Priority Queues and Sorting Methods for Parallel Simulation

Miltos D. Grammatikakis

Faculty of Informatics, Dept. Parallel Systems

C.v.O. University of Oldenburg

D-26111 Oldenburg, Germany

Stefan Liesche

IBM Deutschland Entwicklung GmbH

Schönaicher Straße 220

D-71023 Böblingen, Germany

Abstract

We examine the design, implementation, and experimental analysis of parallel priority queues for device and network simulation. We consider: a) distributed splay trees using MPI, b) concurrent heaps using shared memory atomic locks, and c) a new, more general concurrent data structure based on distributed sorted lists, which is designed to provide dynamically balanced work allocation (with automatic or manual control) and efficient use of shared memory resources. We evaluate performance for all three data structures on a Cray-T3E900 system at KFA-Jülich. Our comparisons are based on simulations of single buffers and a 64×64 packet switch which supports multicasting. In all implementations, PEs monitor traffic at their preassigned input/output ports, while priority queue elements are distributed across the Cray-T3E virtual shared memory. Our experiments with up to 60,000 packets and 2 to 64 PEs indicate that concurrent priority queues perform much better than distributed ones. Both concurrent implementations have comparable performance, while our new data structure uses less memory and has been further optimized. We also consider parallel simulation for symmetric networks by sorting integer conflict functions and implementing an interesting packet indexing scheme. The optimized message passing network simulator can process $\sim$ 500K packet moves in 1sec, with an efficiency that exceeds $\sim$ 50% for a few thousands packets on the Cray-T3E with 32 PEs. All developed data structures now form a parallel library. Although our concurrent implementations use the Cray-T3E ShMem library, portability can be derived from Open-MP or MPI-2 standard libraries, which will provide support for one-way communication and shared memory lock mechanisms.

Index Terms: Concurrent data structure, Cray-T3E, data race, distributed data structure, memory lock, priority queue, parallel simulation, virtual shared memory.

1 INTRODUCTION

In 1964, Knuth considered an abstract data type (ADT), called *priority queue* [17]. It consists of a set of items H. Each item is assigned a *priority value* chosen from a totally ordered domain. A lower value means a higher priority and vice-versa. Two ADT operations *Insert* and *DeleteMin* are provided: a) `Insert(a, H)` inserts a new item into the set ($H = H \cup a$), and b) `a=DeleteMin(H)` returns the item with highest priority and removes it from the set ($a = \min H; H = H - a$).

A parallel priority queue can perform these ADT operations concurrently. The main difference of priority queues compared with other related parallel algorithms, such as distributed databases, or dictionary machines, comes from the centralized control necessary for computing the node with the highest priority, during DeleteMin. This requirement creates memory and network hot spots which result in bottlenecks at this node's neighborhood. Furthermore, since applications may frequently change the data structure size, some sort of load balancing is necessary for efficient implementation.

Priority queues are common in parallel and distributed applications. They represent the optimal data structure for many problems:

- Parallel branch and bound algorithms for decision making in finance, or games are implemented using priority queues.
- In numerical iterative schemes, such as graph algorithms, priority queues are used to compute the "next" item to access; e.g. both Kruscal's algorithm for the minimal spanning tree problem (arizing for example in network design) and Dijkstra's algorithm for the shortest path problem are based on DeleteMin operations.
- Heuristics for NP-complete problems, e.g. bin packing and traveling salesman require backtracking with next visited elements chosen according to size or distance metrics.
- Pattern matching or data compression techniques for quality assurance, and pattern recognition for identification purposes can use priority queues for processing signs ordered by their relative significance.
- In fuzzy search on large data sets, priority queues are used for providing statistical information, e.g. hit lists from internet search engines, and reliability ratings for internet

sites.

- The core of job schedulers in operating systems for thread or process management, and event schedulers in production and simulation applications are priority queues.

Heaps are ideally suited to implement sequential priority queues [33]. A heap is a binary tree with the property that the values stored at any node are always greater than or equal to values stored at both child nodes (if they exist). All levels, except the last, are completely filled. Several parallel priority queues, such as calendar queues, have been proposed and their parallel (PRAM) complexity has been theoretically examined [4, 5, 16, 27, 28].

The experimental evaluation of parallel priority queues consists of both *distributed* [20], and also *concurrent implementations* [15, 26]. All previous implementations have focused on different parallel systems, and thus information on the relative performance of parallel priority queues is very limited.

In this study, we aim at resolving this issue, by developing and evaluating three different parallel priority queue data structures on the same parallel platform, a Cray-T3E 900 MPP system available to us from KFA-Jülich (256 and 512 node configurations).

All data structures are implemented using libraries that support the programming model for which the data structure is proposed. This ensures that all data structures perform as good as possible, without any overheads induced by emulating programming models within our implementations. The concurrent data structures form a software library for further research and development in parallel simulation, directed at:

- simulating internal events in communication switches [32],
- evaluating communications protocols, e.g. flow control and priority schemes,
- simulating large communication networks, e.g. multibutterflies [1, 18, 19].

The first implementation uses MPI to develop a *distributed priority queue structure* (DPQ). This data structure was introduced by Mans in 1996, and targeted point-to-point multicomputers that do not share memory [20]. A *distributed data structure* is allocated to different processing elements (PEs) and may be accessed by many processes simultaneously [25]. A special root PE provides the global view of a heap, with each node consisting of a splay tree of items. Insertions and deletions invoke always the root PE.

The next two *concurrent data structures* use the Cray ShMem virtual shared memory programming library.

The second implementation is based on a concurrent priority queue introduced for uniform shared memory (UMA systems) by Rao and Kumar in 1988 [26] and improved by Hunt, Michael, Parthasarathy and Scott in 1996 [15]. The logical data structure is a heap, i.e. a binary tree. Our *randomly hashed concurrent priority queue* (RCPQ) provides a virtual shared memory implementation. Data for each priority queue node is stored in arrays at predetermined randomly hashed PE locations.

Our main contribution in this paper involves the design and implementation of a new concurrent data structure based on distributed sorted circular lists. We develop the *dynamically balanced concurrent data structure* (BCPQ). Although BCPQ is much more versatile than a parallel priority queue, it can also play this role quite efficiently. BCPQ is designed to provide balanced work allocation and efficient use of the virtual shared memory resources. Both automatic, and user control of the load balancing phase is provided.

Our comparisons of the parallel data structures are based on simulations of single buffers, and a 64 x 64 multicast packet switch. The switch uses multicasting and different PEs model traffic at their preassigned input and output ports. Buffers are modeled by priority queues, with buffer elements distributed across all PEs. Furthermore, we briefly examine performance of these priority queue structures for simulating large packet switching networks; in this case, we propose an alternative data parallel model based on sorting predefined integer conflict functions [7].

Our experiments with up to 60,000 packets and 2 to 64 PEs indicate that, concurrent priority queues are 5-10 times faster than distributed ones, even though DPQ implements asynchronous insertions and deletions. Furthermore, both concurrent implementations are flexible to use and have comparable performance, while RCPQ uses more memory. They can be further improved by using faster lock implementations, such as MCS lock [10], and they can easily be ported into MPI-2 or Open-MP, thus providing platform-wide portability.

Another contribution in our paper is a novel approach for parallel simulation of symmetric communication networks. Our algorithm resolves packet conflicts by sorting appropriate integer conflict functions. We implement our method using C+MPI or C+ShMem, and perform critical improvements aimed at reducing sorting overhead, minimizing inter-processor communication, and optimizing scalar processing. The optimized simulator can process $\sim 500K$ packet moves in 1sec, with an efficiency that exceeds $\sim 50\%$ for a few thousands packets on the Cray-T3E with 32 PEs.

In Section 2, we describe the DPQ and CPQ priority queue data structures and focus on our implementations on the Cray-T3E. In Section 3, we examine the design and implemen-

tation of a new and more general dynamically balanced concurrent data structure (BCPQ). In Section 4, we first study the algorithmic complexity of each data structure; the BCPQ is viewed only as a priority queue. Then, we provide experimental performance evaluations based on priority buffer systems, multicast packet switches, and packet switched networks. Conclusions and extensions are discussed in Section 5. The basic Cray-T3E virtual shared memory (ShMem) functions needed to understand the BCPQ implementation are given in an Appendix (Section 6).

2 PARALLEL PRIORITY QUEUES - IMPLEMENTATIONS

In this Section, we examine the distributed priority queue by Mans (DPQ) and the randomly hashed concurrent priority queue (RCPQ). We also briefly discuss the implementation of these data structures on the Cray-T3E using MPI and the Cray ShMem library respectively.

In our presentation, *items* refer to data elements in the set H, together with special attributes required by each implementation. The *minimum item* (*maximum item*) is the local item with the highest (respectively, lowest) priority. *Nodes* refer to actual PE positions where items are located.

2.1 Distributed Priority Queue (DPQ)

The DPQ priority structure among PEs is based on a *d-ary heap*, or a *binomial tree*. Each PE keeps its local elements ordered using a *splay tree*, a self-adjusting form of a binary search tree [30]. The *splaying* heuristic expects that operations in sequence often perform in the same part of the tree. Thus, the tree is transformed (*splayed*) after each operation. The accessed item is moved to the root of the tree and the depth of each item along the access path becomes smaller. *Splay trees* can perform worst-case sequences of operations as fast as balanced trees [30].

The overall DPQ structure, shown in Figure 1, retains the strict priority ordering of serial heap algorithms. The distributed heap condition implies that *elements at any PE have always higher priorities than their children*. To preserve this condition both ADT operations start at the root PE, i.e. centralized control is needed, and proceed in top-down fashion.

Insert Operation During an *Insert* the invoked PE sends the new item and transfers control to the root PE. Subsequently, the root computes the next PE_p in round-robin fashion, and initiates an insertion along the path to PE_p. PEs on this path react by successively inserting the new item into their own *splay tree*, and pushing down the maximum item. PE_p will

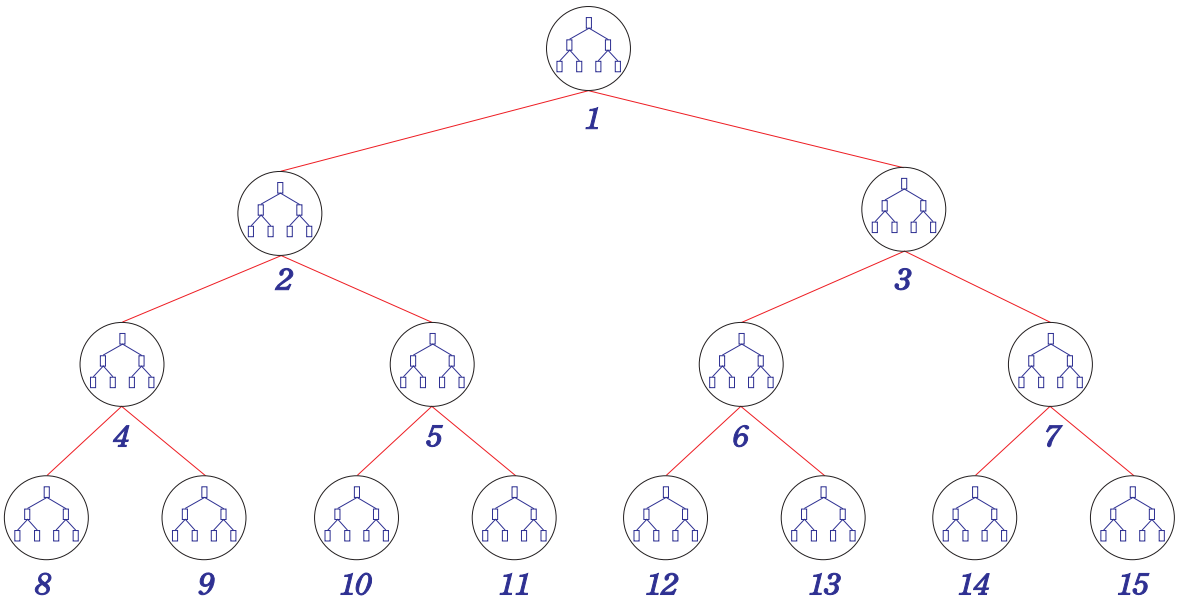

Figure 1: The DPQ data structure as binary heap

finally insert the received item into its local *splay tree*, thus increasing its working load. If the insertion to PE_p violates locally the heap condition, PE_p initiates heapification; this causes further item exchanges down the heap, until the distributed heap condition is satisfied.

DeleteMin Operation A *DeleteMin* request also starts at the root PE. The root PE returns the minimum item from its own *splay tree*. If the priority queue is empty the request is queued until an *Insert* operation is performed. Otherwise, the root computes the next PE_p in round-robin fashion in reverse order. Along the path from PE_p to root each PE sends its own minimum to its parent, thus eventually decreasing the load of PE_p. Upon receiving an item from a child the parent PE has to initiate reheapification if the received item has a larger value than any children item.

In the DPQ, PEs never access the same data simultaneously, thus no synchronization operations are needed.

2.1.1 MPI Implementation of the Distributed Priority Queue

The DPQ has been implemented by Mans [20]. He kindly gave us the right to experiment and modify the DPQ code for our simulation experiments and comparisons. Preliminary performance tests on the Cray-T3E, which has a 3D torus topology have shown that a d-ary heap with degree $d = 5$ has the best performance. Thus, we used a *5-ary heap* in all our experiments.

