delim $$
gsize 11
gfont R

Parallel Queues and Pools, An Evaluation

by

David Wood

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #150
January 1989

Revised : April 1991

This work was supported under National Science Foundation grant No. DCR-8413359.

* The initial version of this paper was accepted as my Master’s Thesis

Introduction

In this paper I examine a broad range of parallel queue and pool (unordered queue) algorithms and present data indicating their relative strengths and weaknesses. The algorithms were implemented on the NYU Ultracomputer prototype, an eight processor MIMD, shared memory, single bus machine. The data are normalized to remove the effects of bus contention. I discuss the implementation of a few basic synchronization primitives and the implementation of each tested algorithm. Two programs were written, one, *cqtst*, provided a generalized arena for timing the different algorithms under different circumstances. The second program, *fifo*, tested the parallel fifo-ness of the queues.

1. The NYU Ultracomputer

The Ultracomputer is an eight processor, multiple instruction-multiple data (MIMD), shared memory machine. Each processing element (PE) is connected to 8MB of shared memory through a single 32-bit bus. Each PE consists of a Motorola 68010 microprocessor with 32KB of private cache memory. The PEs use software long integer multiply and divide. A PDP 11/34 performs I/O for the Ultracomputer.

The basic primitive fetch-and-add, which I discuss in Section 3., is supported in hardware at the shared memory end of the bus. All other fetch-and-phi operations (store and or) are implemented in software through the use of fetch-and-add.

Dynamic shared memory allocation by spawned processes is not supported in the current system. In those algorithms that required such an allocation, I simulated it by allocating enough memory before spawning, which means that the total memory size was larger than it had to be.

For more information on the Ultracomputer, see references [7] and [8]. Reference [7] in particular provides a thorough overview of software and hardware considerations concerning the Ultracomputer.

2. What Is a Parallel Queue?

Queues and pools, which are basic elements in any multi-programming system, are used in many applications. Because they constitute an important and heavily used data structure in operating systems, should be relatively lightweight (in memory) and as fast as possible. There is, of course, a tradeoff between memory and speed. In a serial environment, speed is measured strictly by the number of machine instructions executed on each insert and delete (this discussion assumes the data structure is neither full nor empty). In a parallel environment, the added time is due to synchronization and/or contention. For example, a linked list is a fast serial algorithm, but when synchronization is added to accommodate parallelism, linked lists become a relatively poor queue because each process must serialize to gain access to the list. Thus an algorithm is considered highly parallel if the time it takes n processes to insert or delete in parallel is the same within a constant as the time it takes one process to insert or delete serially.

The term **queue** implies that the an order of the items in the queue corresponds to the order in which each was inserted. What does it mean to be ordered when many processes are inserting items onto the queue concurrently? To discuss parallel queues, then, we must first define what it means to be ordered. Gottlieb, Lubachevsky, and Rudolf (GLR) in [5] defined the characteristics of a parallel queue as follows,

> *If insertion of data item p is completed before insertion of another data item q is started, then it must not be possible for a deletion yielding q to complete before a deletion yielding p has started.*

This implies that once an item is on (allocated a position in) the queue, it must be assigned to a delete process before any later items in the queue are assigned a delete process. This does, however, allow for later items in the queue to be returned earlier than previous. Owing to scheduling, the process assigned to delete q may actually be completed before the process assigned to delete p.

3. Some Synchronization Routines

GLR discuss the various synchronization primitives used in parallel algorithms, the most basic of which is the atomic operation fetch-and-add, also referred to as *faa*. The result of *faa*(i,a) is to increment i by a and return the original value of i. This primitive implements the test-modify-retest paradigms *tdr* (test decrement retest) and *tir* (test increment retest) in which a variable is tested to see if it will exceed some bound when either decremented or incremented. A bound of zero is assumed for *tdr*. Their implementations are as follows:

```
boolean function tir(s,delta,bound)
        tir = false
        if (s+delta <= bound) then
                if (faa(s,delta) <= bound)
                        tir = true
                else
                        faa(s,-delta)
end tir
```

```
boolean function tdr(s,delta)
        tdr = false
        if (s-delta >= 0) then
                if (faa(s,delta) >= 0) then
                        tdr = true
                else
                        faa(s,-delta)
end tdr
```

Race conditions and starvation, which are present in this implementation, are discussed in GLR. *Tdr* and *tir* will be used extensively to see if the queues/pools are empty or full, and can therefore be deleted from or inserted onto.

The most important use of the generalized counting semaphores *P* and *V* is to obtain exclusive access to a data structure, such as the linked list. Each semaphore is implemented as an integer initialized to a value that depends on the algorithm. The following are the implementations of *P* and *V*:

```
procedure P(semaphore,delta)
        repeat until tdr(semaphore,delta)
end P
```

```
procedure V(semaphore,delta)
        faa(semaphore,-delta)
end V
```

In the linked list example, the integer would be given the initial value one. Access to the list would be obtained using *P*(*semaphore*,1) and released with *V*(*semaphore*,1).

The remaining three synchronization primitives are more involved and will be discussed only to describe their semantics.

The first, the group lock synchronization algorithm developed by Dimitrovsky in [6], has the following set of routines:

grp_init(lock)	- initialize a group lock
grp_lock(lock)/grp_unlock(lock)	- enter/leave a group
grp_sync(lock)	- synchronize the group

The above routines provide for the arbitrary grouping of processes at a boundary (*grp_lock* or *grp_sync*). *Grp_lock* acts like a waiting room in which processes wait (busy-wait) for a door to open at which point they all go through. The door is then closed and cannot open again until those processes have exited through *grp_unlock*. *Grp_sync* is used between the *grp_lock* and *grp_unlock* calls to resynchronize the grouped processes.

The next synchronization primitive is the reader-writer lock. In this case a lock is associated with a data structure that can be read by any number of processes when no process is writing (or waiting to write) to the structure. At any time any process (including a reader) may become a writer; at which time all requests for read locks are denied access and forced to busy-wait until the writer has released its lock. The new writer may not proceed until all current readers have released their locks. Only one writer exists at any given time. A naive use of a reader-writer lock in queues would be to avoid integer overflow resulting from *faa* on position counters in the queue. When the counter reaches some critical value, a reader lock upgrades to a writer lock, which waits for all current readers to finish and then modifies the counter. Note that the decrement need not be done with a *faa* because all other modifiers of the counter have been locked out.

The set of required reader-writer routines is as follows:

rw_initlock(lock)	- initialize a reader-writer lock
rw_rlock(lock)/rw_runlock(lock)	- obtain/release a reader-writer read lock
rw_wlock(lock)/rw_wunlock(lock)	- obtain/release a reader-writer write lock
read2write(lock)	- change from a read lock to a write lock
write2read(lock)	- change from a write lock to a read lock

Their implementations can be found in GLR, [10], [1], and [2].

The last synchronization primitive to be discussed is the reader-reader lock in which there are two groups of processes, A and B, that can not access a data structure at the same time. That is, the data structure will not be accessed by processes from both groups concurrently; either group A or group B can have access but not both. One group arbitrarily, is given priority, which can lead to starvation. A reader-reader lock is useful in dynamic hash queues in which the size of the hash table is allowed to contract and expand. Because one would not want both an expansion and a contraction taking place simultaneously, a reader-reader lock is used in which group A is the expanders and group B, the contractors.

The set of required reader-reader routines is as follows:

rr_initlock(lock)	- initialize a reader-reader lock
rr_Alock(lock)/rr_Aunlock(lock)	- obtain/release a reader-reader A lock
rr_Block(lock)/rr_Bunlock(lock)	- obtain/release a reader-reader B lock

Their implementations can be found in [9], [1] and [2].

4. Queue/Pool Overview

The next sections describe the tested algorithms. Treating some of the older ones less thoroughly, I go into more depth with the newer ones. There are three classes of algorithms. The singleton queues are **fifo** queues that do not support multiple-item insertion. The second type, multi-queues, do support multi-items. The third type, which is not a queue but a grouping of items, I refer to as a pool. All algorithms use busy-waiting synchronization primitives as opposed to the nonbusy-waiting (or blocking) type. Pseudocode for the algorithms can be found in Appendix A.

I define the following variables to quantify the various algorithms' performance:

m - number of queues
M - maximum number of queues that can be supported
n - number of items that are currently on all m queues (not counting multiplicities)
N - maximum number of items that can be supported on one queue
p - amount of parallelism to be supported

I use these quantities to define each algorithm's memory and time requirements (see Table 1). Where a quantity is not defined above, see the description of the algorithm. The time complexities in the table represent best/average case times, with the exception of those for *dhqueue*. I assume that a queue/pool is not full when inserting or empty when deleting and that no process has to wait for an item to be inserted or deleted (i.e., assume no cell contention).

Queue	Memory	Time	
		Insert	Delete
llist	$O(m+n)$	$O(1)$	$O(1)$
queue	$O(p\times m+n)$	$O(1)$	$O(1)$
hqueue[1]	$O(M+m+n+hsize)$	$O(1)$	$O(1+n/hsize)$
dhqueue[2]	$O(M+m+n+hsize(n))$	$O(2+2\times MaxLF)$	$O(2+3\times MinLF/2)$
fqueue	$O(m\times n)$	$O(1)$	$O(1)$
vqueue	$O(m\times n)$	$O(1)$	$O(1)$
mqueue[3]	$O(p\times m + n)$	$O(1)$	$O(1)$
tqueue[4]	$O(n+m\times(\log_k n+1))$	$O(1+\log_k n)$	$O(1+\log_k n)$
set	$O(m\times n)$	$O(1)$	$O(1)$
cset	$O(m\times n/wordsize)$	$O(1)$	$O(1)$
pool	$O(p\times m+n)$	$O(1)$	$O(1)$

1 hsize is the size of the hash table.
2 Time complexities shown are worst case. $n/MaxLF \le hsize \le n/MinLF$
3 Performance depends upon the multiplicity (c) of the items. When $p<c$, *time dependency approaches O(1), but in the worst case where* $c = 1$, *the time dependency is O(p) on average. This is because the processes must serialize to obtain successive elements.*
4 The number of children of each node in the tree is variable at compile time. For the standard tests $k=6$.

Table 1.

In general, each queue/pool maintains a *qsize* counter to indicate the number of items currently on the queue. On insertion, either *tir*(*qsize*, *nitems*, *maxqsize*) or *faa*(*qsize*, *nitems*) is performed. The *tir* is used when the queue is of fixed size, and if it fails, insertion fails. Except in the case of multi-queues, the value of *nitems* will be one. At the beginning of each deletion, each queue or pool uses a *tdr*(*qsize*,1) to see if there are any items available. If the *tdr* fails, deletion fails.

Each algorithm uses *head* and *tail* counters as indexes into the queue/pool indicating where to delete from or insert onto. In most cases, these counters will overflow if some action is not taken, although in hash queues, for example, this is not a problem. Overflow only causes a discontinuity in the indexes; because some queues/pools are more sensitive to this discontinuity than others, these may require an adjustment of *head* and *tail* as the data structures age. For algorithms that can not handle overflow in general, check each *faa* of *head* or *tail* to see if the value has reached some predetermined *bound*. If so, decrement the counter by *bound* using *faa*.

All the queues/pools, with the exception of the fixedsized singleton queues/pools, have the ability to prematurely remove an item from the data structure regardless of its position. In all cases this involves leaving what is called a "hole," a temporary placeholder that when encountered will be deleted from the structure but not be returned as a deleted item. Rather, deletion continues through the data structure until the first real item is found. One must leave this hole in most cases because each item, once it is put on the queue, is assumed to be there and thus associated with a given position in the queue/pool. A placeholder must therefore be left in place of removed items. I have not thoroughly tested the various methods used for removal (*cqtst* does provide the capability), but I mention this aspect in order to be complete.

The pseudo-code in Appendix A gives the reader the basic concept of each algorithm. I do not show the full details because it might make the code less readable and therefore less useful. As a result, I have left out overflow handling and support for interior removal. Except in the case of *mqueue*, for which the extra code is included, the code should not require added synchronization to support interior removal. The code as presented is a good representation of what was tested.

4.1. Singleton Queues

4.1.1. A Linked List (llist)

A single linked list, as mentioned earlier, is not a useful queuing algorithm in a highly parallel environment. Because distributed link lists are the basis of the algorithms of *queue*, *hqueue*, *dhqueue* and *mqueue*, I will mention its implementation here. This will also be a useful a reference when comparing other algorithms' performance.

