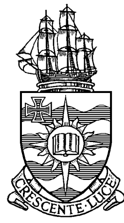

JAMES COOK UNIVERSITY
OF
NORTH QUEENSLAND

Portable Distributed Priority Queues with MPI

Bernard Mans

TR 95/09

DEPARTMENT OF COMPUTER SCIENCE

TOWNSVILLE
QUEENSLAND 4811
AUSTRALIA

Title	Portable Distributed Priority Queues with MPI
Primary Author(s)	Bernard Mans
Contact Information	Department of Computer Science James Cook University Townsville, Qld 4811 Australia Fax: 61-77-814029 bernard@cs.jcu.edu.au http://www.cs.jcu.edu.au/~bernard/
Date	December 1995

Copyright © 1996 Bernard Mans. All rights reserved.

Abstract

Part of this work has been presented in [17].

This paper analyzes the performances of portable distributed priority queues by examining the theoretical features required and by comparing various implementations.

In spite of intrinsic bottlenecks and induced hot-spots, we argue that tree topologies are attractive to manage the natural centralized control required for the *deletemin* operation in order to detect the site which holds the item with the largest priority. We introduce an original perfect balancing to cope with the load variation due to the priority queue operations which continuously modify the overall number of items in the network.

For comparison, we introduce the d-heap and the binomial distributed priority queue. The purpose of this experiment is to convey, through executions on Cray-T3D and Meiko-T800, an understanding of the nature of the distributed priority queues, the range of their concurrency and a comparison of their efficiency to reduce requests latency. In particular, we show that the d-heap combines an adjustable degree with a small depth, which make it efficient in both theory and practice. The Message Passing Interface (MPI) provides us with an adequate portable code.

Keywords Cray-T3D, Distributed Data Structure, Message-Passing Interface, Meiko-T800, Priority Queue.

1 Introduction

Priority queue is recognized as a useful abstraction due to the wide variety of computer applications in which it arises. As defined by Knuth [12], a priority queue is an abstract data structure consisting of a set of m items: a data item i is composed of a pair of field $(Key_i, Record_i)$, where Key_i is a numerical value from an ordered domain, and $Record_i$ is a record field containing the data. The basic operations are:

insert(i,h) : insert a new item i with predefined priority Key_i into priority queue h,

deletemin(h) : delete an item of minimum value (highest priority) from h and return it.

Numerous subtle priority queue implementations have been investigated to improve specific applications [10]: implicit heap [31], leftist heap [4], binomial queue [2], skew heap [29], etc.

In parallel, efficient applications require that several processors should be able to manipulate the priority queue simultaneously. Various efforts to implement concurrent accesses for the basic operations for tightly-coupled multiprocessors have been achieved successfully. We do not present an exhaustive bibliography of these implementations in this paper; some implementations and surveys are given in [5, 6, 11, 15, 26]. However, the availability of an addressable shared-memory, virtual or not, limits the concurrency problem to a trade-off between reducing the contention for shared priority queue and decreasing the number of processor synchronizations.

In a loosely-coupled multiprocessors model, where processors are distributed among a network and communicate only through message-passing, the problem of an efficient implementation of the global data structure has to be re-considered. There is no common memory, and algorithms are event-driven (i.e., processors can not access a global clock in order to decide on their action).

In this model, a **distributed data structure** (denoted *DDS*) is a data structure which is distributed among the sites of a communication network and may be accessed by many processes simultaneously, [25]. It is composed of a data organization that specifies a collection of local data structures storing copies of data item at various sites in the network, and of a set of distributed access protocols that enables processes to issue queries and modification instructions to the

network and to get appropriate responses. To be efficient, a distributed data structure must minimize the communication cost while requiring reasonable overall space and balancing the memory loads over the sites of the network. We do not consider here implementation with some specific processors managing the data structure for other distributed working processors (our purpose is neither a dictionary machine nor a VLSI implementation; these fields have been intensively explored [1, 7, 20, 23]).

In this paper, we deal specifically with **priority queue** data structures–they supports exclusively the operations *insert* and *deletemin*. Compared with data structures used for other applications, such as Distributed Databases or Dictionary machines, the main difference arises from the natural centralized control required to detect the site which holds the item with the largest priority. For this reason, the tenet that a tree topology is not appropriate (by creating hot-spots and bottlenecks) no longer makes sense for distributed priority queues.