The DPQ implementation consists of five software layers: a) the inner splay tree layer, b) the virtual topology layer providing parent, son and step operations on a 5-ary heap topology, c) the request queue layer emulating a FIFO for pending DeleteMin operations, d) the *priority queue layer* implementing ADT Insert and DeleteMin operations, and e) the *application layer* from which calls to the data structure are implemented. For more details on the DPQ implementation refer to [23].

2.2 Randomly Hashed Concurrent Priority Queue (RCPQ)

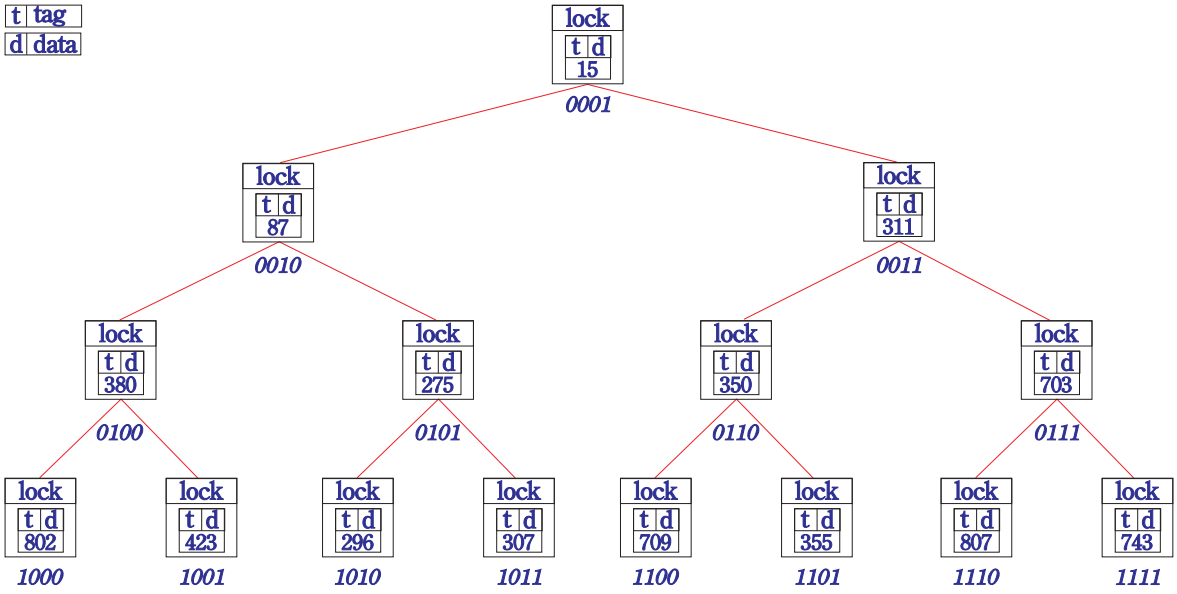

Figure 2: The CPQ data structure

The concurrent priority queue (CPQ) data structure has been proposed for a uniform shared memory environment [15]. The logical CPQ data structure is a binary heap; nodes are arranged in a complete binary tree (see Figure 2). The distributed heap condition implies that *elements at any PE have always higher priorities than their children; furthermore, all PEs except the leaf PEs are nonempty.* In contrast to most heap definitions, CPQ leafs need not be filled in left to right order.

While concurrent operations are performed in the heap, parts of the CPQ data structure may become inconsistent. To avoid locking the whole data structure, and still allow ADT operations to proceed without deadlocks in natural (opposite) order, items in the heap are tagged into either *consistent*, or *transient* states. A consistent state indicates an item that does not violate the heap condition. A transient state indicates an item that may violate the heap condition; this may happen while items are moved by an ADT operation performed on

the heap. Transient states are indicated by storing the process identifier (*pid*) of the process moving the item. Value *Empty* is used to indicate a node that currently does not hold any item.

Insert Operation The *Insert* operation starts at a leaf and traverses the tree bottom-up. To reduce network contention during lock acquisitions, leaf nodes are accessed in shuffled order during concurrent insert operations. This order, similar to the DPQ algorithm, allows consecutive insertions (or deletions) to traverse different subtrees. For example, for the third level of the heap, the shuffle counter would generate the following *binary access sequence*:

$$1000, 1100, 1010, 1110, 1001, 1101, 1011, 1111.$$

A new item is initially stored in the first free node computed (at the spot) by the shuffle counter. From here, the item is moved up level by level. During each move first the parent node, and then the child node are locked. When both nodes are successfully locked and their tags indicate consistent data, item priorities are compared. If the child has a higher priority item, items are swapped. At the end of this step both locks are released. The new item climbs up the tree, step by step, as long as the priority of the item at its parent node is lower than its own priority. When the heap condition is restored, the insert operation is complete.

DeleteMin Operation The *DeleteMin* operation uses the shuffle counter in reverse order, to locate the last *nonempty* node in the heap. It exchanges item *last* stored at the last nonempty heap position with the minimum item r stored at the root; to maintain consistency the corresponding nodes are locked. Once items are exchanged, the last node is marked empty and its lock is released. The root node remains locked, since its item may violate the heap condition. During *heapification* the node holding *last* remains locked. At each step, children are locked and their priorities are compared with that of *last*. If the heap condition is violated, item *last* is swapped with the minimum child item. Only the new node holding *last* remains locked, and DeleteMin proceeds with the next step. DeleteMin completes when item *last* stops to move. Then, the lock of the node holding item *last* is released and DeleteMin returns item r.

Both ADT operations lock tree items. Since *Insert* proceeds top-down and *DeleteMin* bottom-up, cyclic deadlocks can possibly occur. To avoid this problem, previous algorithms have either locked the whole heap data structure, or made *DeleteMin* also proceed in top-down fashion [26]. However, both changes significantly limit concurrency [15]. The CPQ algorithm increases concurrency, and avoids cyclic deadlocks by locking items top-down for both ADT operations [26]. While insert operations proceed bottom up, they always lock first

the parent, and then the child node.

2.2.1 ShMem Implementation of RCPQ

In a virtual shared memory programming model the heap must be distributed among all PEs. Our *randomly hashed concurrent priority queue* (RCPQ) provides a virtual shared memory implementation of CPQ, by allocating priority queue nodes at predetermined randomly selected PE locations. An index list, generated during heap initialization at one PE and subsequently broadcasted to all PEs, maps each heap node to the PE that stores this node. This index enables each PE to remotely access any element in the heap. At PE_0 a virtual shared memory lock is provided for each node. ADT operations for the RCPQ data structure follow closely the corresponding CPQ operations. To avoid inconsistent data or deadlocks, virtual shared memory locks and barriers protect data elements from being simultaneously modified.

Our RCPQ implementation consists of only three software layers: a) the lower *data access layer* providing uniform memory access for set/get operations by abstracting accesses to local and remote memory, b) the *priority queue layer* implementing ADT Insert and DeleteMin operations, and c) the *application layer* from which calls to the data structure are implemented. For more details on the RCPQ implementation refer to [23].

2.2.2 MPI-2 vs. ShMem Implementation

Deciding for a parallel programming environment is a trade-off between portability and performance. Higher level portable libraries are generally slower than platform dependent libraries. The next version of the MPI library (MPI-2) provides one-sided communications along with synchronization routines [22]. MPI-2 routines such as `MPI_Put`, `MPI_Get`, `MPI_Win_Lock` and `MPI_Win_Unlock` support a shared memory programming paradigm. Thus, both our RCPQ and BCPQ (cf. Section 3) concurrent data structure implementations can become portable by using the MPI-2 library. At present, there is no MPI-2 implementation on the Cray-T3E, but implementations on various platforms are now starting to appear.

3 A DYNAMIC CONCURRENT DATA STRUCTURE

Our new concurrent data structure focuses on virtual shared memory NUMA supercomputers, specifically the Cray-T3E. In NUMA systems, access to remote data is much slower, and lock acquisitions can be time consuming. Thus, the aim of our new data structure is to reduce remote data access and lower the number of locks needed to maintain data consistency. Our

data structure can easily be adapted on UMA systems, supporting physical shared memory, by modifying or eliminating the lowest layer data access routines.

3.1 Dynamically Balanced Concurrent Priority Queue (BCPQ)

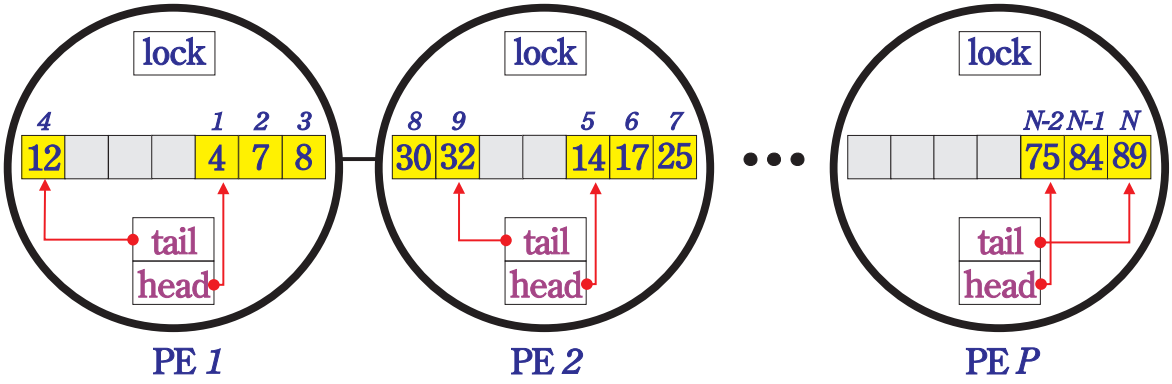

Figure 3: The BCPQ data structure

The BCPQ data structure is essentially a dynamically balanced distributed sorted list which can also play the role of a priority queue. As shown in Figure 3, each PE stores a part of the list in a circular queue, protected by a single lock. PE_0 stores items with the highest priority. Subsequent items are stored at PE_1,...,PE_{P-1} in increasing order (see Figure 3). For simplicity, we refer to the list stored at PE_i as list i.

When the BCPQ is used as a priority queue, the ADT operations are performed as follows.

Insert Operation The *Insert* operation starts with a search for finding the local list on which the new item must be inserted. Thus, the range of nonempty PEs `[0, PE_max]` is binary searched either remotely, or locally.

A *remote binary search* can proceed as follows. Initially, list m, in the middle of the search list is locked, and the priority *prio* of the new item is compared with the minimum and maximum priorities in this list. If *prio* is between these priorities, the target list is m. Otherwise, the lock is released and the search list is halved into two new sublists. If *prio* is smaller than the minimum priority of list m, the search proceeds with the right sublist, otherwise, it continues with the left sublist. The same procedure is repeated until the target list is computed and locked. Special care is needed to avoid an infinite loop if the new priority value falls between two lists. Notice that to avoid deadlocks, the insert operation holds the lock of at most one list at a time. Before locking a new sublist, the lock on the previous list is always released.

For a *local binary search* implementation, each PE needs to know the minimum value (maximum priority value) from every other PE; this can be accomplished with a multinode broadcast. Then, each PE can locally compute (and lock) the appropriate PE.

After the PE binary search, a second binary search locates the target position within the target list. The new item is inserted at this position and the insert operation completes by releasing the PE's lock.

DeleteMin Operation The *DeleteMin* operation first tests if the priority queue is empty, and in that case it returns unsuccessfully. Otherwise, *DeleteMin* must return the minimum item (highest priority) in the first nonempty list. First, list 0 is locked. If it is empty, list 0 is released and the next list is locked. This process is repeated until a nonempty list is found. The minimum item from this list is deleted and returned.

The BCPQ concurrent data structure allows implementing *DeleteMax*, and with minor modifications $delete\_k^{th}\_value$ as efficiently as DeleteMin. *DeleteMax* returns the item with lowest priority. Since the BCPQ remains sorted at all times, one only has to search for the item with the largest index at the last PE. Find operations can also be easily implemented.

3.1.1 Load Balancing the BCPQ data structure

Insert and *DeleteMin* operations may lead to load imbalance in the BCPQ data structure. For example, the target list on which a new item is inserted depends on local list minima and maxima. Since these are in turn influenced by the local number of items and distribution of priority values, some concurrency may be lost if Insert operations are not evenly distributed. Similarly, DeleteMin operations cause an uneven distribution of elements, since they always remove items fr m local lists with small index numbers. Thus, a load balance operation, which can also be called from the application layer is necessary.

To simplify and improve our BCPQ implementation, we assume that the following three conditions, reflecting the state of our distributed list, hold prior to any insert operation.

1. The BCPQ is compact. (All local lists to the right of an empty local list are empty.)
2. In all local lists there is enough memory space to store new items.
3. The BCPQ list is sorted.

Non-random *Insert* operations tend towards violating condition 2 , while *DeleteMin* operations tend towards violating condition 1 . The load balance operation is responsible for maintaining these two conditions. Load balance should only exchange items between neigh-

bors, otherwise condition 3 could be violated. From condition 1 it follows, that any empty lists may exist only at the "extreme right" PEs, i.e. the ones with the highest index numbers.

