the node is fully deleted or inserted. One problem, that arises with non-blocking implementations of priority queue with TMSL that are based on the linked-list structure, is when inserting a new node into the list,because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted then the new node will be deleted as well, as illustrated in Figure 6. This problem can be resolved with the latest method introduced by Harris [25] is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, thereof ore bits 0 and 1 of the address are always set to zero. Any concurrent insert operation will then be notified about the deletion, when its CAS operation will fail.

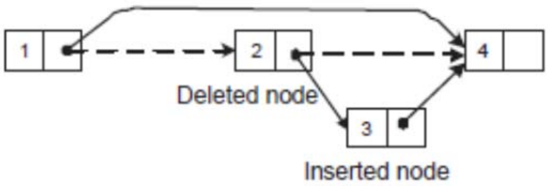

Fig. 6 Concurrent insert and delete operation can delete both nodes.

One memory management issue is how to de-reference pointers safely. If we simply de-reference the pointer, it might be that the corresponding node has been reclaimed before we could access it. It can also be that bit 0 of the pointer was set, thus marking that the node is deleted, and therefore the pointer is not valid. The following functions are defined for safe handling of the memory management: shown in figure 7

```
function READ_NODE (node  **address):
/* De-reference the pointer and increase the reference counter
for the corresponding node. In case the pointer is marked,
NULL is returned */
```

```
procedure RELEASE_NODE(node: pointer to Node)
/* Decrement the reference counter on the corresponding given
node. If the reference count reaches zero, then call
RELEASE_NODE on the nodes that this node has owned
pointers to, then reclaim the node */
```

```
function COPY_NODE(node: pointer to Node):pointer
to Node /* Increase the reference counter for the  corresponding
given node */
```

Fig. 7 Memory management function

The detailed code for insertion and deletion operations appears in underneath subsection:

A. *Insertion*

Subsequently randomly picking a level for the node, a processor searches for whether to create a new level or to insert this new node in the existing levels. The main step of inserting a new node in TMSL is to fix the position of newly inserted node depends on the value of randomLevel function. I) Atomically update left and right pointer of newly inserted node II) update the next pointer of the to-be-previous node and III) atomically update the prev pointer of the to-be-next node.IV) to connect the newly inserted node with threaded chain by updating the thread pointer For doing III step of insertion process update_insert procedure is used and for IVth step is done by update_thread function.

Algorithm: insertion of node in concurrent TMSL

```
function insert_node(key  int , value: pointer to word)
{
node *p,*t,*save[max],*t_right,*up,*found_node
k=randomlevel ()
if(k>current_level)
current_level=current_level+1
temp=current_level
x=create_node (key, value)
COPY_NODE(x)
node1=COPY_NODE(head)
If(k>temp)   // the generated level is more than the existing level
{
        //create new head and tail
        h1=createnode(∞,∞)
        copy_node(h1)
        h1→left=null
        h1→right=x
        RELEASE_NODE(H1)
        t1=CreateNode(∞,∞)
        COPY_NODE (t1)
        t1→left=x
        t1→right=NULL
        RELEASE_NODE (t1)
        x→left=h1
        x→right=t1
        RELEASE_NODE(t1)
        RELEASE_NODE(h1)
}
else  //the generated level is in between the existing levels
{
        level=head_ptr[k]   // head_ptr is array of pointer
                              storing address of head
                              for each level
        while(level→key<x→key)
        level=level→right
        prev=READ_NODE(&level→left)
        next=READ_NODE(&level→left→right)
        while T do
        if prev→right!= <next,F>
        RELEASE_NODE(next)
        next=READ_NODE (&prev→right)
        continue
        x→left=prev
        x→right=next
        if CAS(&prev→right,<next,F>,<x,F>)
        COPY_NODE(x)
        break
        back-off
        update_insert(x,next)
        }
        Update_thread(thread_ptr,x)
  }
```