We have developed a simple portable **distributed priority queue** (denoted *DPQ*) which allows extensions. By portable we mean that the implementation code of the DPQ does not need to be modified for the targeted platform: programs written with the Message Passing Interface (*MPI*) [21], and several users parameters are provided to adapt efficiently the logical structure to the physical network. The MPI standard provides source-code portability of message-passing programs written in C or Fortran across a variety of architectures. In this preliminary approach, we detect the impact of the structural factors (both physical and logical) on each DPQ. Our major goal is the design and analysis of concurrent methodologies which point out the relationship between efficiency and scalability. The next phase, not considered here, consists of supporting Fault-tolerance (through replication and lazy update [9, 14]) and topologically-dynamic network, i.e., with a growing or decreasing number of processors [13]. Simultaneously, the DPQ is used to solve some scientific problems requiring high performance computing, e.g., Branch and Bound for Combinatorial Optimization problems, Event-Set model for Discrete Event Simulation, etc.

The rest of the paper is organized as follows. Section 2 gives an overview of the problems and constraints raised by distributing a data structure, and more specifically a priority queue. Section 3 presents the distributed priority queue implementation based on tree topologies. Section 4 discusses the experimentations results on the Cray T3D and the Meiko T800 Computing Surface. Finally, Section 5 contains some conclusive remarks and presents detailed perspectives.

2 Distributed Data Structures

Networks of processors with the message passing paradigm are increasingly popular for parallel computing implementations. Reasons for this are mainly due to the availability of autonomy, the ease of scalability, the increased of reliability, and the ability to incorporate heterogeneous resources. The centerless organization of the control in the network arises different design issues. This section outlines the different features of efficient distributed data structures. The specificity of distributed priority queues is outlined when exists.

2.1 Distributed computing model

In this paper, we are dealing with arbitrary communication network. We do not make assumptions on the topology in a sense that the priority queue must be efficient, independently of the targeted network. All processors have distinct identities. Messages arrive within finite but unpredictable time and in a FIFO order. The network communication is free of errors.

In all applications using a distributed data structure, the computing model is the following: the processors eventually generate data that need to be stored in the global structure and may be needed by other processors. Each processor runs two intertwined processes: one for

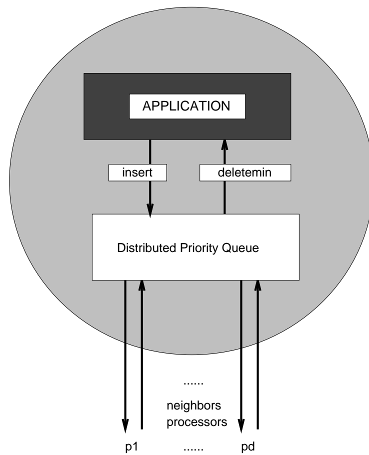

Figure 1: Processus Design in a Network Site

the application and one which maintains the distributed data structure by local access and communication request (see figure 1). The structure is composed of:

- a data organization scheme which specifies how and where to store,
- a set of distributed access protocols that enable a processor to issue modification and query to get appropriate responses.

The set of instructions supported by the DPQ slightly changes from the sequential definition and is defined as follows:

insert(i,h) : insert a new item i with priority Key_i into distributed priority queue h,

deletemin(h) : search the node which detains the item of minimum value (highest priority), delete it, and return it to the querying processor.

2.2 Design choice

The distributed extension of the definition from the sequential (or shared-memory) case does not encompass the intrinsic constraints and functions of a distributed data structure. Even for perfectly defined distributed applications, some aspects of the expected behavior of the data structure are not explicit and, most of the time, are let to the programmer. Most of those choices will not be significant, but some are effective.

Most applications assume that a processor can make a new request only after the processing of the previous one has been completed, or is assumed locally to have been so. Though the definition of the completion of an *insert* query is non trivial, since this operation does not request an acknowledgment for an obvious performance reason. In this case, it is not however possible to manage a peculiar event (e.g., when an identical item already exists).