BCPQ uses a deterministic prefix-based load balance operation, that consists of two phases: a computation phase, followed by a communication phase. Prior to load balance, let the length of local list i be L_i, $0 \leq i < P$.

In the computation phase PE_i $(0 \leq i < P)$ computes the number of items M_i that have to be moved from PE_i to maintain load balance. First, the total number of items N is computed at all PEs, by summing all local counters; in our implementation a total item counter is allocated at PE_0. The average number of items per local list is $\frac{N}{P}$. To equally distribute N items, H_i items must be stored in list i, where

$$H_i = \begin{cases} \left\lfloor \frac{N}{P} \right\rfloor & : \quad i \geq N \bmod P \\ \left\lceil \frac{N}{P} \right\rceil & : \quad i < N \bmod P \end{cases} \qquad (0 \leq i < P) \tag{1}$$

The number of items PE_i has to move to get H_i items is $|K_i|$, where $K_i = H_i - L_i$. A positive value of K_i means that PE_i holds very few items and must get items from its neighbors. A negative value means PE_i has to move items to its neighbors.

Consider prefix list P_i $(0 \leq i < P)$ formed by concatenating all local lists with index smaller or equal to i. Using the previously computed values for K_i, the number of items PE_i must move/get to achieve load balance is:

$$M_i = \sum_{j=0}^{i} K_i \quad (0 \leq i < P) \tag{2}$$

If $M_i > 0$, then PE_i has to get M_i items from the PEs to the right. Otherwise, it has to put $|M_i|$ items to the PEs to the right.

Hence, in the communication phase, all PEs access their right neighbor's memory. Depending on the value of M_i, items are moved either from, or to each local list. At each step as many items as possible are moved, while remaining items are moved in subsequent steps (pipelining). Neighbor to neighbor movements are repeated, until load balance is achieved. For moving items, two local lists must be updated. This means competition in locking local lists. Network congestion is reduced and possible data races [29] (or deadlocks) are eliminated by dividing each communication step into two steps; first, only PEs with even index numbers are active moving items, then odd PEs are active (odd-even pattern).

In Figure 4, an example illustrates the computation and communication phase for load balancing; inactive PEs are shown in gray. Also notice that, the last PE (PE_15) in our load balancing algorithm does not have to perform any local or remote operations.

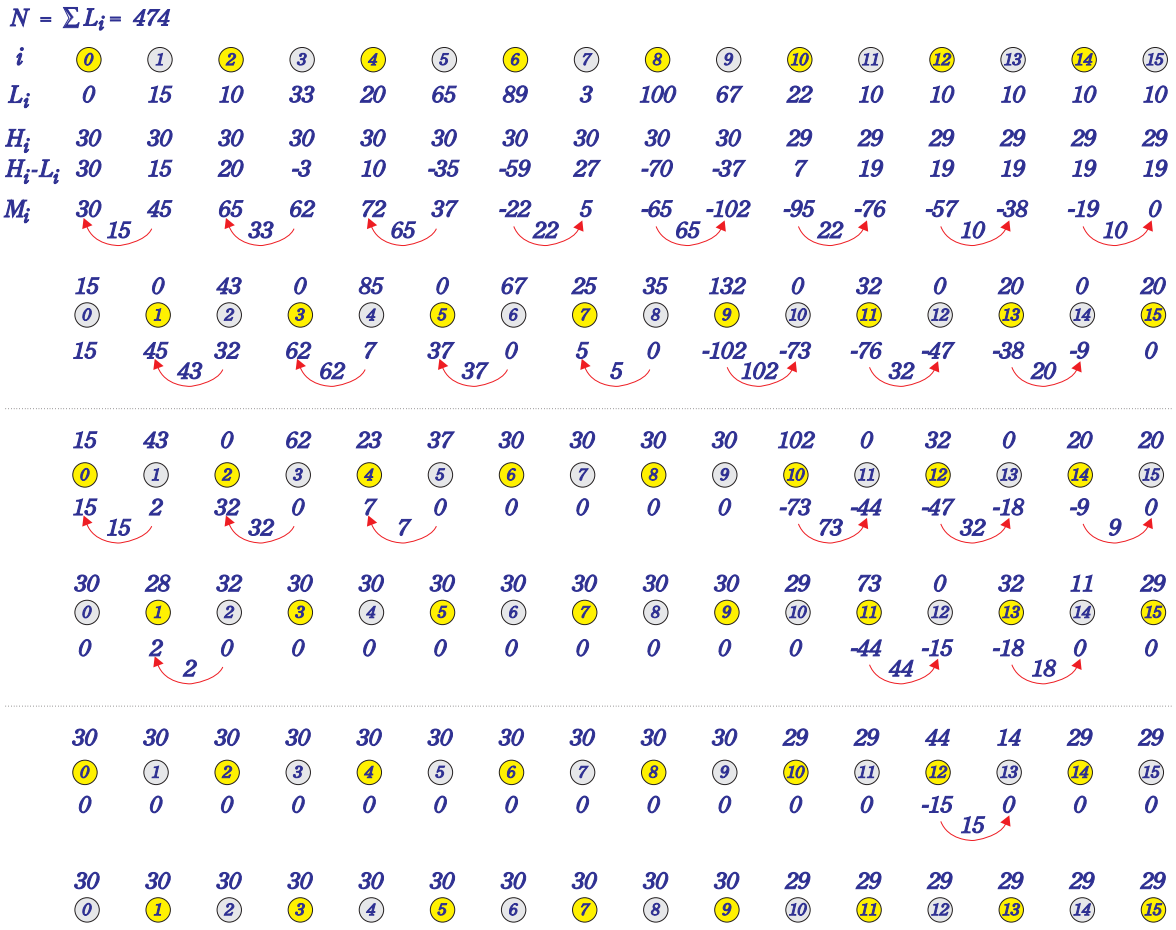

Figure 4: Example of load balance on 16 PEs, using odd-even communication.

3.2 ShMem Implementation of BCPQ

Writing concurrent programs has a reputation for being exotic and hard. We believe it is neither, when the system provides good primitives and suitable libraries and the programmer exercises basic caution and alertness to avoid common pitfalls. We hope that this presentation will help you towards this belief.

Since the BCPQ data structure is new and also more general than either DPQ, or RCPQ, we provide considerable more details of our implementation. As shown in Figure 5, the BCPQ implementation consists of four layers, the *application layer*, the *data organization layer*, the

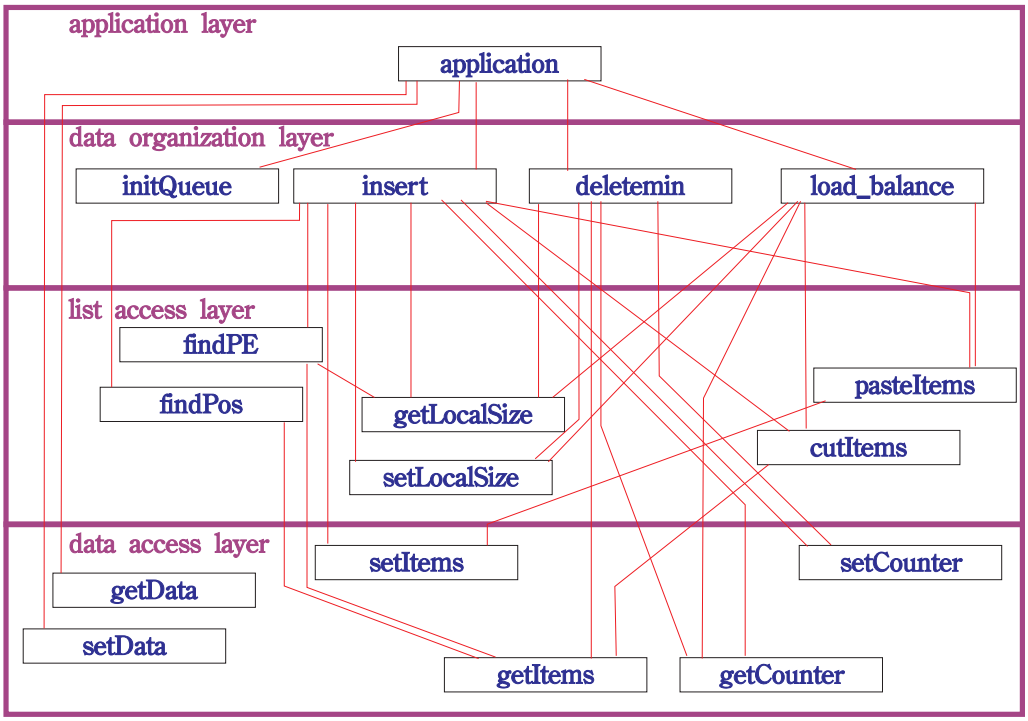

Figure 5: Layers of BCPQ implementation

list access layer, and the *data access layer*.

- Similar to RCPQ, the inner *data access layer* layer uses ShMem functions and local memory operations together, to abstract from the virtual shared memory data distribution. Thus, local and remote accesses are disguised under the same function calls. Functions `getItems` and `setItems` access remote or local items. Functions `getData` and `setData` access remote or local data elements. Functions `getCounter` and `setCounter` access or update the total number of items stored at PE_0. Functions `setNonempty` and `resetNonEmpty` set or reset local list flags representing the number of local elements.

- The *list access layer* abstracts from the implementation of local circular lists. Function `cutItems` cuts a sequence of items from a local circular list, `pasteItems` stores a sequence of items to a given circular list. To examine or update the state of a list, this layer also provides macros, such as `IsEmpty`, `IsFull`, `cyclic_Add`, `cyclic_Dec`, `Size`. Binary search based functions `findPE` and `findPos`, used during the insert operation, are also provided at this layer.

- ADT functions `initQueue`, `Insert`, `DeleteMin` and `load_balance` are provided within the `data organization layer`.
- On the *application layer*, user applications calling the BCPQ data structure can be implemented.

BCPQ Data Structure

```
typedef struct {
  int pe;                                   /* referenced PE */
  unsigned long pos;                        /* data positions in array in that PE */
}pos_t;

typedef struct {
  void *data;                               /* pointer to data (sizeof(data)=N*64Bit */
  int pe;                                   /* Number of PE which has stored the data */
}data_t;

typedef struct {
  long prio;                                /* priority Value different than Noprio*/
  data_t data;                              /* node data */
}node_t;

typedef struct{
   unsigned long first;                     /* array position of first element in list */
   unsigned long last;                      /* array position of last element in list */
}cycle_t;

typedef struct{
   long lock[NPEs][a];                      /* lock for all PEs */
   int nonempty;                            /* true if PE holds at least one element */
   unsigned long numitems;                  /* item counter, used only in root PE */
   cycle_t LocalItems;                      /* item counter for number of local items */
   node_t List[LocalSize+1];                /* array of items + empty pos for ADT cycle */
}queue_t;
```

[a] implemented as global array, because of machine problems

Table 1: BCPQ data structure definitions

The data structure type definitions are shown in Table 1. Items are represented by their priority, a data pointer, and the PE on which the data is stored. Items are arranged in cyclic local lists, that allow appending items to the head or to the tail. Local lists are symmetric arrays across PEs (C language static or global variables), and thus remotely accessible by ShMem operations. A binary variable `nonempty`, is associated with each list; if the local list contains items, it is set to 1, otherwise, it is set to 0. Local lists are protected by locks. The total number of items in the distributed data structure is stored at PE_0 in a global counter `numitems`, protected by a lock `numi_lock`.

At each PE, symmetric variables are used to store the state of the local cyclic list (see Figure 6). `First` stores the index of the smallest item in the circular list (in Figure 6, `first`=5), while `last` stores the index of the largest item (in Figure 6, `last`=1). Items are sorted in anti-clockwise direction. To distinguish between full and empty state, the position

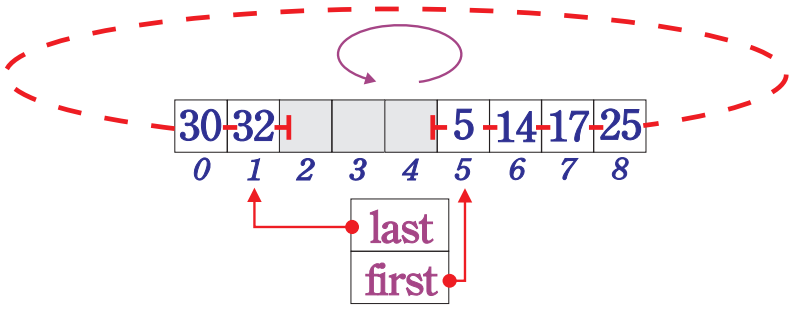

Figure 6: Cyclic sorted list (anti-clockwise direction)

before `first` is always empty. In empty lists, `last` points to this index. A full list is indicated by `last` pointing two positions before `first`. Macros `IsEmpty`, `IsFull`, and `Size` provide the status of a cyclic list. Macros `cyclicInc`, `cyclicDec`, `cyclicAdd` and `cyclicSub` cyclically increase and decrease list counters.

Function `cutItems(pos, n, buffer)` cuts `n` items starting at index `pos` from the local list in anti-clockwise direction and stores them into `buffer`; `buffer` is a `node_t` type array. For instance, `cutItems(7, 3, buffer)` called on the list in Figure 6 removes the items $\{(17, 25, 30)\}$ and stores them into `buffer`. Function `pasteItems(pos, n, buffer)` copies `n` items from `buffer` in anti-clockwise direction into the list, starting at `pos`. In Figure 6, `pasteItems(8,2, (28,29))` would replace 25 with 28 and 30 with 29. Notice that, in this small example, only priority values are given.