Through the use of *P*/*V*, a binary lock (semaphore) is held over the list so that only one process can access the items from the list at one time. This serializes all processes accessing the list and thus will not support highly parallel programming.

4.1.2. A Serial Linked List (sllist)

I added serial linked list to *cqtst* strictly as a reference. The code is identical to that of *llist* except that the synchronization primitives were removed, providing us with a feel for the relative speeds of the other algorithms and possibly an idea of how much synchronization costs.

4.1.3. A Circular Array Queue (queue)

Queue implements a GLR-type queue of unlimited size supporting interior removal. A circular array of linked lists holds the queue items. The array size is user definable but should be larger than the maximum amount of parallelism. A *faa* is performed on a counter, *head* (for delete) or *tail* (for insert), that indexes the array of lists. To order gets and puts, thereby keeping the lists ordered, two tickets are associated with each list. When a ticket indicates that a process is allowed to proceed, a binary lock is obtained on the list and the item is either deleted or inserted. Because items are placed at the end of each list and because of the circular nature of the array, this queue can be thought of as a spiral. Items are inserted at the outside and eventually spiral down to the center, where they are deleted.

The time complexity of this queue is O(1) for both deletion and insertion. It takes constant time to find the correct list, and with the correct pointers to the beginning and end of each list, list manipulation is also constant time. The memory complexity is $O(m*p + n)$.

Within an operating system the number of queues, m, will most likely rise at least linearly with p, the amount of available parallelism, meaning that the memory requirements grow quadratically with p. This important result should be paid close attention to, as some of the other algorithms have more amenable memory requirements.

4.1.4. A Group Lock Queue (fqueue)

Fqueue is a fixed sized group lock based queue in which user items are copied into queue storage. Because items are copied into the queue, memory must be preallocated for the maximum possible number of items (N) on the queue. Interior removal is not supported.

Insertion and deletion are two-phase operations. Inserts are active during the first phase, whereas deletes are active in the second phase. The algorithm allows both deletes and inserts to enter in the same group but serializes the inserts before the deletes within a group.

Head and *tail* counters are maintained in a manner similar to *queue*. A circular array stores the user items, with a *size* counter indicating the available space.

4.1.5. A Baker's Algorithm Queue (vqueue)

Vqueue is a fixed sized baker's algorithm-based queue in which similar to *fqueue*, values are copied into queue storage. As with *fqueue*, interior removal is not supported, and memory must be preallocated for the maximum possible number of items on the queue.

Inserts acquire a ticket number from *tail*, and deletes acquire a ticket number from *head*. The ticket number then orders the processes so that an insert follows a delete that follows an insert that follows a delete and so on. A single ticket associated with each cell in the queue indicates whose turn it is, either an insert or a delete.

Head and *tail* counters are maintained as usual and a circular array stores the user items. The algorithm requires two counters associated with the number of items on the queue. *Filled* and *empty* are the number of cells in the queue that contain values and those that do not, respectively. They determine whether or not an insert or delete will initially fail. The number of items on the queue at any one time is the value of *filled*.

This two-counter scheme is necessary in order to avoid performing a concurrent *tdr* and *tir* on the same counter. This could (and did) lead to deadlock. If n+1 inserts begin and compete for n cells in the queue, as expected, one insert will fail. Now if the failing insert is context switched out before it can increment the count, then there will appear to be n+1 items on the queue, allowing n+1 deletes to begin, one of which will endlessly wait for an item. (Note the pseudo-code in Appendix A, the ticket scheme assumes a queue of at least size two.

4.1.6. A Hash Table Queue (hqueue/dhqueue)

The *hqueue* and *dhqueue* hash queue algorithms are very similar to *queue* except that they can support many queues on one hash table (circular array). Like *queue*, *hqueue* and *dhqueue* allow for an unlimited size queue and interior removal, but differ from *queue* in that each item on each queue is tagged with a position in the given queue. This avoids the need to keep the lists ordered but requires deletes to search the list for the correct item. *Dhqueue* differs from *hqueue* in that it's hash table can expand and contract. Because these are new algorithms (*hqueue* is attributed to Isaac Dimitrovski in reference [1]), I discuss them in more detail.

Each queue in a hash table is given a unique identifier mapped from the range 1..M+1, with M the maximum number of queues that may use the table at once. The extra value identifies an internal queue of independent queue items that may be exchanged for a removed item. The queue identifiers in the hash function distinguish similar items in different queues and keep queues from tracking one another. Tracking occurs if the ith element of queue q1 and the j element of queue q2 fall in the same list, and if the (i+1)th element of q1 and (j+1)th element of queue q2 also fall in the same list. The generalized hash function used is as follows:

hashval = (qid * pos) **mod** hsize,

where *pos* is obtained from a *faa* on either *head* or *tail*, accordingly. The hash function for *dhqueue*, discussed in Section 4.1.6.2, differs slightly but uses a similar concept to avoid tracking.

Along with the notion of queue identifiers (ids) comes the need to track the available ids. In other algorithms, reusing a queue means just re-initialize it, but with these two algorithms the queue ids can only be made available again through a separate function called *dispose*. Disposing of a queue involves putting the id back in some kind of set associated with the hash table. The current algorithm uses the *cset* algorithm, which is discussed in Section 4.3.3. Queue initialization, then, involves labeling the queue with an id from the set.

Interior removal is handled in a unique way. A separate internal queue maintains items that are used to pass back as copies of the removed items. Upon queue initialization the user indicates the number of removals (holes) the queue will support. Only one internal queue supports removal for of all the queues, however; as a result a queue may get more or less removals than were requested at initialization.

4.1.6.1. Hqueue

One critical performance difference between *hqueue* and *dhqueue* is that an *hqueue* deletion searches for the desired item in the list outside of the critical *P*/*V* (list manipulation) section. In contrast to *dhqueue*, then, it is more efficient. Hash table initialization requires the table's size (i.e. fixed size table).

Because the hash function involves determining a value modulo the table size, table size should be given as prime.

Memory requirements are O(M+m+n+$hsize$), with M the maximum number of queues allowed in the hash table, *hsize* the size of the hash table, n the number of items in the hash table and m the number of queues using the hash table. The complexity is linear in the number of queues and processors, assuming M=c*p. Insertion is O(1) time, but deletion becomes O(1+n/$hsize$) time because each operation requires searching a list for an item from the designated queue.

4.1.6.2. Dhqueue

As mentioned above, *dhqueue*'s, hash table can grow and shrink as the load on the table changes. The algorithm is a merging of the Dimitrovsky hash queue above and a serial dynamic hash table algorithm that is discussed in reference [4]. The serial dynamic algorithm is based on a linear expansion method in which each list in the table divides or merges once per cycle. A cycle is the number of splits or merges required to double or halve the table size. Each list in the table splits or merges once per cycle. A counter p, which indicates the next list to be split, is initialized to zero and incremented with each split. When a list splits, a new list with *index*=p+*maxp* becomes a buddy. *Maxp* is the current maximum value that p can obtain before it is reset to zero and the table begins its next phase of doubling. When p is decremented to zero, divide *maxp* by two and set p to the resulting value. When p is incremented to *maxp*, p resets to zero and the value of *maxp* doubles. A compile time constant, a power of 2, is chosen as a minimum table size, so that the value of *maxp* is always a power of 2.

Memory, a group of list pointers designated as a segment, is allocated in pieces that double in size, up to some maximum, with each allocation. Although this is not the only available strategy, it seems reasonable always to allocate space in proportion to the current size of the table.

Like *hqueue*, the hash function uses the queue identifier to eliminate tracking but also depends on the values of p and *maxp* as shown below:

tmp = (qid * pos)
hashval = tmp **mod** maxp
if (hashval < p) hashval = tmp **mod** (2*maxp)

If the first value of *hashval* is less than p, the corresponding list has split, in which case, the queue item may belong in one of two lists: the first value of *hashval* or its buddy at *hashval*+*maxp*.

The value of *maxp* is a power of 2, which makes for a convenient modulo operation but generally provides for a horrendous hash function. If *qid* is even, *hashval* will always be even and every other list will go unused. If *qid* is odd, however, because the values of *pos* are sequential, *tmp* will be alternately odd and even, meaning that *hashval* will exhibit the same behavior. Each odd value of *qid* smaller than the smallest value of *maxp* (minimum table size) results in a different sequence of hash values over the range (0..*maxp*-1). This provides a well distributed hash function without any one-to-one tracking.

Another behavior I call cyclic tracking results in the odd qids 3 and 9, for example. With a tablesize greater than 8, the frequency of such tracking is only 3, that is, every third element of queue 3 collides with every element of queue 9. Another problem with the odd queue identifier scheme if one-to-one tracking is to be avoided, is that it limits the number of queues on a hash table. It seems plausible to make the minimum table size depend on the expected maximum number of queues, although that was not implemented.

The synchronization on an insert and delete is similar to that of *hqueue* with two exceptions. Between the time the list for the *n*th item is determined and the time the lock on that list is obtained, the list may have split or merged, which could result in the item not being found on a delete or an item being misplaced on an insert. The solution for inserts is to check that the list number (*hashval*) has not changed once the lock is obtained; if it has, release the lock and rehash. The solution for deletes, mentioned earlier, is to search the list once per lock on a list. If the item is not found, release the lock and rehash and obtain a new lock, repeating until the item is found.

To simplify the discussion of expansion/contraction I refer for the most part to table expansion; the extension to contraction is a relatively clear one. When an expansion is initiated, a read lock is obtained. A *faa*(*p*,1) is performed, and if a process, call it A, returns *maxp*-1, the lock is upgraded to a write lock. After obtaining the write lock, process A then decrements *p* by *maxp* and doubles the value of *maxp*. Processes entering after A that obtain *p*>=*maxp*, release their read locks, wait for *maxp* to be reset and get a new read lock and a new value of *p*. This synchronization assures that one full cycle of expansion completes (all lists in 0..*maxp*-1 are split) before the next cycle can begin.

Each process now has a unique value of *p* and *p'*=*p*+*maxp*, where *p* is the number of the list to be split and *p'* is the number of the list to split items into. After locking list *p*, each item is rehashed; if it should be moved it is put on a linked list external to the queue. The lock on list *p* is released, and the resulting external list is then put into list *p'* after obtaining a lock there.

To exclude expansions from being concurrent with contractions, a reader-reader lock is used around the calls to each routine. This ensures that while a list is being split it is not also being merged. More important, this ensures that *p* will be either increasing or decreasing during a table size change ensuring that the hash function be reliable.

The memory requirements of this queue are O(*M*+*m*+*n*). In Table 1, I used two quantities, the minimum (*MinLF*) and maximum (*MaxLF*) load factors and which represent the shallowest and deepest average list size. These two constants determine how often contraction and expansion take place. The best case time complexity for insert is O(1) and for delete O(1+*MinLF*/2). The worst case is when the table size must be adjusted. The resulting complexity is O(2+2**MaxLF*) for insert and O(2+3**MinLF*/2) for delete. If a doubly rather than a singly linked list were used, these times could have been reduced to O(2+*MaxLF*) and O(2+*MinLF*/2). Each adjustment requires a scan of two lists of length *MaxLF* or *MinLF*. A full scan is required on a merge to find the end of the parent list. The extra half for a delete is the usual average scan length to find the desired item. If *MinLF* is chosen to be less than 0.67**MaxLF*, the worst case will be for the insert. In the tests, *MinLF*=8 and *MaxLF*=16.

4.2. Multi-Queues

4.2.1. A Circular Array Multi-queue (mqueue)

Mqueue is an algorithm similar to *queue* that incorporates the storage of a multiple item into a single queue item. Items are stored on the familiar distributed array of linked lists. Inserts are similar to that of *queue*, except that when an insert notices it is inserting the first item on the queue, it sets the global queue variable *firstmult* which indicates the multiplicity of the first item. Deletes initially perform the usual *tdr*. After acquiring an item, the delete decrements the multiplicity of the first item; if it goes to zero *firstmult* and *head* are updated. Note that, whereas *queue* requires ordering the deletes *mqueue* does not because of the enforced serialization at the head of the list.