The *deletemin* operation consists of an *Extract_min* primitive: the data must be deleted from the host site and forwarded to the requesting processor. When the deletemin request reaches a processor in which the item with minimal key is supposed to be, two kinds of actions can be taken by this processor when detecting emptiness in its local queue: (a) assume a potential empty DPQ and initiate an appropriate action (e.g., initiate a termination, start a new problem, etc.); (b) return a failure message to the requesting processor, which will take the appropriate decision (e.g., try again later after a time-out or after receiving a message suggesting that such

an item exists, etc.). However, the action does not depend on the definition of the emptiness. Formally, the DPQ is empty if and only if:

- all the local priority queues of the sites are empty,
- there is no current *insert* request on the communication links.

Another example of fuzzy choices concerns the management of items with same priority. Despite the impressive amount of studies of sequential or parallel priority queues, little attention has been paid to the problem of priority ties. For applications in which such a case occurs, the order of extraction of same priority items has an impact on effectiveness (e.g., [19]). This is all the more relevant in a distributed context where the occurrence of several sites with an item with maximum priority is clearly possible and can lead to erroneous results if the protocol is not consistent. The DPQ must be *stable*: able to enforce both the prediction and the prescription of this order to the data structure–a random pick-up with arbitrary result is not adequate, whereas FIFO or LIFO can be achieved. Some studies avoid the problem of priority ties by forbidding them, in this case the detection problem of inserting an item with same priority must be perfectly defined in order to take the appropriate fault-tolerant action.

If the essence of the design of a data structure is correctness, the quest concerns performance. The communication complexity considered must represent both the worst case (an arbitrary processor requesting an arbitrary item) and the amortized case (average cost over the worst sequence of operations performed consecutively on the data structure). This approach distinguishes the two intrinsic components of the DDS: a query performed correctly and immediately (*response time* and *availability*), and the maintenance of some properties which guarantee future performances (*accessibility* and *concurrency*). To provide fairness to the accessibility, *load-balancing* is required but concerns future performances; it can be delayed, or spread over a longer period, according to the expected sequence of queries. A level of *replication* resulting in redundancy may be suitable for availability (reducing the distance to requested item and avoiding bottlenecks by allowing different location access), and for *reliability* (allowing node failure).

It is worth noting that, by now, none of the studies have considered the local cost of distributed priority queue (except as an implicit constant and identical charge in each processor). Load-balancing is required in two cases: when the amount of data is extremely large (i.e., number of items much larger than number of processors and the amount of main memory needed in a processor may be unacceptable) and when the load changes often in a non-uniform way. The local memory is finite and the memory load, by influencing the local computation, can alter the global computation. The choice between perfect or rough load-balancing is subtle but it is easy to show that for specific sequences, it is not possible for a non-centralized protocol to achieve a perfect load-balancing (this issue is omitted here for brevity).

2.3 Analytic communication measure

We are dealing with *time* complexity T_{query}, i.e., the number of time units from the start of a query to its completion, and the *communication* complexity C_{query}, i.e., the sum of the sizes of all messages sent for the completion of the query.

Other parameters are the number of items m, the memory *Load*, the size in bits of an item δ, and the maximal amount of memory $m\delta$ required in total. The notion is fragile: the global load m can change at anytime. The DDS is *balanced* if the $Load = \frac{m}{n}$ or $Load = \frac{m}{n} + 1$ in every site.

A brief presentation of some basic bounds on the communication for the DPQ is furnished, they are immediate extensions of Peleg's results for DDS [25]. The network is described by an undirected graph $G = (V, E)$, consisting of n $(= |V|)$ vertices and $|E|$ edges.

Lemma 2.1 *For every graph G with diameter $D(G)$, the lower bounds of a request in a DPQ h of items of size δ are*

$$T_{ins} + T_{deletemin} = \Omega(D(G)) \quad \text{and} \quad C_{ins} + C_{deletemin} = \Omega(\delta D(G))$$

Proof Immediate. Assume two processors v and w at maximal distance D. The analysis of *insert(i,h)* from v and *deletemin(h)* from w prove the lemma. □

3 Tree Structure: a Portable Topology

The DPQ introduced here is our point of departure for developing a solution in a more general environment.