Before calling `Insert` the distributed list must be compact and enough space must be available in all local lists. This is handled by `load_balance`. The `Insert` function shown in Table 2, inserts item *prio* by first searching for the local list, then for the index where the new item has to be inserted. First, the global counter is incremented (lines 6–7). Parameter `PEm` represents the last nonempty list. It is computed prior to calling `Insert` using a global sum on the symmetric `nonempty` flags. The function `findPE` finds the target PE for itemp *prio* and locks it (line 8); it binary searches only nonempty local lists 0 to `PEm`, locking PEs while examining them as potential target lists. If the list is empty the new item is inserted using a sequence of set operations (lines 9–19). Otherwise, function `FindPos` determines the index in the circular list of target PE, where item *prio* is to be inserted, in the circular list of the target PE. To minimize moving of items the distance to the start/end of the list is first computed (lines 21–45). If the item is closer to the head, the head sublist is moved one position in front, otherwise tail is moved. Moving sublists is done by copying corresponding items into a local buffer (lines 38 and 45). Item *prio* is appended to this list and the new list

BCPQ Insert Function

```
int Insert (long prio, data_t data, int PEm, queue_t *bcpq){
1   static node_t buffer[LocalSize];
2   static pos_t pos;
3   static cycle_t cycle, rightc, leftc;
4   static ulong anzright, anzleft;
4   static short numitem;
5
6   numitem = fast_shmem_short_finc(&(bcpq→numitems),0);
7   if(numitem ≥ QueueSize) return -1;                          /* list is full */
8   pos.pe = findPE(prio, 0, PEm, bcpq);                        /* find PE and lock it */
9   getLocalSize(pos.pe, &cycle, bcpq);
10  if(IsEmpty(cycle)){
11    pos.pos = cycle.first;
12    buffer[0].prio = prio; buffer[0].data = data;
13    setItems(pos, &buffer[0], 1, bcpq);
14    cycle.last = cycle.first = pos.pos;
15    setLocalSize(pos.pe, cycle, bcpq);
16    setNonEmpty(bcpq, pos.pe);
17    ClearLock(&(bcpq→lock[pos.pe]));
18    return 0;                                                 /* success */
19  }else{
20    pos.pos = findPos(prio, pos.pe, cycle, bcpq);
21    if (pos.pos == (cyclicInc(cycle.last)) {                  /* append item to tail */
22      buffer[0].prio = prio; buffer[0].data = data;
23      pasteItems(pos, 1, &buffer[0], bcpq);
24      cycle.last = cyclicInc(cycle.last);
25    }else if (pos.pos == cycle.first) {                       /* append item to head */
26      pos.pos = cyclicDec(pos.pos);
27      buffer[0].prio = prio; buffer[0].data = data;
28      pasteItems(pos, 1, &buffer[0], bcpq);
29      cycle.first = cyclicDec(cycle.first);
30    }else{                                                    /* insert item */
31      leftc.first = cycle.first;
32      leftc.last = cyclicDec(pos.pos);
33      anzleft = Size(leftc);
34      rightc.first = pos.pos;
35      rightc.last = cycle.last;
36      anzright = Size(rightc);
37      if (anzright < anzleft){
38        cutItems(pos, anzright, &buffer[1], bcpq);
39        buffer[0].prio = prio; buffer[0].data = data;
40        pos.pos = cyclicInc(pos.pos);
41        pasteItems(pos, anzright + 1, &buffer[0], bcpq);
42        cycle.last = cyclicInc(cycle.last);
43      }else{
44        pos.pos = cyclicSub(pos.pos, anzleft);
45        cutItems(pos, anzleft, &buffer[0], bcpq);
46        buffer[anzleft].prio = prio; buffer[anzleft].data = data;
47        pos.pos = cyclicDec(pos.pos);
48        pasteItems(pos, anzleft + 1, &buffer[0], bcpq);
49        cycle.first = cyclicDec(cycle.first);
50    }  }
51    setLocalSize(pos.pe, cycle, bcpq);
52    ClearLock(&(bcpq→lock[pos.pe]));
53    return 0;                                                 /* success */
54}  }/*Insert*/
```

Table 2: BCPQ `Insert` function

is stored back at the new index. The local list counter is updated, the lock is released and `Insert` exits.

BCPQ DeleteMin Function

```
   int DeleteMin (long *priority, data_t *data, queue_t *vspq){
 1   static pos_t pos;
 2   static cycle_t cycle;
 3   static node_t min;
 4   static short numitem, value_dec = -1;
 5
numitem = shmem_short_fadd(&(vspq→numitems), value_dec, 0);

if(numitem < 1){ /* list is empty */

   fast_shmem_short_finc(&(vspq→numitems), 0);

   return -1;
}
 8   pos.pe = 0;
 9   do{                                              /* find PE with nonempty list */
10     SetLock(&(vspq→lock[pos.pe]));
11     getLocalSize(pos.pe, &cycle, vspq);
12     if(IsEmpty(cycle)){
13       ClearLock(&(vspq→lock[pos.pe]));
14       pos.pe = (pos.pe + 1) % NPEs;                  /* cycle if active insert */
15     }
16   }while(IsEmpty(cycle));                          /* wait until item inserted */
17   pos.pos = cycle.first;
18   getItems(pos,&min,1,vspq);
19   *priority = min.prio;
20   *data = min.data;
21   cycle.first = cyclicInc(cycle.first);
22   setLocalSize(pos.pe, cycle, vspq);
23   if (IsEmpty(cycle)) resetNonEmpty(vspq, pos.pe);
24   ClearLock(&(vspq→lock[pos.pe]));
25   return 0;                                                    /* success */
26}/*DeleteMin*/
```

Table 3: BCPQ `DeleteMin` function

The `DeleteMin` function is shown in Table 3. First the central counter on PE_0 is updated (lines 6–7). Then, starting with list 0, local lists are successively locked and their state examined until the first nonempty list is found (lines 8–16). If all local lists are found empty, then we assume that there is an outstanding `Insert` operation. Thus, `DeleteMin` continues checking circularly all local lists, until the `Insert` operation is completed and `DeleteMin` can find the item. The minimum item is always stored at the `first` position in the local list. It is deleted from the list and the head pointer and nonempty flag are updated (lines 17–23). `DeleteMin` is now completed.

The above implementation of `DeleteMin` can be called at any point within an application. Notice that, if all conditions that are needed to call `Insert` hold before `DeleteMin` is called, a much simpler implementation is possible: the item with highest priority is always stored at the first position in local list 0.

The implementation of load balance is shown in Tables 4 and 5, for the computation and communication phase, respectively. In the computation phase, the difference K_i between the

BCPQ Load Balance - Computation Phase

```
   int load_balance (queue_t *vspq){
 1   static node_t buffer[Localsize];
 2   static pos_t pos;
 3   static cycle_t cycle;
 4   static ulong movNo, movNo_now, turn;
 5   static int nitems, partial_sum, i, distance, steps, recv_value, calc;
 6   static short count_done, go_on, done;
 7
 8   count_done = 0; go_on = 1; turn = 0;
 9   getCounter(vspq);
10   steps = ((int) (log((double) (NPEs - 1))/log((double) 2.0))) + 1;
11   calc = (vspq→numitems / NPEs) - Size(vspq→localitems);
12   if (MyId < (vspq→numitems % NPEs)) calc ++;
13   barrier();
14   for (i=0, distance=1; i<steps; i++, distance* =2) {                    /* Scan */
15     if (ProcExists(MyId-distance)) {
16       recv_value = shmem_int_g(calc, MyId-distance);
17       barrier();
18       calc += recv_value;
19     }else barrier();
20     barrier();
21   }
22   partial_sum = calc;                                                  /* partial_sum */
23   movNo = abs(partial_sum);
       .
       .   < Load Balance Communication Phase >
72}/*load_balance*/
```

Table 4: BCPQ `load_balance` function (computation phase)

number of items stored at PE_i and the desired number of items H_i is computed (lines 9–12). The result is stored in the remotely accessible variable `calc`. ShMem does not provide scan routines, hence the computation of the partial sums M_i (Equation 2) was implemented within BCPQ using a *recursive doubling* communication pattern (lines 14–21). The algorithm for P PEs consists of $\log_2(P)$ steps. In step j $(0 \leq j < \lceil \log_2 P \rceil)$ PE_i is getting the partial sum from PE_{i-2^j} (line 16), if this PE address is valid (tested by `ProcExists` macro) `calc` is added to the received value (line 18). The two barriers (line 17/19 and 20) are needed to avoid a data race between local writes and remote reads from `calc` and vice-versa. After the second barrier, the cache line for `calc` is also invalidated, so that the register value can be used. One could rewrite the scan loop to use locks instead of barriers; however, this is not so efficient. After all PEs complete the scan operation (line 22), partial sum M_i is stored in `partial_sum` at PE_i, and the absolute value is stored in `movNo`.

As shown in Figure 7, a parallel scan operation (shown as a partial sum, base 2 instantiation) can be performed using several approaches, such as (a) recursive doubling, (b) divide and conquer, (c) odd-even reduction, and (d) recursive broadcasting. The change of values during the communication/computation phase of these patterns is also shown in the figure. These communication patterns are also useful for implementing efficient parallel algorithms

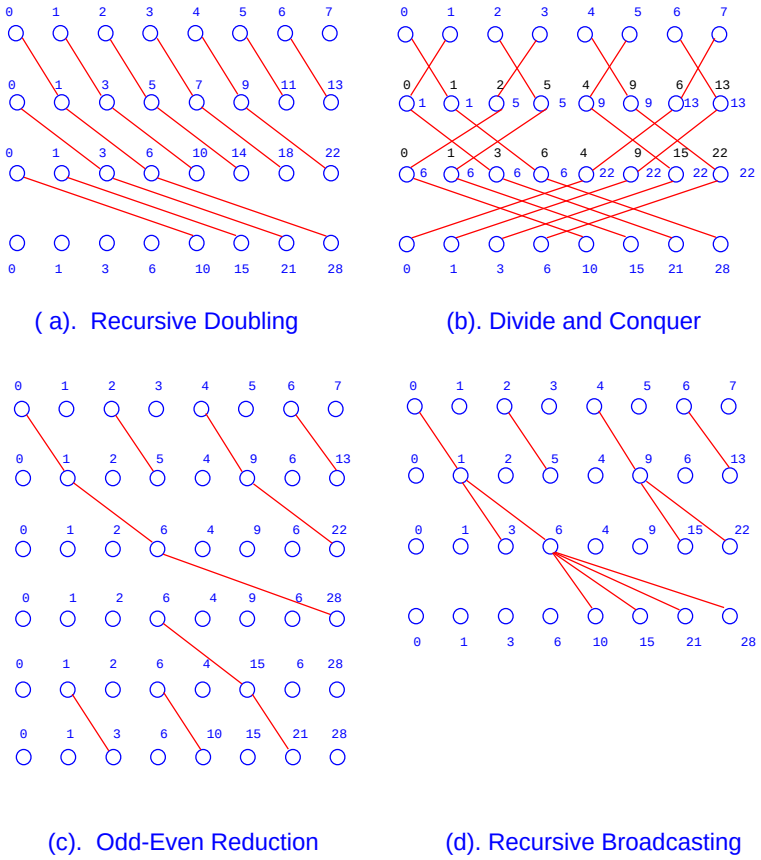

Figure 7: Scan related communication patterns on 8 PEs

for many other problems, including general reduction operations, inner product, linear recurrence, polynomial evaluation, triangular system evaluation, and software implementations of dissemination barriers [21, 24]. For parallel scans, three of the above communication patterns require one local variable per processor, while divide and conquer requires two. The best pattern to realize a scan operation depends on the particular mapping to the physical topology, and the relative cost of arithmetic, send/recv, remote copy, and split window broadcast operations. The aim is to keep all processors busy, while minimizing latency costs. For the Cray-T3E we have found that a recursive doubling communication pattern is faster; other patterns, including centralized based schemas quickly degrade with the number of PEs.

In the communication phase, depending on the sign of `partial_sum`, PEs move items either from local memory to the memory of their right neighbor (lines 45–62), or vice-versa (lines 27–45). The number of elements PE_i has to move is stored at `movNo` locally.