Some extra synchronization is needed to support interior removal. Because items can be removed, the next item on the queue may not be in the next list. For this reason a search is made of the lists until one is found in which the first item is present. I have included the extra code needed to support the interior removal, namely, the reader-reader locks.

4.2.2. A Tree Structured Multi-queue (tqueue)

Tqueue is a k-ary tree-based multi-queue supporting interior removal attributed to Isaac Dimitrovski in reference [2]. This queue provides a fixed O(log*N*) access time and has a more amenable memory requirement than *mqueue*. When *m* queues (*m* different trees) that can each hold a maximum of *N* items are associated with a common pool of memory, the memory complexity has been shown to be $O(m*(1+\log N+N))$.

Each queue is a distinct k-ary tree in which items with unit multiplicity are stored at the leaves of the tree. Items with a multiplicity greater than one are stored in a combination of internal nodes and leaves. An internal node can hold as many items as there are leaves in its subtree.

As in previously discussed algorithms, a *head* and *tail* counter are used to get the position of the item on the tree during a delete or an insert. If I call the depth of the tree *d*, then the position can be represented as follows:

$$\mathrm{pos} = a_{d-1}*k^{d-1} + \ldots + a_0*k^0.$$

To insert or delete an item, follow edge a_i at level *i* in the tree.

Some synchronization and queue adjustment are required to properly function. A reader lock is used around both insert and delete operations, with the delete upgrading to a write lock when it is necessary to adjust the queue. When traversing a path to a position in the tree, the leftmost child of any subtree allocates the root of the subtree. All the other children of that subtree must wait (busy wait) for it to be allocated.

When an internal node is allocated, it is given a use count equal to twice the number of leaves in its subtree. Each time a process traverses a node, the use count is decremented, on both inserts and deletes, thus the factor of 2 above. When a node's use count is decremented to zero, the node can be taken off the tree and returned to the pool of free nodes.

The queue header contains two pointers each of which can point to a tree that can support the maximum number of items, *N*, allowed on the queue. As a result, the queue can hold 2**N* items if there are enough free nodes, the first tree will have to be emptied, however, before any items are re-inserted into that tree. At the end of deletion and when the delete counter *head* equals *N*, the read lock is upgraded to a write lock, *tail* and *head* are decremented, and the second tree moves to become the first tree of the queue.

4.3. Pools/Unordered Lists

4.3.1. A Circular Array Pool (pool)

Pool is very similar to *queue* except that no effort is made to order the linked lists. Like *queue*, *pool* supports interior removal and is of unlimited size. After performing a *faa* to get a list number, an item is either inserted or deleted from the head of the list. Each list, if not for racing at the entrance to the list, would be a LIFO stack.

4.3.2. A Set (set)

Set is a fetch-and-store (*fas*) based algorithm developed by Jan Edler in [3] to provide operations on a fixed size group of integers. Similar to *faa*, *fas*(*a*,*i*) is an atomic operation that stores the value of *i* in *a* and returns the old value of *a*. An array of integers is initialized to the user values, with *head*, *tail*, and *qsize* counters maintained to access the array. Because the user can store any integers (i.e., pointers), this algorithm has all the power of *pool*.

Referring to the pseudo-code in Appendix A, inserts perform a *fas* with the value to be inserted. When zero is returned, the value has been successfully stored. Note that when the loop is performed more than once, the *fas* is returning a value from another process. Similarly, deletes perform *fas* until a value is returned.

4.3.3. A Compact Set (cset)

Cset, also developed in [3], is a bitmapped version of *set* using both *fas* and fetch-and-or, *faor*. *Faor*(*a*,*i*) performs a bitwise logical OR of *a* and *i* and returns the original value of *a*. The algorithm uses both a bitmap and an array of integers indicating words in the bitmap that contain nonzero bits. *Head* and *tail* counters are maintained over the latter array and obtain indexes into the bitmap. By incrementing *head* and *tail* on each bit map access, different processes acquire different words in the bit map and thereby avoid serialization. Serialization can not be avoided when there are more processes than words in the set. After obtaining the correct word in the bitmap, the *faor* is performed on the desired word with the correct bit turned on for an insert or off for a delete. A delete chooses an arbitrary bit in the bitfield to turn off and use as its return value. When a bit is the first/last to be turned on/off, the array of integers is updated using *fas* to add/remove a value from the array as is done in *set*.

5. Cqtst

To test and develop the algorithms, I wrote a general utility that accommodates the following,

1) Multiple queue testing
2) Variable queue size (per test)
3) Multiple copy insertion
4) Interior removal
5) Variable number of spawned processes
6) Queue 'exercising' (multi-process delete/re-insert)
7) Timing with *cqtst* overhead removal

Cqtst, which was written with more capabilities than were utilized in the tests done for this paper, is intended as a general tool, to develop and test new and existing algorithms. Because all the facilities are useful to this end, I will describe them all. For a full list of optional arguments and their functions, see Appendix B.

In addition to the capabilities listed above, *cqtst* has the ability to perform parallel/serial loading or emptying in any combination. Each task, loading or emptying the queues, can be done either in serial or in parallel. When parallel loading and emptying is selected, loading and emptying take place at the same time. This means, however, that one can not parallel load and then parallel empty.

Cqtst is a versatile tool with which to test algorithms in many different ways. The data presented in this paper are drawn from a subset of tests which *cqtst* can perform.

As of this writing there are twelve different queue/pool algorithms incorporated into *cqtst*:

1) llist	2) sllist	3) queue	4) fqueue
5) vqueue	6) hqueue	7) dhqueue	8) tqueue
9) mqueue	10) set	11) cset	12) pool

The items inserted on the queue will either be pointers or values depending on the queue being used. I used a shared memory array of integers as the inserted items. Retrieval is done into a shared memory sink array using a fetch-and-add on the index. After the queuing and dequeuing is finished, the sink is sorted and checked against the source array and multiplicity information. The sink is not sorted when a serial load and serial empty are performed or when queue exercising and preremoval are not done. By not performing the sort for serial-serial testing, an allowance is made to check for the serial FIFO capabilities of the queue. No test is made for parallel FIFO performance. See the fifo test program (*fifo*) for parallel FIFO testing.

5.1. Multiple Queue Testing (-m)

During a run of *cqtst* more than one queue, all the same type, can be operated on concurrently. No item will be inserted into more than one queue. This capability is not allowed when using *sets* or *csets*. *Cqtst* is written so that queues are loaded and emptied in order and each process operates on each queue. Each process does not wait for the queue it is currently working on to be filled/emptied however, before it goes on to the next queue.

If I call *numinserts* the number of items to be inserted into the queues taking into account multiple copies, the maximum number of items on a single queue at one time will be *numinserts/m*, where *m* is the number of queues. Space is allocated for the situation in which each queue may contain *numinserts* items, but the total number of items on all queues will not exceed *numinserts* (i.e., queues are only 1/*m* full). This exploits any increase in performance that the more memory efficient queues may have. See Table I for a list of memory requirements.

5.2. Variable Queue Size (-n)

Here, the number of distinct elements to be supported by the queues can be manipulated. This capability is necessary, because some algorithms' insert/delete times vary depending on the maximum number of items allowed on the queue.

5.3. Multiple Copy Insertion (-c)

This allows a distinct element to occur in the same queue more than once. The user specifies a maximum multiplicity, *c*, then the multiplicity of each element is randomly chosen (before queuing begins) from 1 to *c*, inclusive. Multiple insertion is meaningful only for the multi-queues *tqueue* and *mqueue*.

5.4. Interior Removal (-r)

This aspect allows an element to be removed from a queue prematurely. Whether an item is to be removed or not is a random decision (made before queuing) that results in about 20 percent of the items being marked for removal. Currently, there are two methods by which removal can be done,

1 - During the emptying phase, instead of doing a FIFO empty, perform a remove. Each process is assigned a subset of the items and sequentially runs through the subset removing the item if it was marked for removal; otherwise a delete is performed. This does not guarantee holes because another process may have already deleted the item to be removed. Because some items will be removed, this option allows for the parallel testing of item removal.

2 - Perform all removals before doing any emptying. This requires that loading be done serially, which is done whenever this method is chosen. With this mechanism, one is able to examine the

relative slowdown of an algorithms' retrieval time when removals are done.

After a successful removal, A fetch-and-add on a counter is performed, and the total reported in the output page.

Removal should not be used when items are given a multiplicity greater than one and when a multi-queue, such as *mqueue*, is being tested for which a single removal involves deleting every occurrence of the item on the queue. *Tqueue* does not exhibit this behavior and may be tested in the manner described above.

5.5. Variable Number of Spawned Processes (-s)

The user may specify the number of processes, *p*, in the test. These *p* child processes evenly divided among the loading and emptying tasks. When loading and emptying are done in parallel, there will be *p*/2 processes/task; otherwise all *p* processes will be devoted to the single parallel task, loading or emptying.

P may also be the number of processes involved in performing queue ‘‘exercising.’’ Because serial loading and emptying is forced for exercising, there is no conflict with the alternate definition and all *p* processes perform exercising.

5.6. Queue ‘‘exercising’’ (-p)

This simulates an environment in which the queues/pools are being heavily used by as many processes as the user desires. Queues/pools are first loaded up in serial. Then a user chosen number of processes are spawned (-s flag), which then delete from and re-insert onto the queue/pool a user-specified number of times. After all the processes have finished, the queues/pools are emptied serially. When exercising in parallel, each item may not be deleted and inserted the same amount of times, nor will each item remain in the same position with respect to other items. This is non-deterministic with the caveat that when exercising is over, the queues will still hold the same items. This will, however, destroy the original ordering of the items and so the sink array must be sorted to check for queuing errors.

Queue exercising also provides for testing queue readjustment. Most queues use *head* and *tail* counters to keep track of where items should be inserted and deleted from. As the queue experiences more inserts and deletes, the *head* and *tail* will grow without bound unless some kind of readjustment of these values and possibly the queue itself was performed. Because exercising can be done in parallel or serial, adjustments can be tested in either environment.

Exercising does not work when a multiplicity greater than one is used on the two multi-queues. The *mqueue* algorithm assumes that the user puts one item with a given multiplicity on the queue once and that the item's multiplicity is decremented on each delete. It further assumes that the item just retrieved will not be placed back on the queue. Because this is exactly what queue exercising is, exercising will not work for *mqueue* when the multiplicities are greater than one. With *tqueue*, memory is allocated for *n* distinct items (not $n*c$, *c* is the average multiplicity of items in the queue) that may or may not have a multiplicity. Each time an occurrence of an item with a multiplicity greater than one is deleted and re-inserted, a new distinct item is created, leading to the need for more memory than was originally allocated, causing a failure.

5.7. Timing (-t, -b, -p)

At the end of each run a single page of status information is produced that contains, among other things, information about the amount of wall clock time required to perform initialization, loading, exercising and emptying. There are three times available, user, system and total for each of the four tasks (initialization, loading, exercising, emptying), all in milliseconds. When parallel emptying and loading, an average time for loading and emptying is reported.

Although the option to not subtract the overhead of *cqtst* (see the -b option) is available, a normal run of *cqtst* provides this subtraction. This is accomplished by running through the same operations as in the specified test but not actually calling any queue routines. The amount of time to do this is accumulated and subtracted off the time obtained when the queues are tested.

Inherent in the timings statistics is a variance that the user should be aware of. To reduce this variance which is present in all timing values, the -t option is provided, to allow the user to perform the same operations many times (queue initialization is done only once), and thereby reduce the variance. With the stipulation that, based on the timing techniques, the standard deviations will be no larger than 23 ms/$\sqrt{\text{cycle}}$, the user can select the number of cycles needed to obtain the desired accuracy.

5.8. Output

The printout shown below represents a default run of *cqtst*. The following command produced the second printout.

```
%cqtst -p1000 -t4 -s4 -q3 -h213 -r2 -ec
```

Each printout completely specifies the input options (e.g., queue type, number of items on queue, number of cycles, number of processes...). Notice that serial loading was forced for the second printout because removal method two was chosen. Once again, all times are in milliseconds.

When performing queue exercising with the -p option, the amount of time required to do a single delete/re-insert operation is reported in milliseconds per operation normalized to a single processor. If there are *p* processes and it takes time *t* (wall clock time) for each process to complete *N* delete/re-insert operations, then I define wall clock delete/re-insert time as $T_w = t/N$. Normalization to a single process is done by dividing the wall clock time by the number of processes.