In [24], Peleg introduced a distributed *heap tree*: the m items are perfectly load-balanced on a (minimum distance) spanning tree of the n processors network. The two alternatives of the heap invariant, [31], are considered: all items stored at a vertex have a larger, (respectively a lower) priority than all those stored at any of its children. The heap invariant is maintained after each query (this is denoted as *re-heapification* in the following). The root labeled 0 is the center of the network. The tree nodes are labeled in a Depth-First order (DFS) from 0 to $(n-1)$ in order to embed any shortest path tree spanning the network. This ordering gives immediately the position p where an extra-item must be inserted or deleted: $p = m \bmod n$, and this scheme respects a perfect load-balancing. This processor labeling provides also an efficient compact routing as shown in [27]; namely, it does need an explicit but exhaustive, routing tables in each node (only $O(degree)$ information table), and is scalable with the growth of networks. It is also independent of any underlying topology. In both heap invariant cases, the operations are complex and the root regiments all the queries. In the first proposal, upon a receipt of a deletemin query, the root has to locate the processor with the item of minimum value among the leaves (this is done by sending one message in each subtree of the root, to report the answers, extract the item, and at last re-balance) and, therefore, generates numerous messages and high latency. The deletemin of the second proposal, more natural, since the deletemin value (available at the root) is sent immediately to the requesting processor by the root while the re-balancing is done on background. In any case, the insert query completion is immediate, but its background re-heapification is less complex in the first proposal. These proposals are reasonable on a communication complexity approach since they both respect the bounds introduced in Section 2, but it is obvious to predict low concurrency in a real implementation.

The merit of (minimum) spanning trees of processors lies in the facts that they can be designed in any network, and that, by definition, there is no cycle which is suitable to avoid deadlock. The major drawback arises from the root as a bottleneck. However, regarding the intrinsic function of priority queues, the natural centralized control of the site detaining the item with the highest priority forces this behavior: the set of deletemin queries initiated concurrently creates a message congestion in the neighborhood of this particular processor.

In [24], Peleg proposes to run an initial protocol to compute the desired minimum spanning tree and to label the processors accordingly. This is however rarely possible (specifically on heterogeneous networks where distances change arbitrarily) since logical and physical layers of distributed systems do not share the same level of information.

Our implementation uses predefined trees of n processors with heap invariant on the item priorities which is maintained along the sequence of queries. The root retains items with minimum values. The protocol uses different topologies: the d-heap and the binomial tree which both provide a completely implicit routing (no routing information is required as shown in the following).

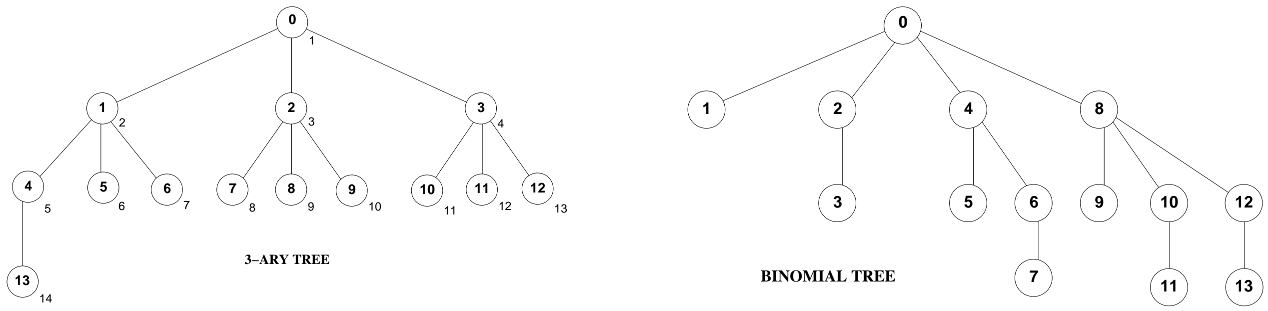

Figure 2: 3-ary Tree and Binomial Tree with 14 processors.

For the d-heap, we use the usual d-ary complete tree labeling: the processor root is labeled 1, the children of a processor i are labeled $d*(i-1)+2, d*(i-1)+3, \ldots, \min(d*i+1, n)$, and the parent of a processor i is $\lceil (i-1)/d \rceil$ (left figure 2 with heap labeling outside of the nodes). The name binomial tree (right figure 2) originated from the fact that, if n is a power of 2, the number of nodes at level i of the tree is the binomial coefficient $\binom{n}{i}$. The processor root is labeled 0 and has children labeled 2^j from $j = 0$ to $\lceil \log n \rceil$, the children of a processor i are labeled $i + 2^j$ (if $< n$) with j from 0 to $k-1$, where k is the power associated to 2 in the prime factor decomposition of i (the number of times that i can be divided by 2). The binomial tree is known to have optimal communication complexity when broadcasting from the root and is widely used in hypercube or complete networks.