Items are moved, in a *pipelining* fashion. If the number of elements available in the source

BCPQ Load Balance - Communication Phase

```
   int load_balance (queue_t *vspq){
       :
       :  < Load Balance Computation Phase >
24   turn = 0;
25   while (go_on) {
26     if ((movNo > 0) && (MyId % 2 == turn)) {
27       if (partial_sum > 0) {                          /* get elements from the right */
28         pos.pe = MyId + 1;
29         movNo_now = getLocalSize(pos.pe, &cycle, vspq);
30         if (Size(cycle) > movNo) movNo_now = movNo;
31         if (movNo_now > 0){
32           pos.pos = cycle.first;
33           cutItems(pos, movNo_now, &buffer[0], vspq);
34           cycle.first = cyclicAdd(cycle.first, movNo_now);
35           setLocalSize(pos.pe, cycle, vspq);
36           pos.pe = MyId;
37           getLocalSize(pos.pe, &cycle, vspq);
38           pos.pos = cyclicInc(cycle.last);
39           pasteItems(pos, movNo_now, &buffer[0], vspq);
40           cycle.last = cyclicAdd(cycle.last, movNo_now);
41           setLocalSize(pos.pe, cycle, vspq);
42           movNo = movNo - movNo_now;
43     if (movNo == 0) done = fast_shmem_short_finc(&count_done, 0);
44         }
45       }else if (partial_sum < 0) {                    /*put elements to the right */
46         pos.pe = MyId;
47         movNo_now = getLocalSize(pos.pe, &cycle, vspq);
48         if (Size(cycle) > movNo) movNo_now = movNo;
49         if (movNo_now > 0){
50           pos.pos = cyclicSub(cycle.last, movNo_now - 1);
51           cutItems(pos, movNo_now, &buffer[0], vspq);
52           cycle.last = cyclicSub(cycle.last, movNo_now);
53           setLocalSize(pos.pe, cycle, vspq);
54           pos.pe = MyId + 1;
55           getLocalSize(pos.pe, &cycle, vspq);
56           pos.pos = cyclicSub(cycle.first, movNo_now);
57           pasteItems(pos, movNo_now, &buffer[0], vspq);
58           cycle.first = cyclicSub(cycle.first, movNo_now);
59           setLocalSize(pos.pe, cycle, vspq);
60           movNo = movNo - movNo_now;
61     if (movNo == 0) done = fast_shmem_short_finc(&count_done, 0);
62     } } }
63     barrier();
64     turn = 1 - turn;
65     done = shmem_short_g(&count_done,0);
66     if (done == (short) NPEs) go_on = 0;
67   }
68   getLocalSize(MyId, &cycle, vspq);
69   if (IsEmpty(cycle)) vspq -> nonempty = 0;
70   else vspq -> nonempty = 1;
71   return 0; /* success */
72}/*load_balance*/
```

Table 5: BCPQ `load_balance` function (communication phase)

list is smaller than `movNo` (line 30 and 48) all items from the source list are moved. Remaining items are moved in consecutive steps. To decrease congestion and avoid data races in local lists, communication is split into odd and even steps (line 26). Since synchronization of PEs in odd and even phases is done with a barrier (line 63), all PEs must loop until all items are moved. Variable `done` is used to count the number of PEs which have finished moving

items (lines 46 and 64). Variable `go_on`, initialized to 1, is set to 0 when all PEs have finished moving items (line 66). Then, all PEs can exit the pipelining loop (lines 25–67) and update locally their `nonempty` field.

3.3 Generality of the BCPQ Data Structure

The BCPQ is a general data structure which can also be viewed as a priority queue. BCPQ code can be reused in many other parallel applications with relatively small changes.

- The list access layer can form as basis for dynamic parallel insertion sort (or bucketsort) implementations.
- The list access layer can be extended to dictionary machines, by additionally supporting *find_k*, *find_min* and *find_max* operations. Since BCPQ is based on sorted lists, these operations can be implemented very efficiently.
- Set operations can be implemented efficiently using sorted lists. The list access layer can be extended to support such operations.
- Our dynamic load balance implementation is deterministic, flexible, and efficient. It could be reused in applications operating on distributed sorted lists or with diffusion based scenarios [31].
- Many dynamic applications process incoming data, e.g. convex hull in military applications. In such situations, either partial, or total order of elements must be maintained. The BCPQ is a flexible data structure for direct engagement in such problems.

4 PARALLEL SWITCH AND NETWORK SIMULATION

Serial switch simulation is considerably slow and suffers from insufficient memory. Provided that suitable data structures exist, switch simulation offers natural parallelism. Trace driven traffic on different input ports is completely independent. These Ports can be scheduled to different processors without any conflicts. Coordination is only needed to handle output port congestion.

Priority queues can be used for event scheduling. They represent the core of the event queue data structure. Events are inserted to the queue, and the next event to execute is obtained by calling DeleteMin. Execution of this event may lead to other events being created, and inserted to the event queue. A cycle-level precise ATM switch simulator, using a priority queue as an event queue data structure is presented in [9].

In this Section, we provide a theoretical evaluation of algorithmic complexity for the examined data structures; the BCPQ is viewed only as a priority queue. Then, we consider a single buffer and a multicast switch model for experimentally evaluating the performance of our parallel data structures. We finally propose a promising network simulation approach based on parallel integer sorting.

4.1 Data Structure Complexity - BCPQ as a Priority Queue?

Consider our parallel data structures when the total number of items (N) is much larger than the number of PEs ($N >> P$), so that P may be considered constant. This condition provides an asymptotic view at the concurrency of our parallel data structures.

The DPQ does not provide a uniform PE workload and offers only $\mathcal{O}(\log P)$ concurrency. The root PE must control all ADT operations and provide a (relatively prime) step for generating the round-robin sequence which allows consecutive ADT operations to proceed on different tree branches. Since ADT operations on the heap examine on the average $\log P$ local splay trees and each splay tree holds $\sim N/P$ items, the average complexity for a DPQ ADT operation is $\mathcal{O}(\log P \ \log (N/P)) = \mathcal{O}(\log P \ \log N)$ steps.

The RCPQ data structure is highly parallel and offers $\mathcal{O}(P)$ concurrency. Since, with high probability, consecutive inserts traverse disjoint paths to the root (items are mapped to random PEs), congestion occurs only at the upper $\log P$ levels of the heap. RCPQ ADT operations take on the average $\mathcal{O}(\log (N/P))$ steps.

The BCPQ data structure is also highly parallel, even if considered only as a priority queue. Since load balance distributes items uniformly, there is always a low number of access conflicts for ADT operations. Thus, BCPQ offers $\mathcal{O}(P)$ concurrency, A BCPQ data structure in random state (e.g. after N random insertions) can be load balanced in just $\mathcal{O}(P \ \log N)$ steps. This follows from the fact that for large N, a maximal $\log N$ deviation from the average size of N/P local elements per PEs is expected. On a balanced BCPQ, *DeleteMin* takes constant time. *Insert* is based on two bitonic searches and an insertion which also moves (copies) data elements. Thus, *Insert* takes $\mathcal{O}(\log P) + \mathcal{O}(\log (N/P)) + \mathcal{O}(N/P) \ = \ \mathcal{O}(N/P)$ total time.

Although our theoretical analysis claims that BCPQ insertions are slower, this will be refuted in Section 4.2. Our idealistic analysis (as most parallel time complexity analyses) places similar weights on communication and computation costs for all local and remote operations, and ignores inherent synchronization costs, such as locks and barriers. For example, nowadays in most parallel systems, local memory copies with stride 1 are highly pipelined,

e.g. using special E-registers, a strided shmem_iget achieves $\sim$ 650MBytes/sec on the Cray-T3E900. We believe that new memory and network performance models are needed to reduce the gap between theoretical and experimental evaluations of parallel algorithms. the real cost of the ADT operations is $\mathcal{O}(\log N)$.

Finally, notice that a new implementation of BCPQ as a "true" priority queue, simply by changing circular sorted lists to heaps or splay trees, requires a complex load balancing procedure. One must either use sophisticated low level remote enqueues which exploit Cray-T3E message queues (instead of simple *cut & paste* procedures), or resort to a less efficient message passing emulation. Since access to message queues is performed in Cray-T3E assembly language and requires special OS permissions, we decided not to invest further in this asymptotically optimal priority queue implementation. However, for architectures which support user control of remote enqueue operations, this priority queue implementation can possibly achieve high speed at a low implementation cost.

4.2 Experimental Switch Simulation

4.2.1 Single Buffer and Multicast Switch Models

We model a 64×64 output buffered crossbar packet switch which supports *multicasting*. For the DPQ, we will only show results for single buffer systems, due to data structure limitations on both memory space, and especially execution time.

The switch uses a central clock. Routing is synchronous and all ports operate at the same frequency. During each clock cycle, an incoming packet at each input port is routed into corresponding output buffers. Since the switch supports *multicasting*, a single packet arrival may trigger multiple packet departure events from different output buffers. If an output buffer is full, packets routed to this buffer are discarded. At the end of each clock cycle, the packet with the highest priority at each output buffer (if any) departs the switch. Hence, multicast copies may depart from output ports at different times. Since, packet departures at different output buffers are independent events, each output buffer can be modeled separately by a priority queue.

Simulation experiments were run on P ($P \in \{2, 4, 8, 16, 32, 64\}$) PEs on one of the Cray-T3E900 systems available at KFA-Jülich. Each PE simulates $\frac{64}{P}$ input ports and supervises $\frac{64}{P}$ output ports. In Figure 8, the multicast packet arriving at input port 1 is copied to output buffers 1, 3, 4, and 64. Figure 8, also shows how 32 PEs can be used to model the 64×64 switch. Input ports 1 and 2 are managed by PE_0, input ports 2 and 3 are managed by PE_1, and so on. Finally, input ports 63 and 64 are assigned to PE_{31}. Output ports are similarly

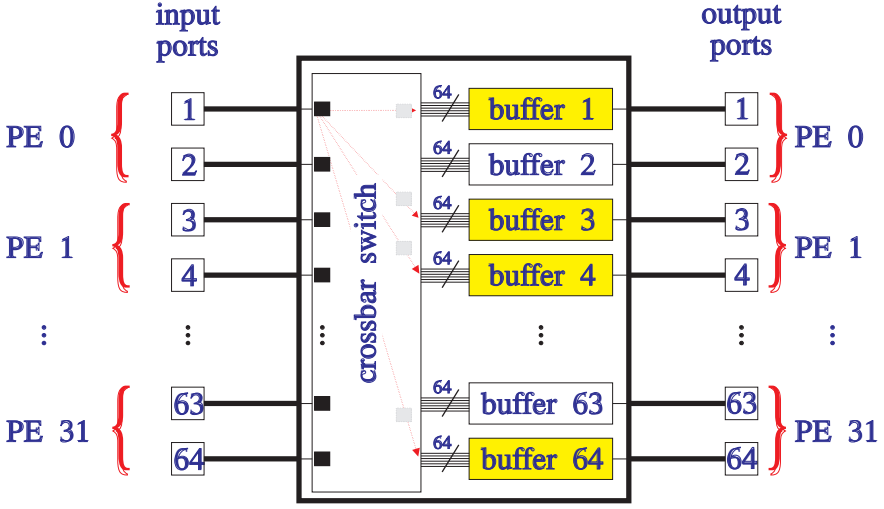

Figure 8: A 64×64 output buffered multicast switch simulated on 32 PEs

assigned to PEs. In contrast, data elements for each buffer are distributed across all PEs according to our parallel data structures. Results

4.2.2 Experimental Framework and Optimizations

Our experiments were designed to focus on data structure performance, independent of initialization overheads, e.g. arizing from `initQueue`. Thus, for all our tests, RCPQ and BCPQ allocate and initialize similar amounts of main memory. The DPQ implementation uses dynamic data allocation (and initialization) by calling `malloc` and `free` memory management operations.

In the RCPQ and BCPQ experiments, sequences (different rounds) of Insert and DeleteMin operations are separated by a barrier. The DPQ implementation is more asynchronous and does not allow one to apply exactly the same level of synchronization. Instead, Insert and DeleteMin cycles are performed in this order, but independently by all PEs. However, for all data structures, the total number of queue elements after each round is similar.

For BCPQ, we initially focused on a flexible and correct implementation. Results from an improved BCPQ version (called BCPQ Opt.) will also be shown. BCPQ Opt applies both algorithmic, and runtime optimizations. It provides a local `findPE` routine for computing the target list, thus avoiding locking. The `findPE` implementation is based on broadcasting maximum priorities of all local lists to all PEs. This all-to-all broadcast is coded using `shmem_int_p` and `shmem_quiet`, which is more efficient than using the Cray MPP `shmem_int_bcast` routine.

BCPQ Opt implements the `PEm` computation using node-to-neighbor communication (textttshmem_int_p) and parallel searching of the distributed `nonempty` field patterns. For

example, a distributed `nonempty` field of 1, 1, 1, 1, 1, 0, 0, 0 implies `PEm`$= 4$, since PE_4 is the last nonempty PE (condition 1 cf. Section 3.1). Both a Cray library *eureka-* or *shmem_event-*based parallel searching approach, and a direct `shmem_int_sum_to_all` based implementation were slower alternatives for the `PEm` computation.

Finally, optimizations have been directed at software lock mechanisms. We implemented Anderson's and MCS lock algorithms which minimize network and memory contention [24]. Improvements over the Cray `shmem_lock` library functions by MCS and Anderson's lock are above three orders of magnitude on a 64-processor Cray-T3E900 (from $\sim$ 5msec to $\sim$ 5usec).

We compiled all implementations with the highest optimization level provided by the Cray C/C++ compiler, compiler switch `-O3`. For time measurements, we use the Cray intrinsic function `_rtc()`, and divide by the number of clock ticks per second. The Cray C/C++ compiler generates inline code for each call, reducing overhead introduced by timers. We measure a) the total execution time, b) time for all insert operations, c) time for all DeleteMin operations, and for the BCPQ, d) time for all load balance operations. The latter times are easily converted to times per operation by dividing with appropriate counters.