In addition to the timing statistics, information about the amount of swaps that took place during testing is available to allow the user to verify that results were not invalidated by interference from the system swapper.

Queue type used : fqueues
Memory : O(n) = 1024

1024 elements were distributed among 1 queue(s)

Multiplicity : 1 Total insertions : 1024/cycle

Number of insert/delete operations per process per queue
(excluding load/empty) during exercising : 0
Number of load/exercise/empty cycles : 1

Loading : parallel with 4 children
Emptying : parallel with 4 children

The following stats have been adjusted for overhead

Wall Clock Time	init	load	exercise	empty	total
user (ms/cycle)	0	230	0	230	460
system (ms/cycle)	0	19	0	19	39
total (ms/cycle)	0	250	0	250	500
Voluntary Swaps	0	0	0	0	0
Involuntary Swaps	0	0	0	0	0

----- No queuing errors detected -----
----- No queuing errors detected -----
----- No queuing errors detected -----
----- No queuing errors detected -----

Queue type used : Hash queues,   hsize = 213

Memory : worst case n = 1024

1024 elements were distributed among 1 queue(s)

Multiplicity : 1   Total insertions : 1024/cycle

Removal method 2 used, 182 elements successfully removed/cycle

Number of insert/delete operations per process per queue
(excluding load/empty) during exercising : 1000
Number of load/exercise/empty cycles : 4

Loading : serial
Exercising : parallel with 4 children
Emptying : serial

The following stats have been adjusted for overhead

User time/insert&delete operation : 0.613542 msec
System time/insert&delete operation : 0.019792 msec

Wall Clock Time	init	load	exercise	empty	total
user (ms/cycle)	0	187	613	400	1201
system (ms/cycle)	200	0	19	16	236
total (ms/cycle)	200	187	633	416	1437
Voluntary Swaps	0	0	0	0	0
Involuntary Swaps	0	0	0	0	0

6. Fifo

Fifo is a relatively short program that tests algorithms to see if they conform to the definition of parallel FIFO given earlier. *Fifo* first fills the queue halfway and then spawns an equal number of inserters and deleters. Each inserter inserts the value of a private integer that is incremented on each insert. The inserters mask the high four bits of each item with the insert's spawn index. Deleters then delete from the queue and thereby obtain items from different insert processes. Each deleter keeps a record of the maximum valued item thus far retrieved for each insert task. If at any time a deleter retrieves an item for an inserter that is lower than the current maximum for that inserter, a fifo error has occurred.

One main difference between *fifo* and *cqtst* is the duties of each spawned process. When exercising, each process of *cqtst* performs a delete followed immediately by an insert, which imposes a balancing of inserts with deletes. *Fifo*, in contrast, spawns processes that are dedicated to a single activity, inserting or deleting. This means that every process is completely autonomous with the exception of the test algorithms' synchronization. Although *fifo* thoroughly tests the queues, no attempt is made to see whether items were lost from the queues.

I do not present any hard data for *fifo*, although all the queues successfully passed this test. *Fifo* is an excellent tool for determining whether a queue is really a queue, as defined earlier.

7. Testing and Results

The following data were gathered using the exercising facility of *cqtst*. The times presented and the data extracted from them are based on the time it takes one process to do a delete followed by a re-insert of the deleted item.

Because the Ultracomputer prototype is a single shared bus machine, care must be taken when determining the relative performance of an algorithm in a parallel or even a shared environment. All data were produced in single-user mode; thus bus contention results only from deliberately concurrent processes. To eliminate the effects of bus contention due to p spawned processes during a single parallel run of *cqtst*, p separate but similar serial and concurrent runs of *cqtst* are made later. The latter test provides the serial speed of the algorithms, but with bus contention similar to that present during the single parallel run of *cqtst*. The following symbols are for the three different available values:

T_s = serial delete/re-insert rate with no bus contention (1 process)
T_{sc} = serial delete/re-insert rate with bus contention (p processes)
T_p = parallel delete/re-insert rate with inherent bus contention (p processes)

From these three values I can extract two quantities as follows:

$$\text{Efficiency} = T_{sc} / T_p, \qquad \text{Contention} = T_{sc} / T_s$$

Efficiency is a measure of the serialization that results from synchronization among sibling parallel processes. A perfect algorithm would not suffer any such serialization and thus score an efficiency of one. The less the efficiency, the worse the algorithm performs in a parallel environment.

Contention measures an algorithms shared memory use. An algorithm with no bus contention would have a value of one. The worse the performance, the larger the value of *contention*. *Contention* is a ratio of local to shared memory access and thus provides no information concerning the intensity of shared accesses.

All tests were performed on a queue/pool that could support up to 1024 elements. In the case of *queue*, *mqueue,* and *pool,* the number of lists always equals the number of processes accessing the data structure. When *hqueue* was tested, the size of the hash table was chosen to be 101, which means that on average each delete had to search a list with ten items. The default value of *k* for *tqueue* was six.

The first set of tests shows the performance of the queues and pools when the load on the system changes. What happens as the ratio of the number of PEs available for scheduling to the number of processes using a queue or pool decreases? For example, initially there may only be four processes competing for eight PEs, in which case there should not be a problem. But as the system becomes loaded with other possibly unrelated tasks, the number of effectively available PEs is reduced and thereby reducing the ratio. (I will refer to this as PE contention, not to be confused with the definition of *contention* given above.) Although in a multi-user system in general the processes will not be of the same type, I believe I effectively simulated this effect by adding more of the same type of processes (i.e., more children of *cqtst*) to the system.

Factors that may contribute to performance characteristics under these circumstances are the scheduling method and that busy-waiting and not non-busy-waiting semaphores were used. The latter is probably the most influential of the two.

Table 2 gives the serial speeds (T_s) of the different algorithms, the time it takes to load and empty the queues and the time it takes to perform a single delete/re-insert operation. Because *sllist* and *llist* differ only in a *P*,*V* reference, it is evident that the binary semaphores add as much as .04 msec per reference pair. In the case of *llist* this amounts to almost halving the speed.

Graphs 2 through 9 show the values of T_{sc}, T_p, *efficiency,* and *contention* versus the number of processes for the queues and pools. Because their performance was so poor when the number of processes exceeded the number of available PEs, data for *queue* and *mqueue* were not taken for *p* greater than 8. This phenomenon, discussed in Section 8.2, is also present in *vqueue* but to a lesser degree. The data presented for *dhqueue* do not reflect time for expansions or contractions during exercising, because not enough items are ever removed. Until the number of processes exceeds eight, the variances are around 3 percent, but as the number of processes grows, the variances grow steadily to about 25 percent.

I also took data indicating *tqueue*'s performance for different values of *k*, the number of children of each node in the tree. Graph 1 shows the data for four different values of *k*: 2, 4, 6, and 7. The data shown here are statistically very good with a maximum variance of less than 2 percent.

Serial Performance, N = 1024			
Queue	Time(ms)		
	Load	Empty	Delete/Re-insert (R_s)
sllist	61	68	.15
llist	113	105	.23
queue	416	433	.85
hqueue†	251	508	.75
dhqueue	423	736	1.11
fqueue	1591	1560	2.34
vqueue	363	376	.73
tqueue	1066	1043	2.03
mqueue	426	458	.89
set	240	261	.50
cset	273	778	1.04
pool	221	266	.48

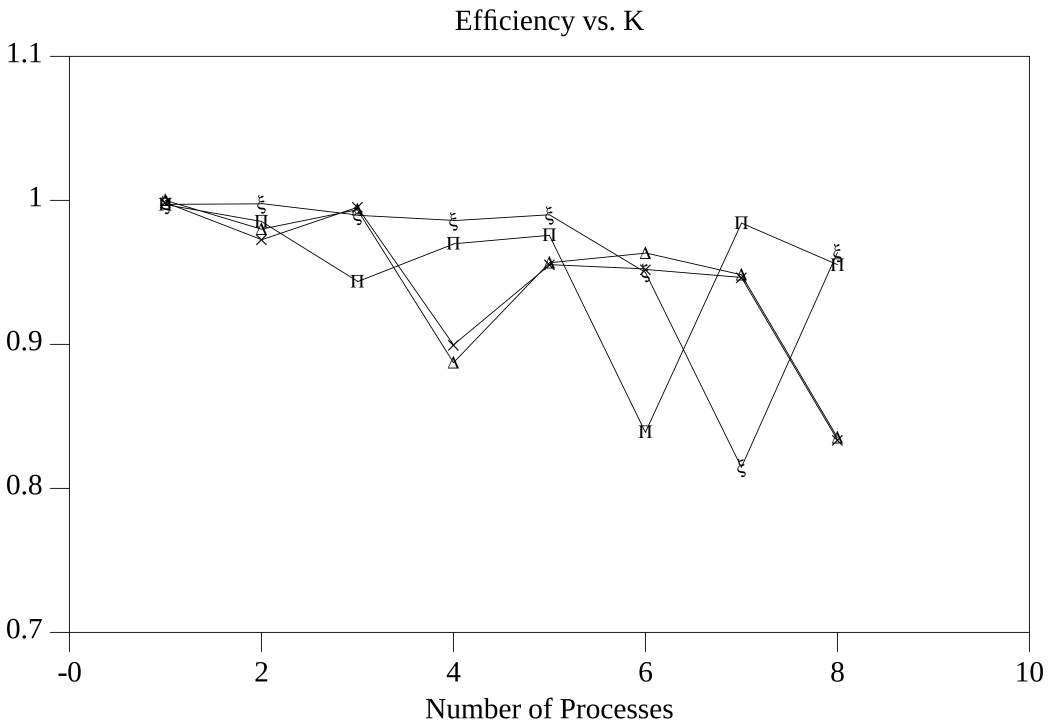
† Average bucket depth = 10.

Table 2.