Algorithm: update the left field of to be next node in concurrent TMSL

```
Procedure update_insert(x,next:pointer to node)
        While  T do
        link1=next→left
        if IS_MARKED (link1) ||  x→right!=<next,F>)
         break
        if CAS(&next→left ,link1 , <x,F>)
        COPY_NODE(x)
        RELEASE_NODE(link→p)
        if IS_MARKED(x→left)
        prev2=COPY_NODE(x)
        prev2=update_prev(prev2,next)
        RELEASE_NODE(prev2)
        break
        back-off
        RELEASE_NODE(next)
        RELEASE_NODE(x)
```

Algorithm: update the thread filed of newly inserted node and next to new node in concurent TMSL

```
procedure update_thread(thread_head,x)
{
    temp=COPY_NODE(thread_head)
    if(temp→key>x→key)
    {
    x→th_ptr=temp
    thread_ptr=COPY_NODE(x)
    RELEASE_NODE (x)
    return
    }
    else
    {
    while(temp!=NULL || temp→key< x→key)
    {
    save=temp
    temp=temp→th_ptr
    }
    x→th_ptr=temp
    save→th_ptr=x
    }
    return
}
```

B. Deletion

The delete_min operation starts from thread_heads node and find the first node (del_node) in TMSL that does not have deletion mark. Once the deletion mark is se,. the next step is to call the help_delete function to write the valid pointer on the right pointer of the previous node of the to-be-deleted node in TMSL chain. The update_prev function will update the left pointer of the right node of the to-be-deleted node in MSL chain. Once the node is deleted from TMSL chain the next step is to update the thread_head, which points the next of del_node. The algorithm has been designed for pre-emptive as well as fully concurrent systems. In order to achieve the lock free property (that at least one thread is doing progress) on pre-emptive systems, whenever a node with deletion mark is set is found, it calls the help_delete operation. The help_ delete operation, tries to set the deletion mark of the prev pointer and then atomically update the next pointer of the previous node of the to-be-deleted node. This operation might execute concurrently with the corresponding delete_min operation, and therefore both operations synchronize with each other. node of node it is updated to be the next node. The update_prev sub-function, tries to update the prev pointer of a node and then return a reference to a possibly direct previous node,

Because the help_delete and update_prev are habitually used in the algorithm for helping late operations that might influence otherwise stop progress of other concurrent operations. The algorithm is seemly for pre-emptive as well as fully concurrent systems. In fully concurrent systems though, the helping approach as well as heavy assertion on atomic primitives can relegate the performance significantly. Therefore after a number of consecutive failed CAS operations in an algorithm, puts the current operations into back-off mode , the thread does nothing for a while,and in this way steer disturbing the concurrent operations that might diversely progress slower. The duration of the back-off is initialized to some value (e.g. proportional to the number of threads) at the start of an operation, and for each consecutive entering of the back-off mode during one operation invocation,the duration of the back-off is changed using some scheme.

Algorithm: deletion of node from TMSL

```
delete_min(thread_ptr **node)
{
    prev=COPY_NODE(thread_head)
    if  (del_node==NULL) then
    RELEASE_NODE(del_node)
    RELEASE_NODE(del_node)
    return null
    i=1
    while T do
       del_node=READ_NODE(&prev→right)
    While(I<=current_level)
    {
    if(head[i]→next==del_node)
    {
    chain_head=head[i]
    break
    }
    else
    i++
    }

    link1=del_node→right
    if  IS_MARKED(link1) then
    help_del(del_node)
    continue
```

```
if  CAS(  &del_node→right,link1<link1.p,T>)
then
help_del(del_node)
next=READ_NODE(&del_node→right)
prev2=COPY_NODE(chain_head)
prev2=update_prev(prev2,next)
RELEASE_NODE(prev2)
RELEASE_NODE(next)
link2=READ_NODE(del_node→thread_ptr)
thread_head=COPY_NODE(link2)
continue
break
RELEASE_NODE(del_node)
RELEASE_NODE(link2)
back-off
return
}
```