4.2.3 Single Buffer Experiments

A synthetic priority queue application was used for our first comparisons of DPQ, RCPQ, and BCPQ data structure implementations. Insert and DeleteMin operations are performed on a single buffer modeled by a parallel priority queue. During each cycle, each PE performs one or two Insert operations, followed by one DeleteMin operation. For BCPQ, each cycle ends with a call to load balance.

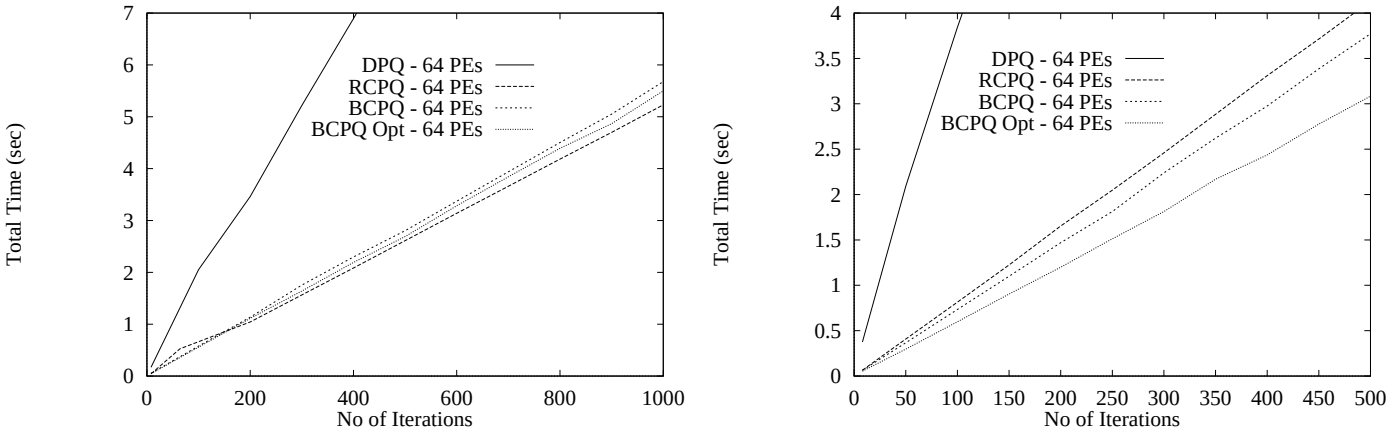

Figure 9: Runtime for single buffer: a) 1 Insert and DeleteMin, b) 2 Inserts and 1 DeleteMin per cycle)

In Figure 9 we show the total execution time vs. the number of cycles for 64 PEs. Similar results have been obtained for other numbers of PEs. From Figure 9 (a) notice that, for an equal number of Insert and DeleteMin operations, the DPQ implementation is ~ 3.5 times or more slower than our virtual shared memory implementations. For up to 150 cycles (9,600 Insert and 9,600 DeleteMin operations) BCPQ performs better than the RCPQ implementation. For a greater number of iterations the RCPQ implementation is the fastest one, but execution times of both concurrent implementations are comparable. The optimized version of BCPQ is $\sim 5\%$ faster.

Increasing the number of Insert operations per cycle makes DPQ perform much worse. It is ~ 6 times slower than our RCPQ and BCPQ implementations (see Figure 9 (b)). Since the initial implementation of the Insert function in the BCPQ is slower than the corresponding RCPQ implementation, RCPQ appears slightly faster. However, the optimized BCPQ version with a local `findPE` routine is $\sim 25\%$ faster than RCPQ.

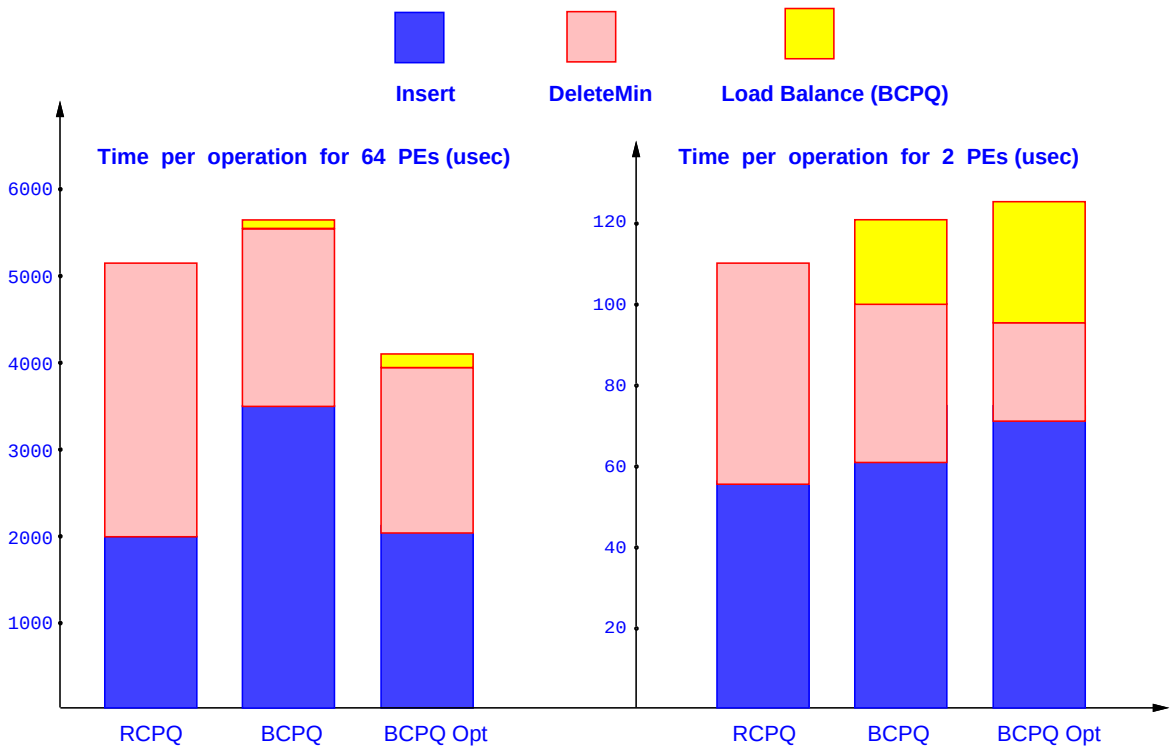

Figure 10: Time per operation for RCPQ and BCPQ implementations (2 Inserts and 1 DeleteMin per cycle)

Figure 10 compares times per Insert, DeleteMin, and load-balance operations for the

RCPQ and BCPQ implementations, for 2 and 64 PEs; typically 500 iterations were used. Both implementations show comparable per operation execution times. The optimized version of BCPQ executes DeleteMin operations faster than RCPQ, because DeleteMin operations on heaps require reheapification. For 64 PEs, Insert time for the optimized BCPQ version decreases, and approaches that of the RCPQ implementation. For BCPQ, load balance is faster and scales better than Insert and DeleteMin operations. The relative time spent for load balance on 2 PEs is relatively larger, compared to the other two ADT operations, due to a constant computation overhead which shows a stronger impact if the total execution time is smaller (notice that the vertical scales are different).

Thus, single buffer experiments indicate that DPQ is always much slower than RCPQ and BCPQ. Using MPI instead of low level shared memory libraries significantly deteriorates performance. The decision for a programming model together with an appropriate library has a major impact on application performance. Before deciding on a library, one must also consider if portability is really needed, or expected.

4.2.4 Multicast Switch Experiments

For our switch experiments, we used a constant input traffic model with fixed average multicasting rate. The arrival rate was set to one packet per cycle. Independently chosen random values for priorities and packet destinations were used.

After each clock cycle, the BCPQ data structure is dynamically load balanced. Barriers are used after each operation to satisfy our synchronous switch requirement. Users can further improve BCPQ performance by providing a threshold condition for load balance. This condition will normally depend on the application, and can be implemented either as fuzzy logic, or by using neural network learning techniques. Notice that, no load balancing degrades BCPQ performance severely; in our experiments, a BCPQ which does not invoke load balance performs ~ 5 times slower.

Performance and scalability experiments for the 64×64 crossbar switch, with multicasting rates of 1 (unicast) and 64 (broadcast) are shown in Figures 11 and 12.

In Figure 11, we show the total execution time vs. the number of simulation cycles for unicast or broadcasting with 64 PEs; similar results have been obtained for other numbers of PEs and multicast rates. Both concurrent implementations perform almost linearly to the number of simulation cycles. By comparing the two graphs, we see that the total execution time slightly increases with the number of items stored in the priority queue.

For unicast, as many items are removed from the priority queues as are inserted, thus

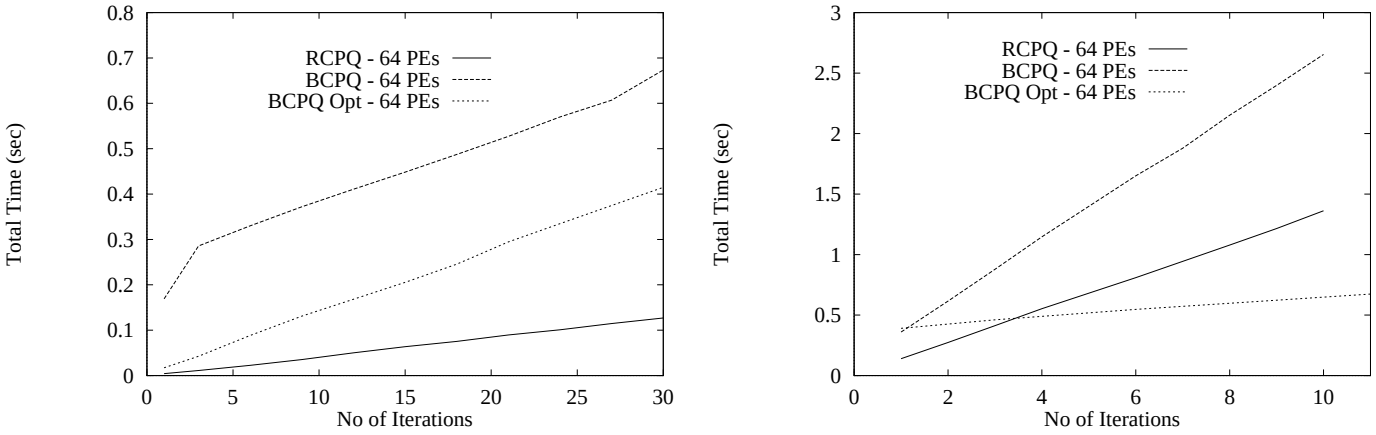

Figure 11: Runtime for crossbar switch (unicast and broadcasting)

keeping priority queues always empty between consecutive simulation cycles; this would correspond to circuit switching, or hot-potato routing protocols, e.g. for simulating optical communication networks. Thus, in this case, load balancing the BCPQ is not needed. As shown in Figure 11 (a), for all implementations the execution time scales well with the number of simulation cycles. Also, RCPQ outperforms both BCPQ versions. This is partially due to an overhead associated with maintaining maximum list priorities for the `findPE` computation; for unicast, this extra work is largely unnecessary, since on the average buffers contain very few elements. To speed up BCPQ, one can rely on the fact that a simpler DeleteMin implementation is now possible; the item with the highest priority for each queue is always stored at the first position at the local list of PE_0. Furthermore, one can precompute a random assignment of each distributed queues to PEs, so that the "minimum" local list for any given queue is shuffled across all PEs. This would remove network and memory contention at PE_0 during concurrent DeleteMin operations to different queues, at the expense of some local only computation.

For broadcasting (see Figure 11 (b), although the number of items at each queue increases a lot between consecutive simulation cycles, the total time changes only slightly. The optimized BCPQ version exhibits better scalability and superior performance to RCPQ for any reasonable number of simulation cycles. Thus, with the dynamic load balanced BCPQ data structure, the overhead of handling priority queue ADT operations on tens of thousands of data items is relatively small.

For a fixed problem size, Figure 12 reveals interesting scalability issues of our concurrent implementations. We consider unicast and broadcasting on a 64×64 crossbar switch, with 2, 4, 8, 16, 32, and 64 PEs. Figure 12 shows an optimal number of PEs for both BCPQ

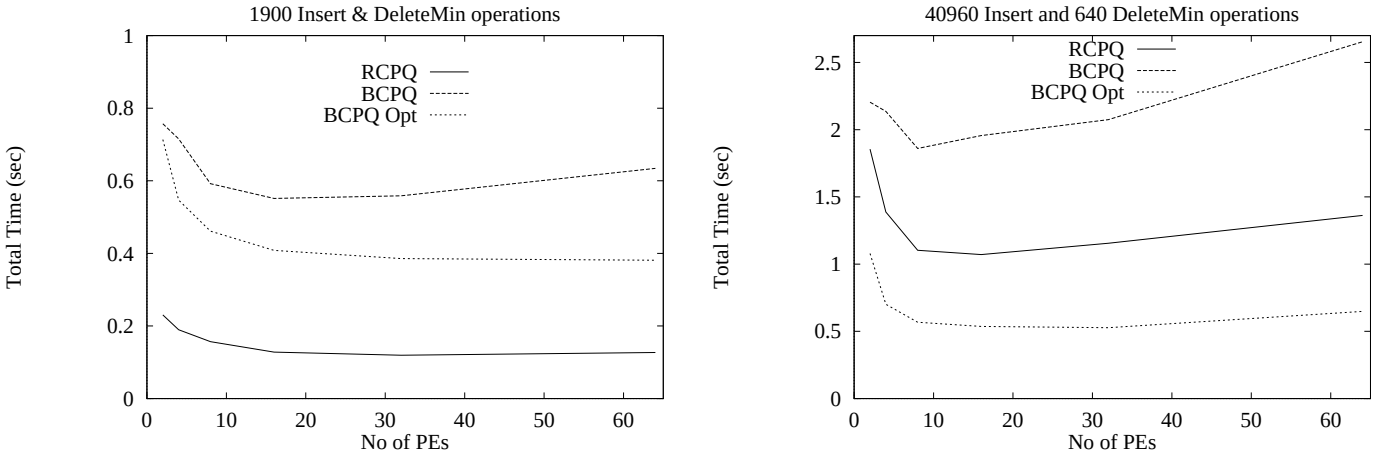

Figure 12: Scalability for crossbar switch (unicast and broadcasting)

implementations, around 16 PEs. Before this point, the total execution time is larger because each PE must simulate more inputs and output ports. After this point, the execution time increases due to a shift from computation towards communication work; for the optimized BCPQ this trend is obviously less profound. For broadcasting, the optimal number of PEs decreases to approximately 8, thus indicating a dependence on the number of more expensive insert operations. For both cases, RCPQ seems to saturate faster, possibly because of increased network contention.