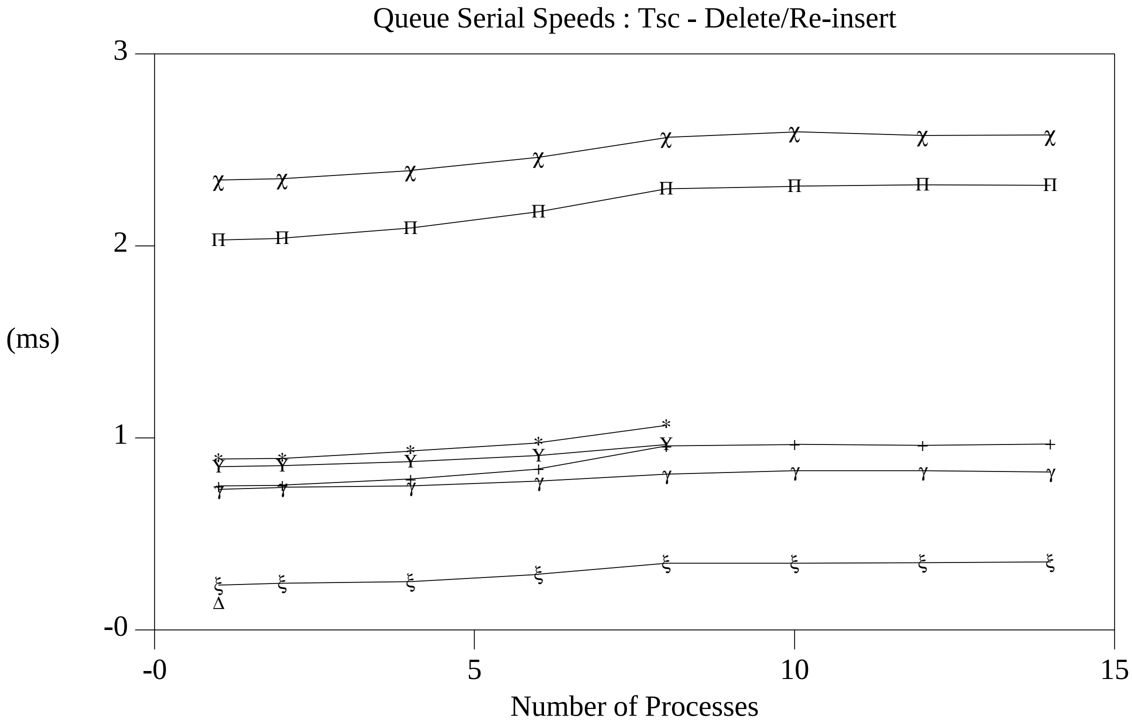
7.1. Data Plots

Legend

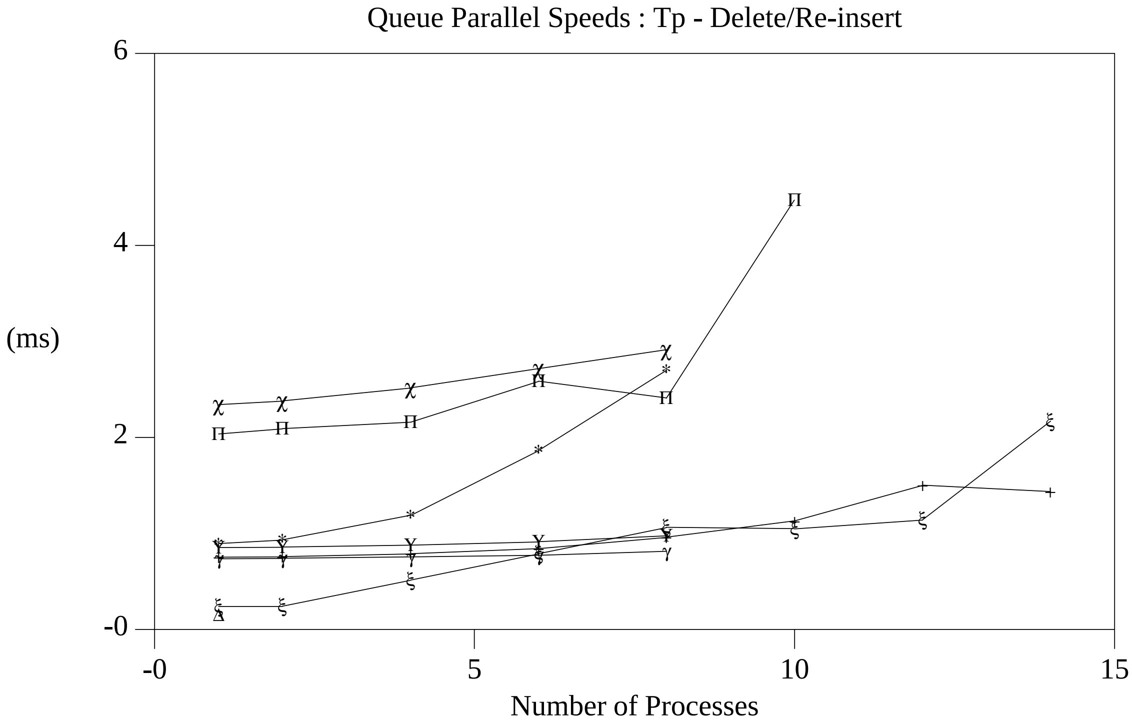
γ	vq
Δ	sll
ξ	ll
Υ	q
*	mq
+	hq
Π	tq
χ	fq
ϕ	pool
×	set
∅	set
ξ	tq, k=7
Π	tq, k=6
Δ	tq, k=4
×	tq, k=2



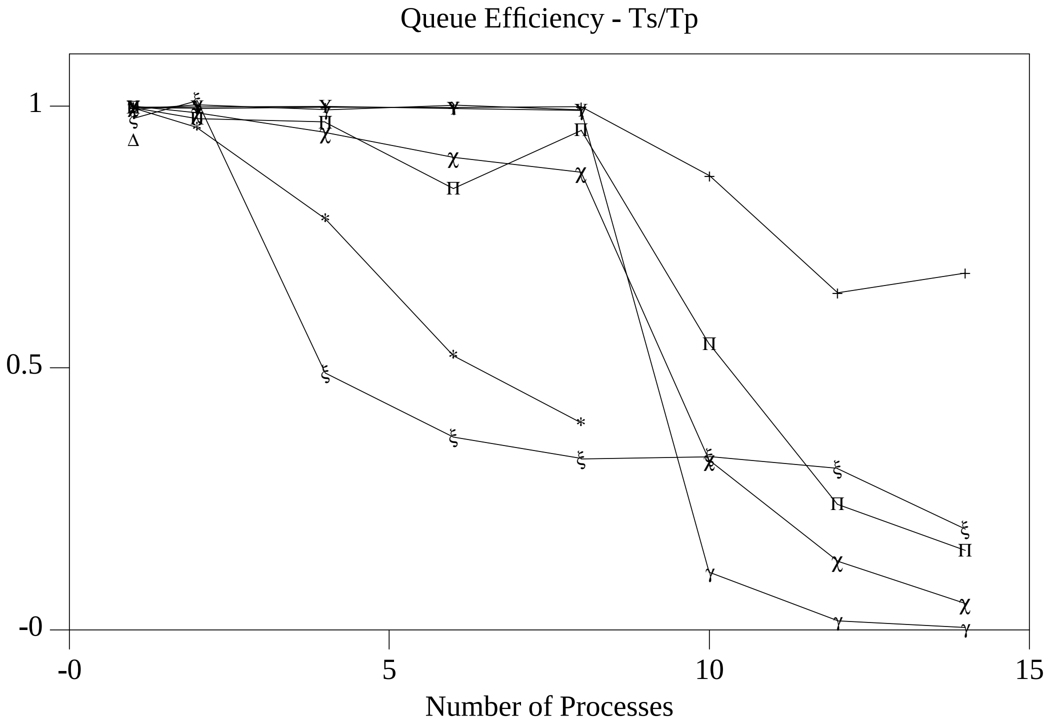
Graph 1.



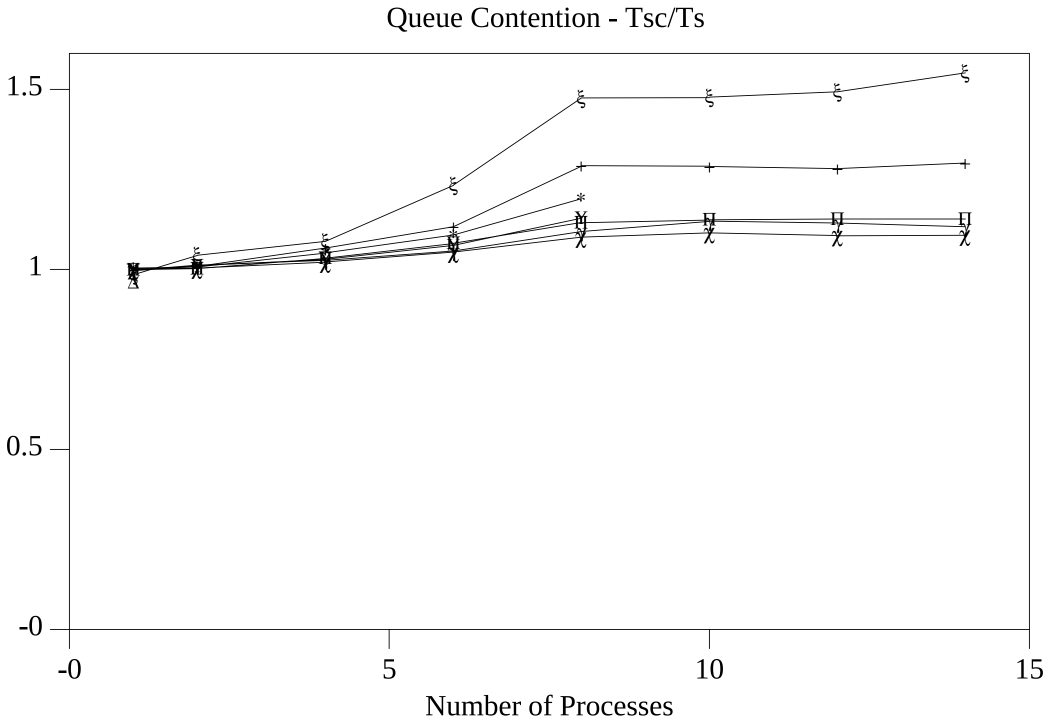
Graph 2.



Graph 3.

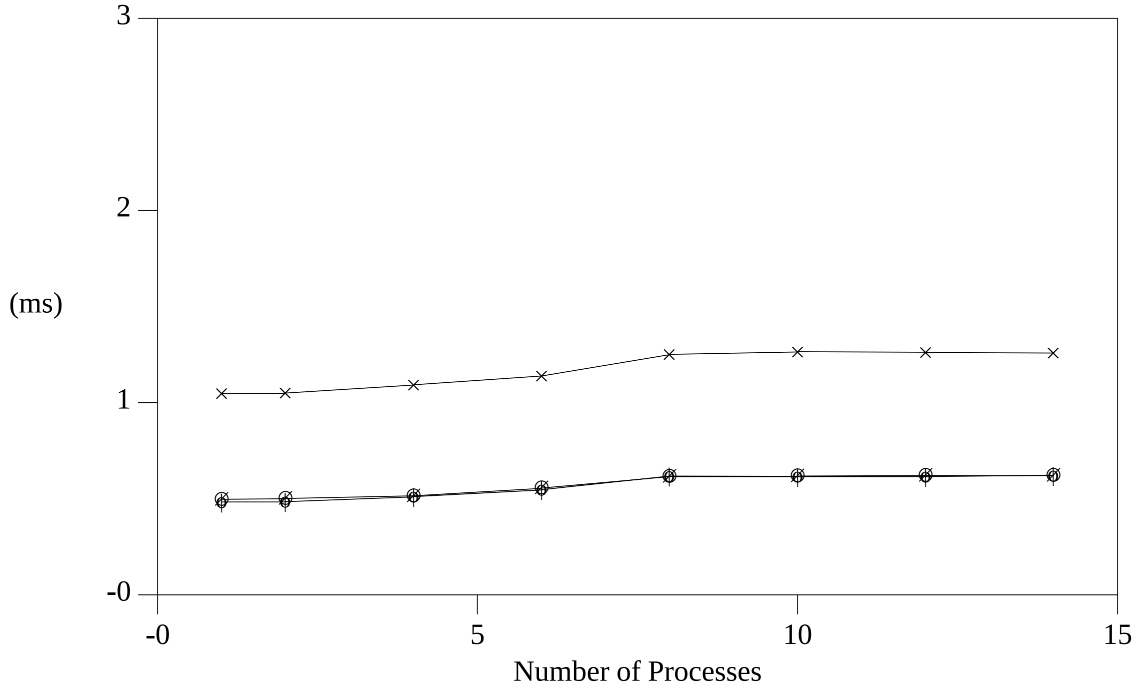


Graph 4.



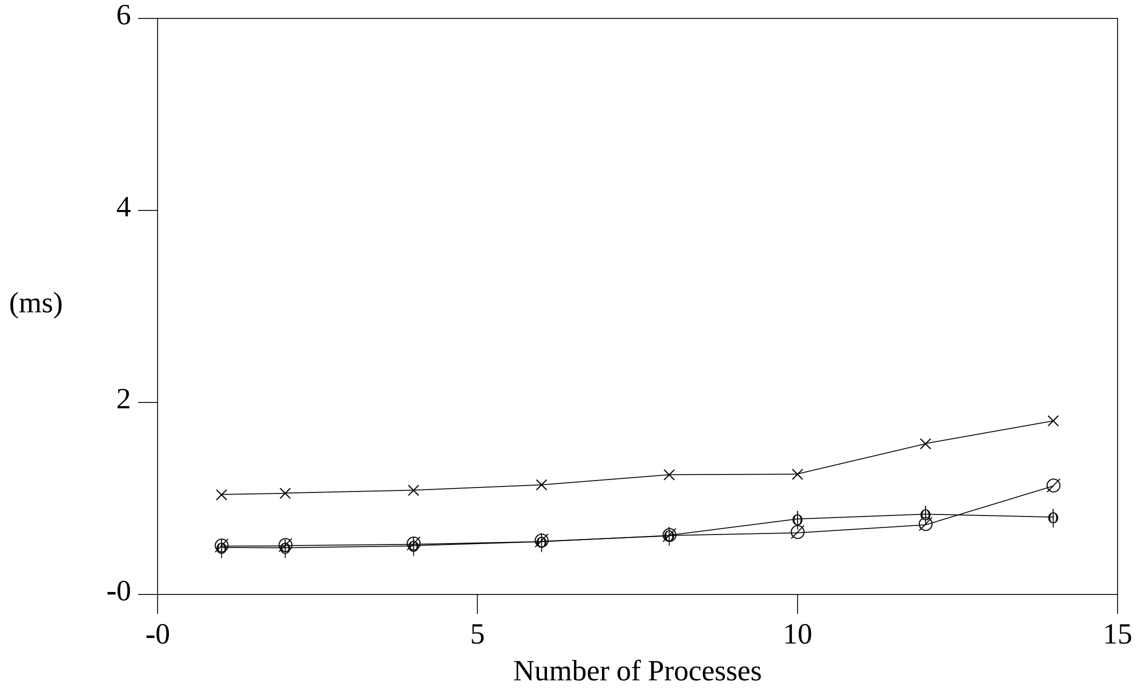
Graph 5.