3.1 Load-balancing

The distribution of the data respects a perfectly balanced scheme: each processor inserts or deletes an item in a round-robin procedure. In the above proposal of Peleg [24], the position p where an extra-item must be inserted or deleted is set to $(m \bmod n)$. Unfortunately, any sequence of the same query generates a congestion along the path from the root to the position p: the position $p+1$ (or $p-1$) has a high probability of locating a node of the same subtree (only 2 ($\log\ n\ -\ 1$) nodes over n are not in this case). Hence, we introduce a step value s in the round-robin where s is the smallest relatively prime number with n chosen between $\lceil n/4 \rceil$ and n. When an extra-node must be inserted (resp. deleted) the consecutive position p is set to $(p+s) \bmod n$ (resp. $(p-s) \bmod n$). (Of course, care must be taken if $m < n$, then to respect the heap invariant s is set to 1). By definition of the relatively prime, the common divisor between n and s is 1 and each processor will be visited in a round-robin scheme but not in the labeling order. The appropriate choice of the value (close to $n/4$) based on observation of the structure of the d-heap or the binomial trees, will adequately spread the consecutive accesses in different subtrees. For instance, with 14 processors, s is set to 5 and a sequence of n insertions consecutively locates processors:

$$\langle 0, 5, 10, 1, 6, 11, 2, 7, 12, 3, 8, 13, 4, 9 \rangle$$

3.2 Protocol

To complete an insert, a processor sends the new item to the root. Upon receipt of the query, the root computes the position p of the future host processor, selects the appropriate path, and initiates the insertion along the path. In each node, if the priority of the new item is greater than the lowest priority item in the currently visited processor (i.e., if the heap invariant is violated), the new item and this item are exchanged. The remaining item is relayed along the path until the host processor is reached. If necessary, the host processor initiates a re-heapification among

its descendants (i.e., makes sure that its maximum priority item is smaller than the minimum priority item of all the minimum priority items of its children, and exchanges those two if not).

To complete a deletemin, a processor sends the request to the root (this processor cannot initiate another query before receiving the extracted item). Upon receipt of the query, depending on the status of its local queue, the root either (a) immediately sends back its minimum value item, or (b) stores the number of the requesting processor in the waiting list. Case (b) occurs when the DPQ is empty, the item of the next incoming insert query is sent immediately to the first waiting processor. In case (a) the root has to respect the load-balancing by requesting a (minimum priority) item from its processor child on the path to position p. This process is repeated along the path until the processor labeled p is reached and its *Load* decreased. Simultaneously, if the item extracted from the child has a greater priority than the other children, a re-heapification is required with other children.

Clearly, all the modifications (even re-heapification) are done downward from the root, and at a given time involve only two nodes on the same path. Indeed, a processor can check locally, in a priority array, the minimum value of the items in each of its children and can initiate the exchange with the appropriate son. This fact lies on the protocol feature which guarantee, by induction, that an item can not be inserted in a subtree without passing through its root. Assuming that the initial heapification builds these priority arrays, a processor can maintain information of the minimum priority value of each of its children. Obviously, this can be done without any communication overhead and requires only $O(degree)$ memory space in each node.

3.3 Concurrency and consistency

The protocol presented above adapts to message-passing a well-known concurrent technic in global shared-memory tree data structure: the *user view serialization*, [3, 16]. Considering that a basic operation on a data structure is usually made of successive elementary operations, each processor sees and modifies the shared structure as if it could hold the entire tree excluding access. The processors access the data structure downward, inspect a part of the structure that the previous processors make no further changes to, and leave parts of the data structure (modified or not) in a consistent state for incoming processors.

In our message-passing context, the role of the processors and the requests are exchanged. The processors cannot execute a process along the path of the data structure, but relay a request in a top-down scheme. Moreover, the processors deal with one request at a time, providing **atomicity** of modifications in each node. The root by managing the requests forces **serializibility**, providing consistency and concurrency among different subtrees. The proof of correctness is similar to the shared-memory case.