Hence, our experimental analysis based on packet switch simulation experiments exhibits good performance for virtual shared memory implementations, especially for the optimized BCPQ version.

4.3 Experimental Network Simulation

We have also examined priority queues for network simulation. For example, packets in a butterfly network can move from stage to stage by issuing repeated Insert and DeleteMin operations on different priority queues. In our preliminary experiments with medium size networks ($\sim$ 128 nodes) we discover that, BCPQ performs better with large switch configurations (e.g., 8×8 crossbars) and intensive, nonuniform traffic conditions (e.g. multicast, hot spot traffic, or asynchronous links). RCPQ is more efficient with smaller switch configurations and lower traffic requirements (e.g. unicast, or synchronous links). Precise performance tradeoffs are still under investigation.

For very large symmetric networks, such as the butterfly, the hypercube, and the torus, we now propose a conservative parallel simulation approach which detects packet collisions in the simulated network by sorting a conflict function (f). The function f is essentially

an edge-congestion function. A similar f value indicates packets which compete over the same communication link. In this case, only the packet with the highest priority can be routed, while remaining packets may be stalled, deflected, or dropped depending on the underlying communication protocol. A nonoptimized implementation of this approach moves (during sorting) not only the conflict function array, but also all corresponding packet status information among PEs.

An optimized version avoids unnecessary relocation of packet information and *updates all packets in place* by relying on bit-vectors and a global packet indexing scheme. Initially, an index array stores at each PE a unique number for each packet; e.g. PE_0 may number packets as $[0, \ldots, N/P-1]$, PE_1 as $[N/P, \ldots, 2N/P-1]$, ... where N is the total number of packets. During sorting, we allow only the conflict-function and the corresponding index array to be exchanged among PEs; other packet information, such as origin, current, and destination node, packet delays, distance from destination, and number of deflections are never exchanged. After sorting, each PE computes a bit-vector of size N based on local conflict-function information. The bit-vector at position I is set to *True* iff the PE: a) contains I in its index array, and b) the corresponding conflict-function and neighbor conflict-function values are different, i.e. the packet is able to move. The overall bit-vector can now be computed using a (bit-wise OR) MPI_Allreduce, and the update phase can be done in place, i.e. all simulated packets actually reside in the same location in the PEs' memory during the entire simulation.

On top of our bit-vector optimization, we have also used (as before) the `-O3` Cray-T3E compiler option. This option performs aggressive automatic scalar optimization, including fast divide using multiply-by-reciprocal, cache alignment, loop alignment, and loop unroll. Also, automatic inlining is performed to avoid function call overhead.

As an example, we have chosen to simulate a synchronous binary hypercube topology. In our experiments, each hypercube node generates a packet with a single random destination, and routing is performed by correcting bits in right to left order. In cases of traffic congestion, packets with larger network delays are always given priority. We focus on the time required for realizing a random communication pattern versus the number of PEs, for different numbers of packets. Dynamic experiments with intensive communication patterns (such as bit reversal and broadcasts) have also been used during the testing phase.

We considered bitonic sorting, odd-even transposition, merge-bitonic sort (intra-processor mergesort on N/P integers followed by inter-processor bitonic sort), quicksort-bitonic, and parallel sorting based on insertions to the BCPQ data structure. Although samplesort is on

the average faster than the above algorithms, we did not use samplesort because it results in a different number of integers at each PE [11]; this is inefficient for network simulation, since we use indirect indexing to access most of the array structures. Our experiments indicated that merge-bitonic is the best choice for parallel integer sorting [6]. BCPQ was measured about 5 times slower.

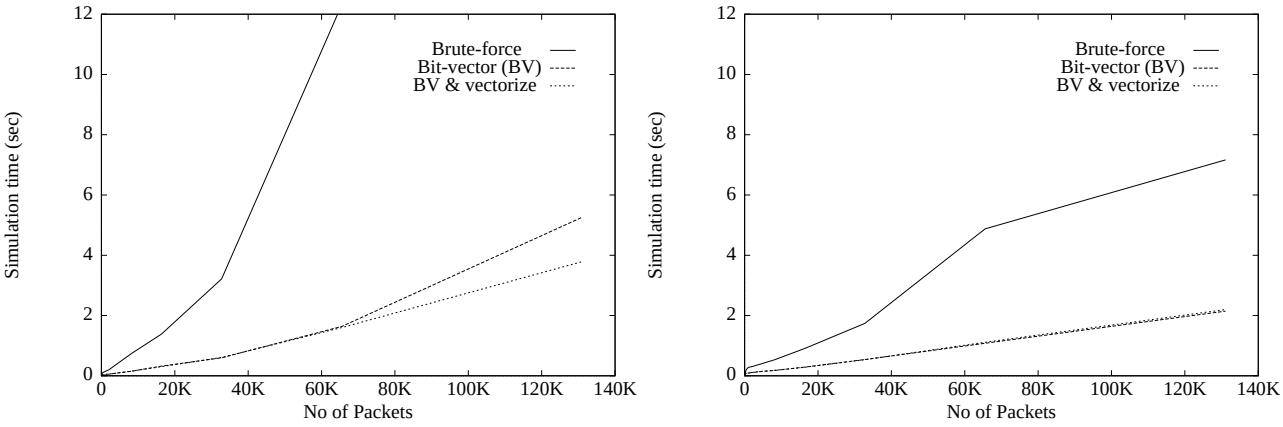

Figure 13: Simulation time on Cray-T3E with: (a) 32 PEs, and (b) 128 PEs.

Using merge-bitonic sorting, in Figure 13 we compare the Cray-T3E simulation time for all three versions (nonoptimized, bit-vector optimization (BV), and bit-vector plus compiler (BVC) optimizations) for 32 and 128 PEs, respectively. Both graphs indicate that the savings in simulation time are very important. For 128 PEs, simulation is between 2.2 and 4.5 times faster for both BV, and BVC optimization. For 32 PEs, simulation is between 2.9 and 5.8 times faster for BV optimization, while they are between 3.1 and 8.1 times faster for BVC optimization; the additional performance gain obtained from compiler optimization is due to the larger size of data that must be sorted and processed locally at each processor.

In the nonoptimized version $\sim 95\%$ of the simulation time is spent on sorting and relocating packet information, while only 5-10% is spent on reductions, permutations and statistical time-keeping routines. The bit-vector approach reduces sorting time to only 45-60% of the total simulation time. For a very large number of packets the scalability remains very good. 1M packets ($\sim$ 20M moves) can be simulated using the BVC algorithm on the Cray-T3E with 32 PEs in just 43.3 sec.

This performance exceeds by almost three orders of magnitude that achieved by commercial simulators (e.g. BONeS) on the same problem. We have not attempted to compare our network simulation approach (presented in this Section) with other sequential or parallel C/C++ network simulation engines, e.g. C-Sim, NS, TeD, and TopSim. However, our achieved rate of $\sim$ 500K packet moves per sec, in a very large (1M) node network, while sup-

porting a priority-based routing protocol and maintaining consistent 17 different information and statistics arrays for each network packet, is currently competitive.

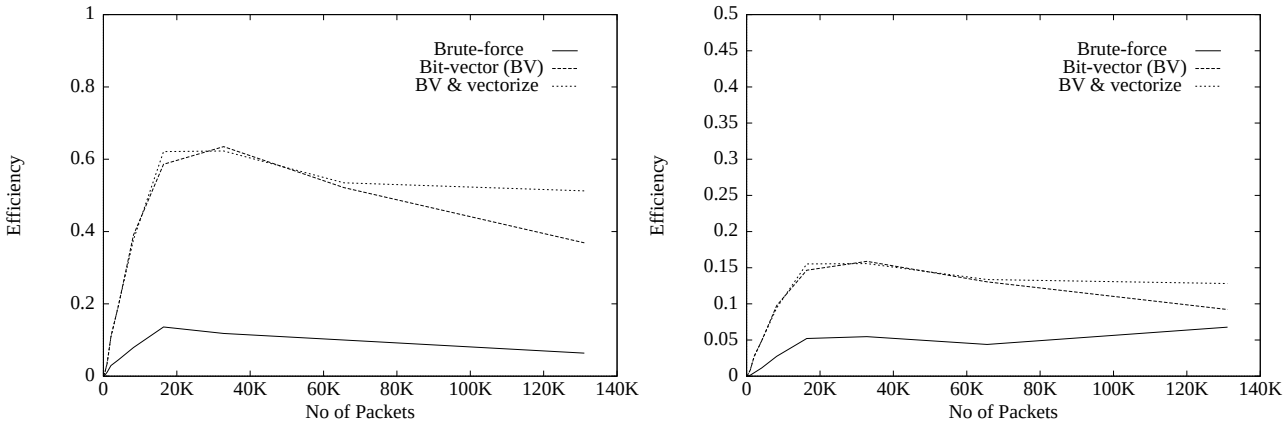

Figure 14: Parallel efficiency on Cray-T3E with: (a) 32 PEs, and (b) 128 PEs.

In Figure 14 we compare the Cray-T3E parallel efficiency vs. the number of packets, for 32 and 128 PEs, respectively. The efficiency is $E = S/P$, and the speedup $S = T_1/T_P$, where T_1 is the best sequential execution time, and T_P is the parallel algorithm execution times. Notice that, efficiency is above 50% for 32 PEs, but only above 13% for 128 PEs. This means that one should preferably use 32 PEs when running experiments with more than a few tens of thousands of packets.

5 CONCLUSIONS AND EXTENSIONS

We have provided fast and flexible implementations of distributed and concurrent data structures which form a library for further applications in device and network simulation. Our implementations consist of $\sim 8{,}000$ lines of documented C code ($\sim 2{,}050$ DPQ, $\sim 1{,}100$ RCPQ, $\sim 1{,}750$ BCPQ and $\sim 3{,}000$ for various switch/network simulator applications).

We have simulated single buffers and also 64×64 multicast-based switches using up to $60{,}000$ packets and 2 to 64 PEs. Our experiments on a Cray-T3E system at KFA-Jülich indicate that concurrent priority queues are 5 to 10 times faster than distributed ones, even though the distributed DPQ implementation uses asynchronous insertions and deletions. Our two concurrent data structure implementations, based on predetermined random mapping of a heap data structure (RCPQ), and dynamically load balanced distributed sorted lists (BCPQ) achieve comparable performance, although BCPQ is more efficient in assigning memory to PEs.

We have also proposed a novel approach for parallel discrete-event simulation of symmetric

communication networks. Our algorithm resolves packet conflicts by sorting integer conflict functions. We have implemented our methods initially on CM-5, Cray-T3D, and later Cray-T3E systems using C+MPI or C+ShMem, and performed critical improvements aimed at reducing sorting overhead, minimizing inter-processor communication, and optimizing scalar processing. Performance results for a packet-switched hypercube topology indicate that our parallel simulation approach achieves good scalability and efficiency; our optimized simulator can process $\sim 500K$ packet moves in 1sec, with an efficiency that exceeds $\sim 50\%$ for a few thousands packets on the Cray-T3E with 32 PEs.

Both concurrent data structures are easy to use from their application layer. There are no application restrictions or side effects arising from the core data structure, and they can can be extended and improved in various ways. Although, there is already upward compatibility of ShMem routines on new Cray systems, we expect that MPI-2 will open new horizons for extensions of this work. While the data access layer will have to be rewritten, most of the main part of the code can remain unchanged. An MPI-2 version would essentially make our source code portable on a variety of hardware platforms.

Our work raises interesting practical and theoretical questions on all data structures and algorithms.

- An improvement to DPQ could reduce the workload imbalance which exists at root PE (and the upper heap levels) during ADT operations. While this could improve fault tolerance, software complexity would also increase considerably.

- To reduce memory and network traffic contention, the centralized total items counter (accessed during concurrent ADT operations) in RCPQ and BCPQ could be implemented by emulating a *counting network*. *Counting networks* provide a distributed mechanism for implementing *Fetch&Increment*, by reducing bottlenecks associated with accessing a single (virtual) shared memory location [2, 12].