Pool Serial Speeds : Tsc - Delete/Re-insert



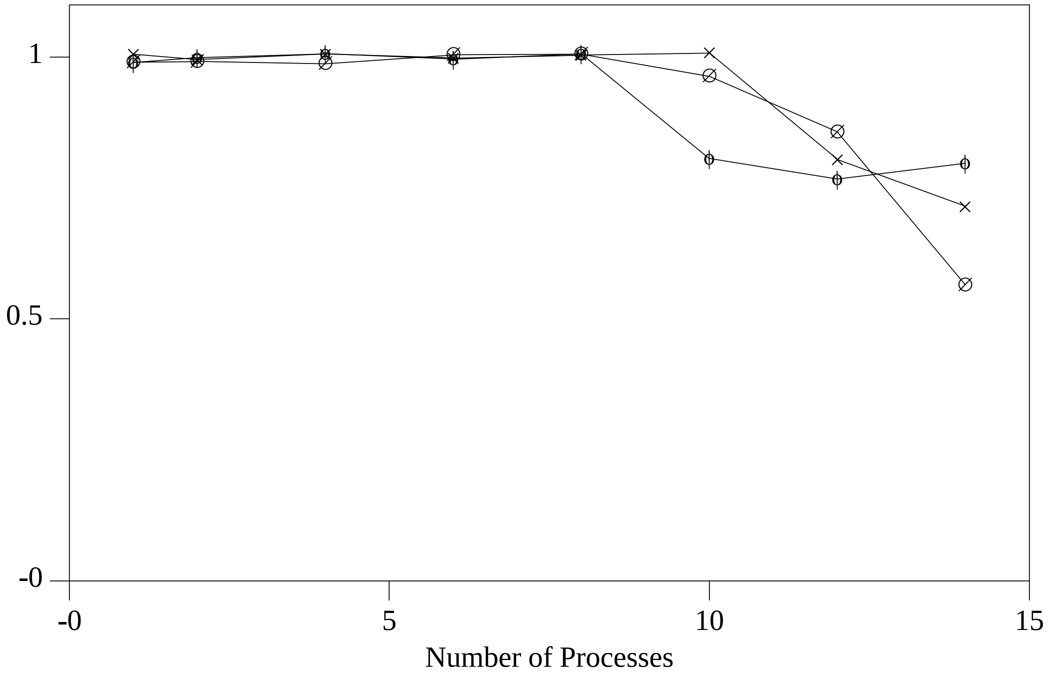
Graph 6.

Pool Parallel Speeds : Tp - Delete/Re-insert



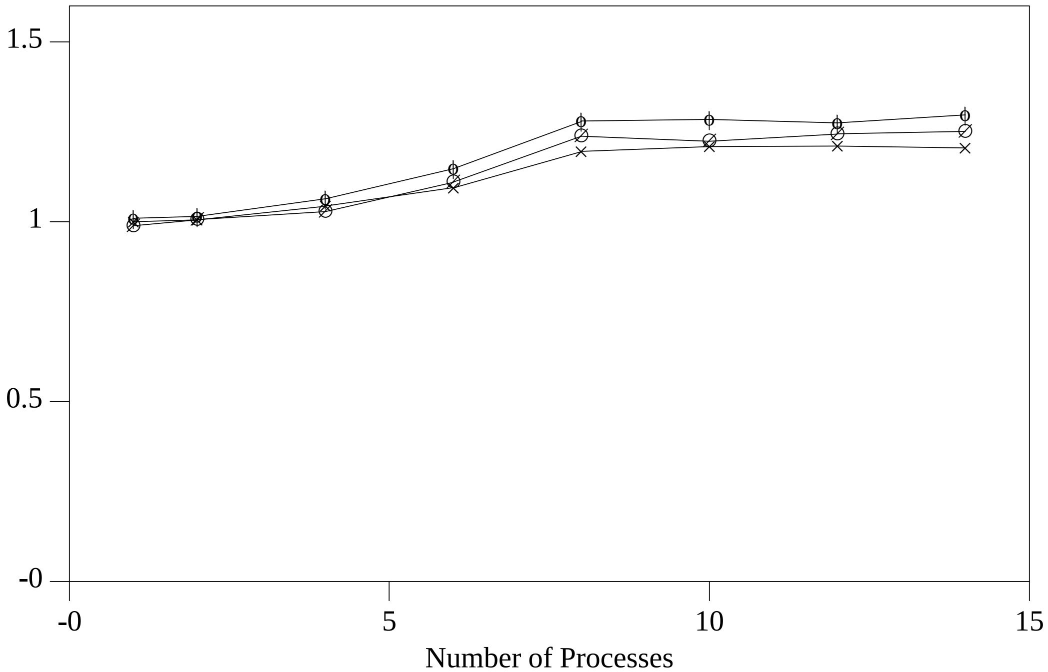
Graph 7.

Pool Efficiency - Ts/Tp



Graph 8.

Pool Contention - Tsc/Ts



Graph 9.

8. Discussion

In Table 2, the serial linked list, *sllist*, provides a reference point from which to judge the other algorithms' raw speed allowing us to neglect the absolute speed of the Ultracomputer and focus on the relative speeds of each algorithm. Neglecting the parallel linked list, *llist*, the closest speed is that of *vqueue* or *hqueue*, which are each about five times as slow. Because these two algorithms operate with near perfect efficiency until they encountered PE contention, they would need five dedicated processors to equal the performance of the serial linked list on one processor.

The number of dedicated PEs required for each algorithm to equal the performance of *sllist* on one processor is shown in Table 3.

Number of Processors to Equal Performance of Serial Linked Lists	
Queue	Required Number of Dedicated PEs Serial Speed/((*Sllist* Speed)×(Efficiency @ 8 Processes))
sllist	1.0
llist	5.1†
queue	5.7
hqueue	5.0
dhqueue	9.3†
fqueue	17.8†
vqueue	4.9
tqueue	14.1†
mqueue	15.0†
set	3.3
cset	7.0
pool	3.1

† Efficiency not constant over 1 to 8 processes

Table 3.

Using the efficiency at eight processes is not strictly valid unless the efficiency is constant over the range from one to eight. I have flagged those cases where it was not constant over the range but reported the numbers as they may be of interest. In general, it would be more accurate to determine the efficiency as a function of the number of dedicated PEs. From the data in Table 3, it is clear that an eight processor machine can outperform a similarly equipped serial machine.

It may also be important, that the Ultracomputer does not have fast divide (and modulo) or multiply hardware. which could reduce an algorithm's relative raw speed. If these slow operations take place in critical sections, then we would expect the efficiency to be less than optimal. We would expect *tqueue* to be affected the most, because divide and modulo operations are performed at each level of the tree. The hash queue algorithms may also be affected owing to an extra multiply and modulo in the hash function, although *dhqueue*'s modulo is really just a logical AND.

In the next analyses, I am for the most part interested in *efficiency*: if an algorithm is efficient it can be made to be useful by putting it on a machine with enough PEs, regardless of raw speed. Also, the plotted speeds contain bus contention, whereas this has been normalized out of the *efficiency* data.

8.1. Singleton Queues

Parallel linked lists, *llist*, provide a reference against which the other algorithms can be compared. Almost any algorithm that has better efficiency than *llist* can be considered to be somewhat parallel. Initially, with less than eight processes, all the queues perform better than *llist*, but as the processes begin to compete for PEs only *hqueue* and *dhqueue* outperform it in efficiency.

Fqueue performs well until there are more processes than PEs, which is most likely because of the context switching of critical processes. The effect is magnified through the use of *gsynch* because it forces the group to reform, which can be very expensive when there is PE contention. To alleviate this problem, it would be useful to have a premature unlock in which the processes doing the inserts (the first phase) can leave the group as soon as they finish. The size of the groups may also play an important role.

Dhqueue performed remarkably well considering the potential for extra overhead. Although it does not show even efficiency out to eight processes, it possesses a graceful decline in performance out to fourteen processes, something none of the parallel queues exhibit. This is important because operating systems that suddenly become very slow are more frustrating than ones that do so gradually. Although presently I do not understand this behavior, I plan to examine it in future work.

Overall, *hqueue* seems to be the best queue. It has a perfect efficiency out to eight processes and drops the least beyond that; it is also one of the fastest of the truly parallel queues. It would be the clear choice in any application in which the queue size does not fluctuate too much. (When the size does fluctuate, *dhqueue* may prove to be a better choice, even with the poor efficiency.)

8.2. PE Contention

As mentioned earlier, I took no data for *queue* and *mqueue* above eight processes. *Queue* and *mqueue* behave in a similar manner to that of *vqueue*, but with more severe results. Performance dwindled to the point that it would have taken days to get the rest of the data. The behavior is attributable to three things: the ticket scheme, the busy-waiting synchronizations, and the scheduling.

After a few minutes of running more processes than PEs, these three factors caused all the processes to wait on a single cell or list for their ticket to be called. This becomes fairly slow when the process whose ticket is called is currently context switched out; it becomes extremely slow when a process that is at the head of the line for the bottlenecked cell/list gets scheduled and then inserts and deletes all the way around the queue until it comes back to the original cell/list where it is then at the end of the line at the same bottleneck. This continues ad infinitum with all the processes waiting on the same cell/list. Conceivably, one could get only *circular_array_size* operations per scheduling interval. Because *vqueue* had a much larger array (1024) than did either *queue* or *mqueue* (number of processes) it performed the best of the three.

This behavior is partially attributable to busy-waiting because it forces the scheduler to choose which process is context switched out. A method that chooses whether to busy-wait or nonbusy-wait based on the processes' position in line seems more reasonable. (See reference [11] on busy-waiting versus blocking semaphores.) The scheduler, however, is also at fault. After the first process is scheduled, the second process should be ready to go, which would mean that at least the list that is being waited on would move around the queue. This was not the case however. Presumably all processes had the same priority, and thus selection by the scheduler is essentially random. If the scheduler knew who was waiting, it could do a better job.

Most importantly, however, is that a ticket is used to order the processes well before the critical section is encountered, thus allowing all the context switching that caused the problem. The algorithm of *pool* is almost identical to that of queue but does not enforce an ordering (i.e., no tickets) and thus does not exhibit this behavior. The two hash queues avoid the problem by tagging the items with the position in the queue and thereby avoiding the need to order the operations. Therefore, preassigned, ticket-based algorithms without non-busy-waiting semaphores will not be optimal.

8.3. Multi-Queues

When viewing these data, remember that this test is not a fair one for *mqueue*, in that items with unit multiplicity were put on the queues. *Mqueue*, which was strictly designed as a multi-queue makes no attempt to perform well as a singleton queue. Because the number of processes is always larger than the multiplicity of any item on an *mqueue*, any process that tries to obtain the next item on the queue locks out any others that are also trying to delete an item. This causes a serialization of the processes that results in reduced performance. *Tqueue* does not suffer because it is also able to function well as a singleton queue. Even so, although *tqueue* maintains much better efficiency ratings throughout, *mqueue* outperforms *tqueue* in raw parallel speed until the addition of about the seventh or eighth process.

Tqueue also suffers some serialization, but because each process is forced to obtain an item from a unique leaf of the tree, serialization can be avoided with good process scheduling. Serialization occurs at interior nodes on insertion when the tree is being built. Assume two processes, A and B, enter the queue at the root, with B entering immediately after A. Further assume that A is assigned the leftmost leaf in the tree, with B assigned the second leftmost leaf. A and B travel the same paths down the tree until the lowest level, when they split off to their own leaves. Because A entered first, it will creates the path to the bottom of the tree as it goes, meaning that B will be forced to wait behind A until the very last internal node. With many processes there will be many such threads of serialization, all of different lengths ($0..\log_k N$) and each with a different number of processes. The worst case occurs when the leftmost process in any subtree blocks the processes near the top of the tree. This would most likely be due to PE contention but can also result when the critical process has been delayed waiting for the bus or any other resource.