ALGORITHM Mark previous

```
procedure mark_prev(pointer to node node)
while T do
 link1=node→left
 if IS_MARKED(link1) OR
CAS(&node→left,link1,<link1.p,T>
 break
```

Algorithm Help delete for deletion of already marked

```
pointer to node  function Help_Del(node: pointer
to Node)
 Mark_Prev(node)
last=NULL
prev= READ_NODE(&node→left)
next= READ_NODE (&node→right)
 while T do
 if prev == next  then
break
 if IS_MARKED(next→right) then
mark_prev(next)
 Next2= READ_NODE (&next→right)
 RELEASE_NODE(next)
next=next2
 continue
 prev2= READ_NODE (&prev→right)
 if prev2 = NULL then
 if last != NULL then
 MarkPrev(prev)
next2= READ_NODE (&prev→right)
if CAS(&last→right,<prev,F>),<next2,F>)
 RELEASE_NODE(prev)
 else
 RELEASE_NODE(next2)
 RELEASE_NODE(prev)
prev=last
 last=NULL
 else
prev2=READ_NODE(&prev→left)
 RELEASE_NODE(prev)
prev=prev2
 continue
 if prev2 != node then
 if last !=NULL   then
RELEASE_NODE(last)
 last=prev
 prev=prev2
 continue
RELEASE_NODE(prev2)
 if CAS(&lprev→right, <node,F>,<next,F>)
 COPY_NODE(next)
 RELEASE_NODE(node)
 break
 back-Off
 if last != NULL then RELEASE_NODE(last)
 RELEASE_NODE(left)
RELEASE_NODE (next)
```

ALGORITHM Update the previous node

```
function update_prev(prev,nodex: pointer to
Node): pointer to Node
 last=NULL
 while T do
prev2:=READ_NODE(&prev→right)
 if prev2 = NULL
 if last != NULL
mark_prev(prev)
 next2:=READ_NODE(&prev→right)
 if CAS(&last→right,<prev,F>,<next2,F>)
RELEASE_NODE (prev)
else
 RELEASE_NODE (next2)
 RELEASE_NODE (prev)
 prev=last
 last=NULL
 else
 prev2=READ_NODE(&prev→left)
 RELEASE_NODE (prev)
prev=prev2
 continue
 link1=node→left
 if IS_MARKED(link1)
RELEASE_NODE (prev2)
 break
 if prev2!= node
 if last!= NULL
 RELEASE_NODE (last)
 last=prev
 prev:=prev2
 continue
 RELEASE_NODE (prev2)
 if link1→p = prev
break
 if    (prev→right    =    node)    &&
CAS(&node→left,link1,<prev,F>)
COPY_NODE(prev)
 RELEASE_NODE (link1→p)
 if IS_MARKED(prev→left)
 break
 back-Off
if last != NULL
RELEASE_NODE (last)
 return prev
```

V. CORRECTNESS

In this section we describe the correctness of presented algorithm .here we outline a proof of linearizabilityM. Herlihy et. al [26] and then we prove that algorithm is lock-free. Few definitions are required before giving proof of correctness.

Definition 1: We denote with Mt the abstract internal state of a threaded modified skip list as priority queue at the time t. Mt is viewed as a set of of values (p,w) consisting of a unique priority p and a corresponding value w.The operations that can be performed on the structure are Insert (I) and Delete_min(DM). The time t1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t2 is defined as the time just after the atomic execution of the same operation. The return value of true2 is returned by an Insert operation that has succeeded to update an existing node, the return value of true is returned by an Insert operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is M_1 : O_1; M_2, where M_1 is the conditional state before the operation O_1, and M_2 is the resulting state after performing the corresponding operation:

$$[p_1,\_\,] \in M_{t1} : I_1([p_1,w_1]) = \text{TRUE},$$

$$M_{t2} = M_{t1} \cup \{ [p_1 , w_1] \} \quad (1)$$

$$[p_1,w_{11}] \in M_{t1} : I_1([p_1,w_{12}]) = \text{TRUE}_2$$

$$M_{t2} = M_{t1} \setminus \{[p_1 , w_{11}]\} \cup \{[p_1 , w_{12}]\} \quad (2)$$

$$[p_1 ,w_1] = \min \{ [\min p , w] \mid [p , w] \in Mt_1 \}$$
$$DM_1() = [p_1 , w_1] , M_{t2} = M_{t1} \setminus \{[p_1 , w_1]\} \quad (3)$$

$$M_{t1} = : DM_1() = \text{NULL} \quad (4)$$

Definition 2: In order for an implementation of a shared concurrent data object to be linearizable [M. Herlihy et al.[1990], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.

Definition 3: The pair [p, w] is present ([p, w] ∈ M) in the abstract internal state M of implementation, when there is a connected chain of next pointers (i.e. prev→link→right) from a present node in the doubly linked list that connects to a node that contains the value w, and this node is not marked as deleted (i.e. is_marked (node) =false)).

Definition 4: The decision point of an operation is outline as the atomic statement where the consequences of the operation is finitely decided, i.e. independent of the result of any sub operations after the decision point, the operation will have the same result. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.

We will now practice these definitions to show the execution history of point where the concurrent operation occurred atomically.

LEMMA 1 :AN insert_node operation which flourish (I [p, w]) = true), takes effect atomically at one statement.

PROOF: the decision point for an insert operation which succeeds (I [p , w] = true) when the CAS sub-operation CAS(&prev→right,<next, F>,<x, F>) of insert operation succeeds, and the insert operation will finally true. The atate of the list (Mt1) directly before passing of the decision point must have been [p, _] € Mt1, otherwise the CAS whould have failed . The state of the list directly after passing the decisison point will be [p , w] ∈ Lt2.

LEMMA 2 : A Delete_Min operation which get ahead (D() = [p , w]), takes effect atomically at one statement.

PROOF: the verdict point for an delete_min operation which sccceeds (DM ( ) = [p , w]) is when the CAS sub operation CAS (&del_node →right,link1 < link1.p,T>) flourish. The state of the list (M_t) directly before passing of the decision point must have been [p ,w] ∈ Mt , otherwisw the case would have failed . the state of the list after passing the decision point will be [p ,_] € L_t

LEMMA 3 : A delete_node operation which fails (DM() =NULL),takes effect atomically at one statement

PROOF The decision point for a delete operation which fails (DM() =NULL) is the check in line if (del_node==NULL) . state of the list (M_t) directly before the passing of the state-read point must have been $M_t = \emptyset$.

LEMMA 4: With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.

PROOF: Here we examine the possible execution paths of our implementation of TMSL. There are numerous conceivably unbounded loops that can stalling the termination of the operations. We call these loops retry-loops. The retry-loops take place when sub-operations search-out that a shared variable has changed value. This is observed either by a subsequent read sub-operation or a failed CAS. These shared variables are only adapted concurrently by other CAS sub-operations. According to the explanation of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding Insert or delete_min operation will progress. Thus, the one operation will always progress independent of any number of concurrent operations.

VI. CONCLUSIONS

Here we make known to concurrent threaded modified Skiplist using a remarkably simple algorithm in a lock free environment. Our enactment is raw, various optimization to our algorithm are possible like we can extend the correctness proof. Empirical study of our new algorithm on two different multiprocessor platforms is a pending work. The presented algorithm is first step to lock free algorithmic implementation of priority queue with modified skip list; it uses a fully described lock free memory management scheme. The atomic primitives used in our algorithm are available in modern computer system.

REFERENCES

[1] W.Pugh. June 1990.Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM 33 .

[2] S. Cho and S. Sahni1998. Weight-biased leftist trees and modified skip lists. ACM J. Exp. Algorithmics,

[3] R. Ayani. Lr-algorithm: concurrent operations on priority queues. In Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing pp. 22-25, 1991.