Except for the initial requests directly sent to the root, a bottom-up message can only be an immediate answer to a top-down message initiated by a processor waiting for it (exchanging an item, asking for minimum priority value,...), and, hence cannot create a waiting cycle. All the messages sent are non-blocking: the processor does not wait for the completion of the communication and can immediately manage incoming messages. To guarantee fairness between the request messages and the control, all the receptions are blocking and respect a FIFO scheme. The top-down scheme and the FIFO ordering of messages guarantee atomicity and correctness, without either lock mechanisms or collective communications.

3.4 Local priority queues: the splay tree

Each processor uses a Splay Tree as a local priority queue [28]. Some extra-operations have been defined and implemented to serve the above protocol: InsertSplay, Deletemin, Deletemax,

Findmin and Findmax. Each operation has an amortized cost of $O(\log t)$ for a t-items local priority queue.

Splay trees are self-adjusted data structures and are related to Red-Black trees and AVL trees, [12]. Splay trees maintain balance without explicit balance condition. A **Splay** operation is performed at each access. The Splay operation consists of restructuring heuristically the access path when searching an item i. When top-down splaying at an internal item i, the path from the root x to i is traversed performing a single rotation at each item until $x = i$. If x has a son y but no grandson, we rotate at y (ZIG, see figure 3 (a)). If x has a grandson z, we rotate at z depending of the structure of the path (see figure 3 ZIG-ZIG (b) and ZIG-ZAG (c)). Symmetric variant are omitted. Assembled subtrees, R and L are used, where R (resp. L) contains values greater (resp. lower) to the value i. The overall effect of the splay is to move i to the root while rearranging the rest of the original path to i.

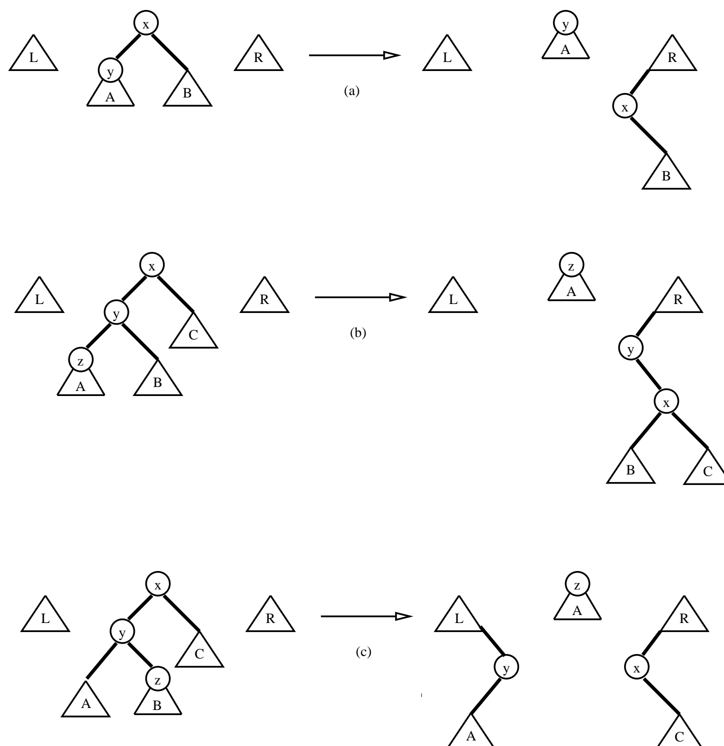

Figure 3: Splay Tree Rotations (a) ZIG, (b) ZIG-ZIG, (c) ZIG-ZAG.

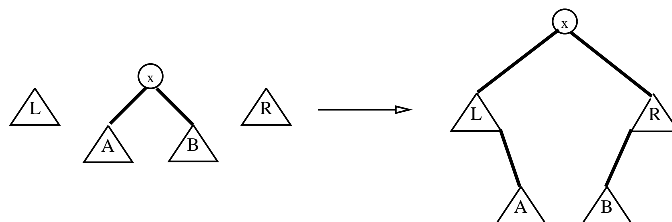

Figure 4: Completion of Top-Down Splay Tree.