All this however, is not meant to imply that *n* insertions by *n* processes cannot be done in $O(\log_k N)$ time. If there are a random number of waits at each level, independent of the number of inserters, the number of waits is only a function of the depth of the tree. The total time for all processes entering at the same time and directed toward the smallest subtree will be the sum of the maximum wait time at each internal node.

The *tqueue* data exhibit a decrease in performance when the number of processes equals six. This is statistically significant; the data in Graph 1 show that this is not a coincidence.

Overall, *tqueue* may be the better general purpose multi-queue because it also performs well as a singleton queue. It is slower than *mqueue* but has a guaranteed upper bound on time complexity in the absence of PE contention. *Mqueue* would be useful in a situation where multiplicities are equivalent to the number of processes using the queue or where there is not expected to be much contention for the queue among processes.

8.4. Pools

Comparisons among the pools show that *set* and *pool* are dead even in raw speed. *Cset*'s speed is reduced because of the overhead of the separate internal queue that keeps track of usable words in the bit map. *Set* and *cset* suffer an extra decrease in raw speed because they utilize the *faa*-emulated *fas* and *faor*(*cset* only). Efficiency graphs show that both sets have a more graceful decline in performance than does *pool*. As discussed earlier, this may be important.

When choosing a pool, one should consider the application. *Pool*s can maintain lists of available resources and the two sets' primary uses are for maintaining list of available tags or identifiers (integers). In either application one pool will most likely be associated with many tasks. For *pool*, pool size rises linearly with the number of processors, whereas for the two set algorithms, space does not grow with the number of processes, only with the number of queues. But because it is not likely that the number of queues will grow with the number of processors, the two sets require less space in the highly parallel system.

8.5. Tqueue Performance vs. K-ariness of Tree

The data in Graph 1 clearly demonstrate a performance dependent on the values of *k*. The decrease does not occur when *k* divides the number of processes (*p*). If this were the case, I would expect to see a drop for p=6, when k=2. We do, however, see a drop at p=2,4,8 when k=2 or 4. We also see a drop at p=3,6 when k=6. This suggests a drop in efficiency in either of the two following cases...

1) p divides k

2) p is a power of k greater than one.

This is because busy-waiting goes on at internal nodes when the process assigned to allocate the next node in the tree has been delayed resulting in a forced synchronization or packetizing, of the processes at the internal nodes. This is not due just to PE contention, for it occurs even when adequate PEs are available. The packetizing has the worst consequences in the two cases above. In case one, one group of size k/p is on average moving through the queue "together." Because the size of the group evenly divides the number of leaves in a subtree, the group stays packetized as it moves through the queue. In case two, the number of processes is equal to the number of leaves in some subtree of the queue. In this situation one group of processes are on average grouped at the lowest root of each subtree of depth $\log_k(p)$. The effect becomes worse as the depth of the subtree increases.

Choosing a value of k as a prime number near the expected number of processes that will be using the queue effectively eliminates the effects just discussed.

9. Conclusion

Many of these algorithms require the graceful overflow of integers from the largest representable integer to zero to avoid using more synchronizations to adjust the counters. Thus any parallel machine should utilize a processor that, at least optionally, handles overflow without trapping.

When using distributed linked lists, as in *queue*, it is better to tag each item in the list with a number indicating its position in the queue, rather than trying to keep the list itself ordered. To keep the list ordered one must order the inserts and possibly the deletes; because of race conditions, however, this is somewhat cumbersome. Not ordering the lists and putting items at the head of lists, the approach taken by *hqueue* and *dhqueue*, means that in the worst case one will have to search through all the items. This reduces performance, but seems necessary, in the absence of non-busy-waiting semaphores, to implement a distributed cell or list based queue efficiently.

An alternative to this problem is the technique of "temporary nonpreemption," discussed in [12], in which a processes can be in one of two states, *okpreempt* or *nopreempt*. *Okpreempt* was effectively used throughout these tests; *nopreemt* is a temporary condition that can be requested by the process so as to eliminate the type of problem seen in *queue* and *mqueue* effectively.

As we just saw, it is recommended that the k-ariness of the tree in *tqueue* be chosen as a prime number near the expected number of processes using the queue. This concept can most likely be extended to the general class of parallel tree-based algorithms as well.

10. Future Work

This research suggests several avenues for future work in the areas of parallel queues and their possible improvements. Ideally, more queuing algorithms should be tested when they become available (i.e., developed). Among the two hash queues, small variants regarding list manipulation and interior removal could also be tested. For example, instead of items being put on the head of a list, they could be put on the end; this way, the next item to come off the queue would not be pushed deeper into the list.

A more critical test of these queues would be to run them on a machine that supports more than eight processors, possibly thirty (Sequent/Balance) or sixty-four (IBM/RP3), which would then give a more accurate measure of these algorithms' scalability. This is of critical importance. Speed up factors of five to ten (Table 3), cannot justify the time and effort being put into parallel systems today. If, however, the scalability is constant with the number of processors, as with *hqueue* and the pools, then it could be justified.

A related issue is that of the effects of PE contention. For example, it would be useful to study how non-blocking synchronizations would affect the pathologic behavior of *queue* and *vqueue*. One should use an intelligent scheduler that recognizes critical processes or the temporary nonpreemption scheme covered in [12].

Another related topic is the "barrier" effects in *fqueue* and *tqueue* in which a group of processes is forced to proceed only as fast as the slowest process in the group. In some situations this may not be avoidable, but there may be a way to reduce the effects. For instance, the group lock family could include a routine that provides premature exit from a group.

As always, work should proceed on new synchronization algorithms. Eric Freudenthal at NYU's Ultracomputer group is currently working an a family of synchronizations that he believes can provide the functionality of reader-reader, reader-writer, or group locks. These and other algorithms (busy-waiting or blocking) should then be used in place of the synchronizations used here in order to find the optimal implementation of each set of algorithms.

11. Acknowledgments

I want to thank all the people in the Ultracomputer group who have helped me along the way on this project, especially Eric Freudenthal and Wayne Berke for helping me find the bugs that can be so elusive in parallel programs. Without Jan Edler I would never have began this project; he and I spent hours discussing the various algorithms and the best ways in which to test them and, his suggestions and those discussions led to the development of *dhqueue*. Many, many thanks, Jan.

12. Appendix A. Pseudo-Code

Llist		
Initialize(q,parallelism)	Insert(q,item)	Delete(q)

q.cnt := 0
q.head := NULL
q.tail := NULL
PVinit(q.lock,1)
	P(q.lock,1)
	    - put item at end of list
	    q.cnt = q.cnt + 1
	V(q.lock,1)
		P(q.lock,1)
		  if (q.cnt > 0)
		    - get item from head of list
		    q.cnt = q.cnt - 1
		  else
		    - delete failed
		  endif
		V(q.lock,1)

Queue		
Initialize(q,parallelism)	Insert(q,item)	Delete(q)

q.cnt := 0
q.tail := 0
q.head := 0
q.size := parallelism
for i := 0 to q.size-1 do begin
    q.lists[i].cnt := 0
    q.lists[i].putticket := i
    q.lists[i].getticket := i
    PVinit(q.lists[i].lock,1)
endfor
	mytail := faa(q.tail,1)
	mylist := mytail **mod** q.size
	list := (address of)q.lists[mylist]
	repeat
	until (list->putticket = mytail)
	P(list->lock,1)
	    - put item at end of list
	V(list->lock,1)
	faa(list->putticket,q.size)
	faa(list->cnt,1)
	faa(q.cnt,1)
		if (not tdr(q.cnt,1))
		    return(queue empty)
		myhead := faa(q.head,1)
		mylist := myhead **mod** q.size
		list := (address of)q.lists[mylist]
		repeat
		until (list->getticket = myhead)
		repeat
		until (tdr(list->cnt,1))
		P(list->lock,1)
		    - get item from head of list
		V(list->lock,1)
		faa(list->getticket,q.size)

Appendix A. (cont)

Fqueue

Initialize(q,size)	Insert(q,item)	Delete(q)

q.cnt := 0
q.tail := 0
q.head := 0
q.size := size
grp_init(q.glock)
	grp_lock(q.glock)
	if (not tir(q.cnt,1,q.size))
	grp_synch(q.glock)
	grp_unlock(q.glock)
	return(queue full)
	endif
	mycell := faa(q.tail,1) **mod** q.size
	- put item into cell
	faa(q.cnt,1)
	grp_synch(q.glock)
	grp_unlock(q.glock)
		grp_lock(q.glock)
		grp_synch(q.glock)
		if (not tdr(q.cnt,1))
		grp_unlock(q.glock)
		return(queue empty)
		endif
		mycell := faa(q.head,1) **mod** q.size
		- take item from cell
		grp_unlock(q.glock)

Vqueue

Initialize(q,size)	Insert(q,item)	Delete(q)

q.empty := size
q.filled := 0
q.tail := 0
q.head := 0
q.size := size
for i := 0 to q.size-1 do begin
q.cell[i].ticket := i
endfor
	if (not tdr(q.empty,1))
	return(queue full)
	mytail := faa(q.tail,1)
	mycell := mytail **mod** q.size
	cell := (address of)q.cells[mycell]
	repeat
	until (cell->ticket = mytail)
	- put item into cell
	faa(cell->ticket,1)
	faa(q.filled,1)
		if (not tdr(q.filled,1))
		return(queue empty)
		myhead := faa(q.head,1)
		mycell := myhead **mod** q.size
		cell := (address of)q.cells[mycell]
		repeat
		until (cell->ticket = myhead+1)
		- get item from cell
		faa(cell->ticket,q.size-1)
		faa(q.empty,1);

Hqueue

Initialize(q)	Insert(q,item)	Delete(q)

q.cnt := 0
q.tail := 0
q.head := 0
q.id := cset_delete(cset);
	pos := faa(q.tail,1)
	mylist := hash(q,pos)
	list := (address of)q.lists[mylist]
	- mark item with q.id and pos
	P(list->lock,1)
	- put item on list
	V(list->lock,1)
	faa(q.cnt,1)
		if (not tdr(q.cnt,1))
		return(queue empty)
		pos := faa(q.head,1)
		mylist := hash(q,pos)
		list := (address of)q.lists[mylist]
		repeat
		- search list for item with
		(i.qid = q.id and i.pos = pos)
		until (item is found)
		P(list->lock,1)
		- get item from list
		V(list->lock,1)

Appendix A. (cont)

Dhqueue		
Initialize(htab)	Initialize(q,htab)	Hash(q,pos)

htab.cnt := 0
htab.size := MinTabSize
for i := 0 to MinTabSize-1 do
    initialize_list(htab.lists[i])
csetinit(htab.qids,max_queues)
rw_initlock(htab.rwlock)
rr_initlock(htab.rrlock)
	q.cnt := 0
	q.tail := 0
	q.head := 0
	q.htab := htab
	q.id := 2*cset_delete(htab.qids)-1;
		tmp := q.id * pos
		p := q.htab.p
		maxp := q.htab.maxp
		hash := tmp **mod** maxp
		if (hash < p)
		    hash := tmp **mod** 2*maxp