- Our RCPQ implementation uses a random node distribution to PEs. A complex distribution which takes into account parent-child communication patterns could reduce latency during heapification. However, one must notice that the Cray-T3E may use adaptive message routing, and thus deterministic allocation could have its pitfalls.

- The BCPQ implementation also allows low level runtime optimizations. The communication overhead can still be reduced by experimenting with different ShMem subroutines, access patterns, and physical PE allocation schemes. We have noticed an interesting tradeoff between keeping up to date all system variables at all times and

minimizing program latency. In this respect, the minimum amount of redundancy which provides for a consistent and efficient data structure is an open question.

- In some processors, e.g. the MC680x0 series, a double Compare&Swap atomic operation is available. Two separate words are compared with test values, and if both words match the test values they are replaced with new values. The asynchronous BCPQ data structure can be transformed to be lock-free [13]. The *double Compare&Swap* operation allows updating both head, and tail pointers at once, while testing at the same time if these pointers have remained unchanged.

- A rigorous analysis on issues such as time and space complexity, and runtime semantics for asynchronous ADT operations, remains an interesting open question. To obtain any realistic time complexity results, a model on the cost of virtual shared memory accesses for different remote operations is needed. Reverse profiling could prove a valuable method in this direction [14].

ACKNOWLEDGMENTS

Our research has been supported by KFA-Jülich (project INTENS), and partially supported by EPCC-Edinburgh (project EU ESPRIT TMR TRACS). The majority of this research was carried out while the authors were with the Institute of Informatics at the University of Hildesheim. Several students have also helped in simulation projects: Helidon Dollani (Technical University of Graz), Torsten Merker (Hewlett-Packard, Hamburg), Nikos Fideropoulos and Alexandros Zachos (Technical University of Braunschweig).

We are grateful to Prof. Michael Scott and Dr. Bernard Mans for their critical help at initial stages of this work. They kindly provided free access to CPQ pseudocode and preliminary DPQ code. We would like to also thank an anonymous referee for the insightful comments which have helped our presentation.

References

[1] S. Arora, F.T. Leighton, and B.M. Maggs. On-line algorithms for path selection in a nonblocking network. *SIAM J. Comput.*, **25** (3), pp. 600–625, 1996.

[2] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, **41** (5), pp. 1020–1048, 1994.

[3] H. Attiya, and R. Friedman. Programming DEC-Alpha based multiprocessors the easy way. *Proc. 6^{th} ACM Symp. Parallel Alg. Arch.*, pp. 157–166, 1990.

[4] G.S. Brodal, J.L. Träff, J. L., and C.D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, **C-49** (1), pp. 4–21, 1998.

[5] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. An alternative to Fibonacci heaps with applications to parallel computation. *Comm. ACM*, **31** (11), pp. 1343–1354, 1988.

[6] M.D. Grammatikakis, N. Fideropoulos, F. Howell, S. Liesche, T. Thielke, A. Zachos, Network simulation on the CM-5 by sorting integer conflict functions. *Proc. Parallel Computer Conference*, Bonn, Germany, 1997. Parallel Computing: Trends and Applications, Elsevier Science Publishers, to appear.

[7] M.D. Grammatikakis, N. Fideropoulos, and A. Zachos. Network simulation on Cray-T3E using MPI. *3^{rd} Cray-SGI MPP conf.*, Paris, France, 1997. Available from http://armoise.saclay.cea.fr/∼workshop/Program.html

[8] M.D. Grammatikakis, D.F. Hsu, M. Kraetzl, and J. Sibeyn. Packet routing in fixed-connection networks: a survey. *J. Parallel Distrib. Comput.* Academic Press, 77–132, 1998.

[9] M.D. Grammatikakis, and M. Johl. Clock-cycle level simulations of an ATM Switch. *Proc. 1^{st} SCS Euro Media Conf*, London, England, pp. 149–156, 1996.

[10] M.D. Grammatikakis, and S. Liesche. Synchronization on Cray-T3E virtual shared memory. *40^{th} Cray Users Group conf.*, Stuttgart, Germany, 1998. Available from http://theoretica.informatik.uni-oldenburg.de/ mdgramma/mutex.ps.gz

[11] D.R. Helman, D. Bader, and J. JáJá, Parallel algorithms for personalized communication and sorting with an experimental study. *Proc. ACM Symp. Par. Alg. Arch.*, pp. 211–220, 1996.

[12] M. Herlihy, B.H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Trans. Comput. Syst.*, **C-13** (4), pp. 343–364, 1995.

[13] M. Herlihy, and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. *Proc. Int. Symp. Comp. Arch.*, San Diego, USA, pp. 289–301, 1993.

[14] F.W. Howell. Reverse Profiling. *Proc. 1st Int. Workshop Parallel Distrib. Soft. Engin.*, Berlin, Germany, pp. 245–255, 1996.

[15] G.C. Hunt, M. Michael, S. Parthasarathy, and M.L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Proc. Letters*, **60** (3), pp. 151–157, 1996.

[16] D.W. Jones. Concurrent operations on priority queues. *Comm. ACM*, **32** (1), pp. 132–137, 1989.

[17] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching. Addison-Wesley*, 1973.

[18] F.T. Leighton, D. Lisinski, and B.M. Maggs. Empirical evaluation of randomly-wired multistage networks. *Proc. IEEE Int. Conf. Comp. Design*, pp. 380–385, 1990.

[19] B.M. Maggs. Randomly-wired multistage networks. *Stat. Sci.*, **C-8** (1), pp. 70–75, 1993.

[20] B. Mans. Portable distributed priority queues with MPI. *Concurrency: Practice and Experience*, **10** (3), pp. 175-198, 1998.

[21] Z.G., Mou, and M. Goodman. A comparison of communication costs for three parallel program paradigms on hypercube and mesh architectures. *Proc. 5th SIAM Conf. Parallel Proc. Sci. Comput.* pp. 491–500, 1992.

[22] MPI-2: Extensions to the message-passing interface. *MPI Forum*, July 1997.

[23] S. Liesche. MPI and shared memory implementations of priority queues for parallel simulation on Cray-T3E. *Diplomarbeit, Institute of Informatics, University of Hildesheim, D-31141 Hildesheim, Germany*, May 1998. Available from: http://members.aol.com/liesche/darbeit.zip

[24] J.M. Mellor-Crummey, and M.L. Scott. Algorithms for scalable synchronization on shared memory multiprocessors. *ACM Trans. Comput. Syst.*, **C-9** (1), pp. 21–65, 1991.

[25] D. Peleg. Distributed data structures: a complexity-oriented view. *Proc. Int. Workshop Distr. Alg., Lect. Notes Comput. Sci.*, Springer Verlag, **486**, pp. 71–89, 1991.

[26] V.N. Rao, and V. Kumar. Concurrent access of priority queues. *IEEE Trans. Comp.*, **C-37** (12), pp. 1657–1665, 1988.

[27] P. Sanders. Fast priority queues for parallel branch-and-bound. *Int. Workshop Alg. Irreg. Struct. Prob., Lect. Notes Comput. Sci.*, Springer Verlag, **980**, pp. 379–393, 1995.

[28] P. Sanders. Randomized priority queues for fast parallel access. *J. Parallel Distrib. Comput.* **C-49** (1), pp. 86–97, 1998.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *Proc. 16^{th} ACM Symp. on OS Princ.*, Saint-Malo, France, pp. 26–37, 1997.

[30] D. Sleator, and R.E. Tarjan. Self-adjusting binary search trees. *SIAM J. Comput.*, **15** (1), pp. 52–69, 1986.

[31] R. Subramanian, and I.D. Scherson. An analysis of diffusive load-balancing. *Proc. ACM Symp. Par. Alg. Arch.*, pp. 220–225, 1994.

[32] J. Turner, and N. Yamanaka. Architectural choices in large scale ATM switches. *IEICE Trans.*, to appear, 1998.

[33] J.W. Williams. Algorithm 232: Heapsort. *Comm. ACM*, **7**, pp. 347–348, 1964.

6 APPENDIX

The Cray-T3E implements a logically shared address space over physically distributed memories (with up to 2GB per processor).

Several atomic, synchronization, and collective operations are supported on all Cray MPP and PVP systems by calling Cray ShMem library functions. ShMem library calls can be used within Fortran and C/C++ programs. Since our implementations are in C only relevant C/C++ routines are presented.

The Cray-T3E provides a plethora of atomic operations on arbitrary memory locations, allowing an unlimited number of synchronization variables. The atomic operations provided by the Cray-T3E are: Fetch&Inc, Fetch&Add, Compare&Swap and (masked) Swap. Although Test&Set is not provided, it can be easily implemented using Fetch&Store, or Compare&Swap.

Generic Name	ShMem Function	Description
Fetch&Inc	`shmem_short_finc(`$\&a$`,`n`)`	return $M_n(a)$; $< M_n(a) = M_n(a) + 1 >$
Fetch&Add	`shmem_short_fadd(`$\&a$`,`b`,`n`)`	return $M_n(a)$; $< M_n(a) = M_n(a) + b >$
Fetch&Store	`shmem_swap(`$\&a$`,-1,`b`,`n`)`	return $M_n(a)$; $< M_n(a) = b >$
Compare&Swap	`shmem_short_cswap(`$\&a$`,`b`,`c`,`n`)`	if $M_n(a) \neq b$: return $M_n(a)$ else : return $M_n(a)$; $< M_n(a) = c >$
(Masked) Swap	`shmem_short_mswap(`$\&a$`,`b`,`c`,`n`)`	return $M_n(a)$; $< M_n^i(a) = c^i >$, $\forall i \in \{0, \ldots, 31\} : b^i = 1$
Remote Read	PE$_p$: `shmem_get(`$\&a$`,`$\&b$`,`c`,`n`)`	$< M_p(a) = M_n(a); \ldots;$ $M_p(a + c - 1) = M_n(b + c - 1) >$
Remote Write	PE$_p$: `shmem_put(`$\&a$`,`$\&b$`,`c`,`n`)`	$< M_n(a) = M_p(b); \ldots;$ $M_n(a + c - 1) = M_p(b + c - 1) >$

Table 6: Atomic read-modify-write and remote read/write in ShMem

Atomic operations available to users and operating system programmers on the Cray-T3E are shown in Table 6. The operands are generally short (32-bit) integers; some operations allow also for other data types, such as int, long, float, and double. A definition of each operation listed above is provided below:

Fetch_$\Phi(a, b)$ operations return the memory contents $M(a)$, while storing at location a the function $\Phi(M(a), b)$.

Compare&Swap(a, b, c) primitives atomically compare the content of memory location

$M(a)$ with a replacement value b, and store a third value c if they match; the operation also returns a condition flag indicating either success, or failure.

(masked) Swap(a, b, c) operations store selected bits from c in $M(a)$. Selection is done by the mask b. It returns the previous content of $M(a)$.

Shared memory (ShMem) locks provide atomicity during parallel update operations, while barriers help in avoiding certain race conditions. Synchronization barriers allow a set of participating processors to determine when all processors have signaled some event (typically reached a certain point in their execution of a program). Barrier routines, e.g. (`shmem_barrier_all()`) or simply (`barrier()`, are implemented in hardware on the Cray-T3E using combining trees. Also, *shmem_set_lock(a)* and *shmem_clear_lock(a)* routines together provide mutual exclusion. The first call to *shmem_set_lock (a)* (acquire lock) returns immediately. A subsequent call returns after a *shmem_clear_lock(a)* (release lock) corresponding to the first *shmem_set_lock (a)* is executed. This ensures that only one process is holding the lock at a time.

Cache coherence is implemented on the Cray-T3E using cache-invalidate protocols. However, the T3E is not stream coherent; prefetching must be turned off to avoid data race conditions which may cause inconsistent results, program aborts, or hangs. Due to adaptive packet routing, the Cray-T3E may also reorder instructions from a given PE. Since successive calls to `shmem_put` are not guaranteed to arrive in order, special calls to `shmem_quiet` (or `shmem_fence`) are always needed to verify that all previous puts from a particular PE to all other PEs (respectively, to a given PE) have executed in order. Global synchronization barriers can also be used, but one must be careful in avoiding deadlock problems; this will become clear in our discussion of a parallel Scan implementation in 4.

Biographies:

Miltos D. Grammatikakis, received MSc (1985) and PhD (1991) in Computer Science from the University of Oklahoma, USA. He was a postdoctoral fellow in the parallel processing laboratory of ENS-Lyon (1991-1992), researcher in the Institute of Computer Science, F.O.R.T.H., and lecturer at TEI Technological College, both in Heraklion, Greece (1992-1995). He served as scientific assistant in Computer Science (C1) at the University of Hildesheim (1995-1998), and the University of Oldenburg (1998-1999). He recently joined INTRACOM S.A. in Athens, Greece as software engineer. His research interests include discrete-event simulation, and parallel communication systems.

Stefan Liesche received his Diplom in Computer Science from the University of Hildesheim in 1998. During his studies his major interest was in the area of parallel and distributed algorithms. His thesis work on "MPI and shared memory implementations of priority queues for parallel simulation on Cray-T3E" was supported by an EU TMR/TRACS scholarship. Since Summer 1998, he is working in commercial software development at the IBM research laboratory in Böblingen.