In the original version, the tree is finally assembled, as described in figure 4, when the searched item i has been found. However, several releases exist for the Splay operation (all are stable). The Semi-Splay is different from the Splay in ZIG-ZIG rotation. In this case, the tree

is partially assembled at each ZIG-ZIG rotation. A *Top Tree*, assembled progressively, is added to L and R trees, as shown in figure 5. The searched item is not always the root of the tree. A *Simple-Semi-Splay*, more easier to code, can be obtained when executing two ZIG rotations instead of one ZIG-ZAG.

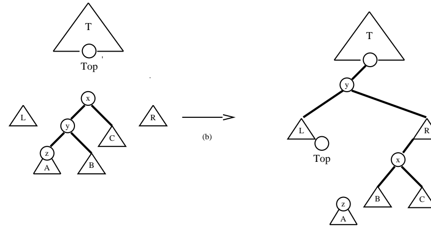

Figure 5: Semi-Splay modified ZIG-ZIG.

The deletemin (as well as the findmin, the deletemax and the findmax) uses ZIG-ZIG rotations and, at most, one ZIG rotation (the item with the least value is the leftmost): there is no ZIG-ZAG rotation. The insert can use the Splay operation in order to find the position for the new item. As an example, two consecutive deletemin and insert operations are depicted in figure 6.

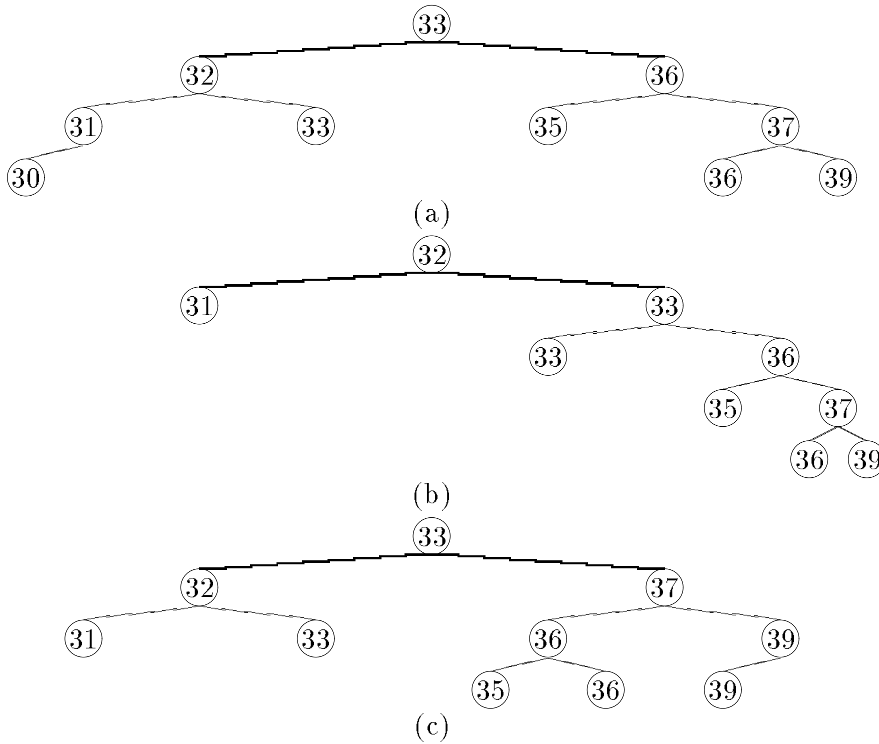

Figure 6: Splay Tree :(a) → (b) (deletemin: 30), (b) → (c) (insert 39)

4 Experimentation and Results Analysis

We have implemented and validated the distributed data structures on a Meiko-T800 Computing surface and on a Cray-T3D (at EPCC, Edinburgh University).