[4] J. Biswas and J.C. Browne. Simultaneous Update of Priority Structures In Proceedings of the 1987 International Conference on Parallel Processing, August 1987, pp. 124{131.

[5] Sajal K. Das, Maria Cristina Pinotti, F. Sarkar. Distributed Priority Queues on Hypercube Architectures. In International Conference on Distributed Computing Systems (ICDCS) 1996:620-628

[6] N. Deo and S. Prasad. Parallel Heap: An Optimal Parallel Priority Queue. In The Journal of Supercomputing, Vol. 6, pp. 87-98, 1992

[7] Q. Huang. An Evaluation of Concurrent Priority Queue Algorithms. Technical Report, Massachusetts Institute of Technology, MIT-LCS/MIT/LCS/TR-497, May 1991.

[8] G.C. Hunt, M.M. Michael, S. Parthasarathy and M.L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. In Information Processing Letters, 60(3):151{157, November 1996.

[9] Carlo Luchetti and M. Cristina Pinotti. Some comments on building heaps in parallel. In Information Processing Letters, 47(3):145-148, 14 September 1993

[10] B. Mans. Portable Distributed Priority Queues with MPI. In Concurrency: Practice and Experience, 10(3):175-198, March 1998.

[11] Optimal Parallel Initialization Algorithms for a Class of Priority Queues. In IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 4, October 1991.

[12] Sushil K. Prasad and Sagar I. Sawant. Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors. In Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP 95), 1995.

[13] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and Locality in Priority Queues.In IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, October 1994

[14] V. N. Rao and V. Kumar. Concurrent access of priority queues. IEEE Transactions on Computers 37, 1657-1665, December 1988.

[15] P. Sanders. Fast priority queues for parallel branch-and-bound. In Workshop on Algorithms for Irregularly Structured Problems, number 980 in LNCS, pages 379-393, Lyon, 1995. Springer.

[16] P. Sanders. Randomized Priority Queues for Fast Parallel Access. In Journal of Parallel and Distributed Computing, 49(1), 86 - 97, 1998.

[17] Y. Yan and X. Zhang. Lock Bypassing: An Efficient Algorithm for Concurrently Accessing Priority Heaps. ACM Journal of Experimental Algorithmics, vol. 3, 1998.http://www.jea.acm.org/1998/YanLock/

[18] J. Boyar, R. Fagerberg and K.S. Larsen. Chromatic Priority Queues. Technical Report, Department of Mathematics and Computer Science, Odense University, PP-1994-15, May 1994.

[19] T. Johnson. A Highly Concurrent Priority Queue Based on the B-link Tree. Technical Report,University of Florida, 91-007. August 1991.

[20] N. Shavit and A. Zemach. Concurrent Priority Queue Algorithms. In Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, pages 113-122, Atlanta, GA, May 1999.

[21] LOTAN, N. SHAVIT. Skiplist-Based Concurrent Priority Queues. International Parallel and Distributed Processing Symposium, 2000.

[22] H. Sundell and P.Tsigas.2003. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In Proceedings of the 17th International Parallel and Distributed Processing Symposium, page 11. IEEE press.

[23] J. D. Valois.1995. Lock-Free Data Structures. PhD. Thesis, Rensselaer Polytechnic Institute, Troy, New York.

[24] M. Michael, M. Scott.1995. Correction of a Memory Management Method for Lock-Free Data Structures. Computer Science Dept., University of Rochester.

[25] T. L. Harris. Oct. 2001.A Pragmatic Implementation of Non-Blocking Linked Lists. Proceedings of the 15th International Symposium of Distributed Computing,

[26] M. Herlihy and J. Wing.1990. “Linearizability: a correctness condition for concurrent objects,” ACM Transactions on Programming Languages and Systems,vol. 12, no. 3, pp. 463–492.