Insert(q,item)	Delete(q)	Expand(htab)

pos := faa(q.tail,1)
- mark item with q.id and pos
repeat
    mylist := hash(q,pos)
    list := (address of)q.lists[mylist]
    repeat
    until (list is initialized)
    P(list->lock,1)
        if (mylist = hash(q,pos))
            - put item on list
    V(list->lock,1)
until (item is put on a list)
faa(q.cnt,1)
faa(q.htab.cnt,1)
if (Depth(q.htab) > MaxLF)
    rr_Alock(q.htab.rrlock)
    expand(q.htab)
    rr_Aunlock(q.htab.rrlock)
endif
	if (not tdr(q.cnt,1))
	    return(queue empty)
	pos := faa(q.head,1)
	repeat
	    mylist := hash(q,pos)
	    list := (address of)q.lists[mylist]
	    repeat
	    until (list is initialized)
	    P(list->lock,1)
	    - search list once for item with
	    (i.qid=q.id and i.pos=pos)
	    if (found)
	        if (mylist = hash(q,pos))
	            - take item from list
	    endif
	    V(list->lock,1)
	until (item is found)
	faa(q.htab.cnt,-1)
	if (Depth(htab) < MinLF)
	    rr_Block(q.htab.rrlock)
	    shrink(q.htab)
	    rr_Bunlock(q.htab.rrlock)
	endif
		rw_rlock(htab.rwlock)
		faa(htab.size,1)
		maxp := htab.maxp
		p := faa(htab.p,1)
		if (p = maxp-1)
		    read2write(htab.rwlock)
		    htab.p := htab.p - maxp
		    htab.maxp := 2*maxp
		    write2read(htab.rwlock)
		else
		    rr_runlock(htab.rwlock)
		    -wait for htab.maxp to be reset
		    p := p **mod** maxp
		    maxp := htab.maxp
		    rr_rlock(htab.rwlock)
		endif
		P(htab.lists[p],1)
		    - rehash items in list p
		    - put items rehashing to
		     p+maxp list into external list
		V(htab.lists[p],1)
		initialize_list(htab.lists[p+maxp])
		P(htab.lists[p+maxp],1)
		    - link in external list of items
		V(htab.lists[p+maxp],1)
		rw_runlock(htab.rwlock)

Appendix A. (cont)

Mqueue

Initialize(q,parallelism)

```
q.cnt := 0
q.tail := 0
q.head := 0
q.size := parallelism
q.firstmult := 0
rw_initlock(q.rwlock)†
for i := 0 to q.size-1 do begin
   q.lists[i].cnt := 0
   q.lists[i].putticket := i
   q.lists[i].getticket := i
   PVinit(q.lists[i].lock,1)
endfor
```

Insert(q,item,mult)

```
rw_rlock(q.rwlock)†
mytail := faa(q.tail,1)
mylist := mytail mod q.size
list := (address of)q.lists[mylist]
repeat
until (list->putticket = mytail)
P(list->lock,1)
   - put single item with given
     multiplicity at end of list
   if (mytail = q.head)
     q.firstmult := mult
V(list->lock,1)
faa(list->putticket,q.size)
faa(list->cnt,1)
faa(q.cnt,1)
rw_runlock(q.rwlock)†
```

Delete(q)

```
rw_rlock(q.rwlock)†
if (not tdr(q.cnt,1))
   rw_runlock(q.rwlock)†
   return(queue empty)
endif
repeat
   repeat
   until (q.firstmult > 0)
until (faa(q.firstmult,-1) > 0)
mylist := q.head mod q.size
list := (address of)q.lists[mylist]
repeat
until (tdr(list->cnt,1))
- use item at head of list
if (faa(list->first->mult,-1) = 1)
   P(list->lock,1)
      - remove first item from list
   V(list->lock,1)
   repeat
     mylist := (mylist+1) mod q.size
     list := (address of)q.lists[mylist]
     P(list->lock,1)†
        q.head := mylist
        q.firstmult := list->first->mult
     V(list->lock,1)†
   until (q.firstmult > 0)
endif
rw_runlock(q.rwlock)†
```

† Necessary in order to support interior removal.

Appendix A. (cont)

Tqueue

Initialize(q,size)

```
q.cnt := 0
q.tail := 0
q.head := 0
q.tailcheck := 0
q.headcheck := 0
q.size := size
q.logsize := log_k(size)
q.root[0] := null
q.root[1] := null
rw_initlock(q.rwlock)
```

Insert(q,item,mult)

```
rw_rlock(q.rwlock)
if (not tir(q.qtcheck,mult,2*q.size))
   return(queue full)
pos := faa(q.tail,mult)
if ((pos < q.size) and (pos+mult > q.size))
   additems(q.root[0],item,pos,
          q.logsize,q.size-pos)
   additems(q.root[1],item,0,
          q.logsize,mult-q.size+pos)
else if (mult = q.size)
   q.root[pos div q.size] :=
      initnode[2*q.size,multinode)
else
   child := pos div q.size
   if (pos mod q.size = 0)
      q.root[child] :=
        initnode]2*q.size,leafnode)
   else
      repeat
      until (q.root[child] <> null)
   endif
   additems(q.root[child],item,pos,
      q.logsize,mult)
endif
rw_runlock(q.rwlock)
```

Delete(q)

```
rw_rlock(q.rwlock)
if (not tir(q.headcheck,1,q.qtail))
   return(queue empty)
pos := faa(q.head,1)
node := q.root[pos div q.size]
while (node is a not leaf- or multi- node)
   lastnode := node
   - based on pos go down a level
     in the tree, changing node
   checkfree(lastnode,1)
endwhile
repeat
until (node.item <> null)
item := node.item
checkfree(node,1)
rw_runlock(q.rwlock)
if (last delete for 0th root)
   rw_wlock(q.rwlock)
   q.root[0] := q.root[1]
   q.root[1] := null
   q.tail := q.tail-q.size
   q.head := q.head-q.size
   q.tailcheck := q.tailcheck-q.size
   q.headcheck := q.headcheck-q.size
   rw_wunlock(q.rwlock)
endif
```

Initnode(usecnt,type)

```
node := new(nodetype)
node.usecnt := usecnt
node.type := type
return((address of)node)
```

Additems(node,item,pos,level,mult)

```
if (mult = 1)
   while (not at leaves)
      lastnode := node
      -go down a level based on pos
      if (left most child in subtree)
         node.child[dest] :=
           initnode(k**level,leafnode)
      else
         repeat
         until (node.child[dest] <> null)
      endif
      checkfree(lastnode,1)
   endwhile
   node.item := item
else
   - recursively distribute mult items
     among children
   checkfree(node,mult)
endif
```

Checkfree(node,n)

```
if (faa(node.usecnt,-n) = n)
   free(node)
```

K is the number of children of each node in the tree

Appendix A. (cont)

Pool		
Initialize(q,parallelism)	Insert(q,item)	Delete(q)

q.cnt := 0
q.tail := 0
q.head := 0
q.size := parallelism
for i := 0 to q.size-1 do begin
   q.lists[i].cnt := 0
   PVinit(q.lists[i].lock,1)
endfor
	mylist := faa(q.tail,1) **mod** q.size
	list := (address of)q.lists[mylist]
	P(list->lock,1)
	   - put item at head of list
	V(list->lock,1)
	faa(list->cnt,1)
	faa(q.cnt,1)
		if (not tdr(q.cnt,1))
		   return(queue empty)
		mylist := faa(q.head,1) **mod** q.size
		list := (address of)q.lists[mylist]
		repeat
		until (tdr(list->cnt,1))
		P(list->lock,1)
		   - get item from head of list
		V(list->lock,1)

Set		
Initialize(s,size)	Insert(s,value)	Delete(s)

s.cnt := 0
s.tail := 0
s.head := 0
s.size := size
for i := 0 to s.size-1 do
   s.cells[i] := i + 1
	mycell := faa(s.tail,1) **mod** s.size
	repeat
	   value := fas(s.cells[mycell],value)
	until (value = 0)
	faa(s.cnt,1)
		if (not tdr(s.cnt,1))
		   return(set empty)
		mycell := faa(s.head,1) **mod** s.size
		repeat
		   value := fas(s.cells[mycell],0)
		until (value <> 0)
		return(value)

Cset		
Initialize(s,size)	Insert(s,value)	Delete(s)

s.cnt := 0
s.tail := 0
s.head := 0
s.size := size
s.words := size/bitsperword
for i := 0 to s.words-1 do begin
   s.q[i] := i
   s.bitmap[i] := all bits on
endfor
	word := (value-1)/bitsperword
	bitmask := 1 **shl** (value **mod** bitsperword)
	if (faor(s.bitmap[word],bitmask) = 0)
	   mycell := faa(s.tail,1) **mod** s.words
	   repeat
	      word := fas(s.q[mycell],word)
	   until (word < 0)
	endif
	faa(s.cnt,1)
		if (not tdr(s.cnt,1))
		   return(set empty)
		mycell := faa(s.head,1) **mod** s.words
		repeat
		   word := fas(s.q[mycell],-1)
		until (word >= 0)
		bits := fas(s.bitmap[word],0)
		pos := position of set bit in bits
		- turn off corresponding bit in bits
		value := word*bitsperword+pos+1
		if (faor(s.bitmap[word],bits) = 0)
		   mycell := faa(s.tail,1) **mod** s.words
		   repeat
		      word := fas(s.q[mycell],word)
		   until (word < 0)
		endif
		return(value)

13. Appendix B. Cqtst option flags

The following are the option flags that allow almost any combination of the options above. When illegal combinations are specified, such as multiple item insertion on sets, the user is notified and the options are automatically adjusted.

-q<Q> : Q determines queue type to be tested.

0	No queuing done, base timing.		
	Fifo		
fq	fqueue	tq	tqueue
hq	hqueue	mq	mqueue
q	queue	ll	llist
dhq	dhqueue	sll	sllist
vq	vqueue		
	Non-Fifo		
set	set	cset	cset
pool	pool		

-m<M> : M is the number of queues of the specified type to use.

-n<N> : N is the number of distinct elements to be distributed among the M queues. Default = 1024.

-c<C> : C is a number such that the multiplicity of each of the N elements is randomly chosen to range from 1 to C. Default = 1.

-s<S> : S is the number of children to spawn, during parallel processing. If loading and emptying in parallel then the number of children used for each task is S/2. Default = 8.

-r<R> : R specifies the interior removal method to be used.
1 Removal is done in place of empty.
2 Removal is done before any emptying is done. For this reason serial load is forced. Default = 0, no removal.

-l<c> : Load queues serially/parallelly. c = s/p,
s -- serially p -- parallelly.

-e<c> : Empty queues serially/parallelly. c = s/p,
s -- serially p -- parallelly.

-ec : Turn on queue error checking. Default is none.

-t<T> : T is the number of cycles the queues are run through. The effect is to run *cqtst* T times. This is used mostly just to get better averages. Since the queue is not reinitialized for each cycle, T>1 may force an adjustment of the queue(s).

-ps : Queue exercising is to be done serially.

-pt : Produce short form (terse) print out.

-p<P> : P is the number of delete/re-insert operations to be performed per queue per child.

-b : Turn off overhead subtraction.

-h<H> : Valid only for hqueues. H is the size of the hash table to be used for the M queues. Default = 1031.

14. Bibliography

[1] *Two New Parallel Queue Algorithms*, Isaac Dimitrovsky,
Ultracomputer System Software Note #58, February 1985.

[2] *A New Parallel Multiqueue Algorithm*, Isaac Dimitrovsky,
Ultracomputer System Software Note #61, March 1985.

[3] *Restricted Parallel Sets,* Jan Edler, Unpublished, March 1988.

[4] *Dynamic Linear Hashing*, Per-Ake Larson, Communications of the ACM,
Vol. 31 No. 4, April 1988, pp. 446-450.

[5] *Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors,*
Allan Gottlieb, B.D. Lubachevsky, and Larry Rudolph,
ACM TOPLAS Vol. 5, April 1983, pp. 164-189.

[6] *A Group Lock Algorithm, With Applications*, Isaac Dimitrovsky,
Ultracomputer System Software Note #112, November 1986.

[7] *An Overview of the NYU Ultracomputer Project*, Allan Gottlieb,
Ultracomputer Note #100, October 1986.

[8] *The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer,*
A. Gottlieb, R. Grishman, C.P. Kruskal, et al.
IEEE Transactions on Computing Vol. C-32 No. 2, February 1983, pp. 175-189.

[9] *Readers/Readers Synchronization*, Jan Edler
Ultracomputer System Software Note #46, March 1984.

[10] *Upgrading a Read Lock to a Write Lock*, Jan Edler
Ultracomputer System Software Note #54, September 1984.

[11] *Spinning Versus Blocking in Parallel Systems with Uncertainty*,
John Zahorjan and Edward Lazowska (University of Washington),
Derek Eager (University of Waterloo)
Tech. Report 88-03-01, March 1988.

[12] *Process Management for Highly Parallel UNIX Systems*, Edler, Lipkis, Schonberg
USENIX 1988 Supercomputer Workshop

Table of Contents