4.1 MPI and the Platforms

The C code developed uses the MPI/CHIMP interface available at the EPCC, University of Edinburgh. The MPI stands for Message Passing Interface [21, 22, 8]. The goal of MPI, simply stated, is to develop a consistent standard for writing message-passing programs (for Fortran 77 and C). One reason for this is due to the wide number of platforms which can support message passing. Programs written with MPI can be executed on distributed or shared memory multiprocessors (Cray, CM5, Meiko,...), networks of workstations (Sun, Dec, Silicon Graphics,...) or a combination of these. The MPI forum was opened to the whole community and was led by a working group with in-depth experience of the use and design of message-passing systems (including PVM, PARMACS, CHIMP, LAM, etc.). More than a standard, MPI intrinsically implies performance by providing some efficient routines (e.g., compared with PVM, it efficiently manages message buffers, with memory-to-memory copy, provides a wide range of communications: synchronous and asynchronous, blocking or not, buffered or not,...). As such, the MPI interface should establish a practical, portable, efficient, and flexible standard for message passing.

Typically a programmer writes one program which is replicated across all the processors of the network. This *Single-Program-Multiple-Data* model (denoted *SPMD*) refers to a restriction of the message-passing model which requires that all processes run the same executable. In practice this is not usually a problem to the programmer, who can incorporate all the different types of process required into one overall executable.

A MPI program communicates with other MPI program by calling MPI routines (MPI comprises a library). The initial loading of the executables onto the parallel machine is out of the scope of the MPI interface (for instance, there is no MPI routine to assign a logical process to a specific processor). There are rules for datatype-matching, point-to-point and collective communications, topology and group constructors.

To confirm portability, the experiments were conducted on two platforms, different by their topologies and their hardware. The **Cray-T3D** massively parallel processing system is a scalable MIMD system with a physically distributed, globally addressable memory. The T3D stands for the physical topology: a Torus 3-dimensions. Each node is composed of two DEC Alpha processors, each with CPU, local memory, and a memory control unit. The data channels are bidirectional, independent in each 3 dimensions, and have a transfer rate of 300MB/s providing low latency. The routing scheme is a combination of "virtual-cut-through" and "wormhole". We also used a **Meiko Computing Surface** consisting of 150 T800 transputers (processors with communication hardware limited to 4 neighbors). This machine, configured as a multi-user resource, continues to provide a MIMD service both to academia and for EPCC internal use. In this instance, we were limited to 51 processors.

4.2 Simulation model

Typically, a priority-queue-based application consists of deletemin-work-insert cycles, in the sense that each extracted item leads to the creation of zero or more new items which must be inserted and will be extracted eventually. To model such a behavior, we tested our code through a *self-scheduling* application and an *event-set* application. In the case of the self-scheduling (commonly used in technics such as divide-and-conquer, Branch and Bound,...), the priority queue is initially empty; the root inserts an item with minimum value; each processor repeatedly

requests a deletemin and inserts two items with the randomly incremented value (unless this value is greater than a maximum value); the application terminates when the (distributed) priority queue is empty. In the case of the event-set, the priority queue is initially filled with a given number of nodes; each processor repeatedly requests a deletemin followed by an insertion of an item with a randomly modified value; the application terminates when each processor has completed a given number of iterations.

Different features of the DPQ are outlined: for the event-set problem, the size of the data structure is relatively constant along the execution (even with an infinite sequence of deletemin and insert, and an initial global load m, the global load is in $[m - n, m]$), whereas for the self-scheduling techniques the global load increases from 0 to a large value ($m \gg n$) and down to 0; creating extreme load-balancing overheads. For this latter, since the oldest waiting deletemin request is served and dequeued with any incoming insertion, the root processor initiates the termination if the waiting queue is full. Regarding the priority domain, the event-set application provides items with arbitrary values, whereas the self-scheduling application tends to provide monotonously increasing priority values of items. In both applications, the termination is broadcast asynchronously downward, and ensures that all the current modifications in between neighbors are completed. Regarding the priority domain, the event-set application provides items with arbitrary values, whereas the self-scheduling application tends to provide monotonously increasing priority values of items.

Detailed timing were made of the completion of the programs in each processor by using the MPI timing routines. (Startup and shutdown costs, these system-dependent components are excluded).

Nondeterminism due to random numbers can be controlled by using a reproducible generator. The value returned by the last deletemin is used as the seed number of the random number generator so that the value of the items generated is independent of the processor host. Each execution has been repeated at least three times in order to avoid random noise and to discard perturbed data.

4.3 Measuring the contention

The severity of the contention problem is determined not by the duration of a request from a processor but by the time that the processor has to wait between two requests. This depends on the application, usually changes throughout the execution and, hence, is difficult to simulate. The speed-up is limited by the ratio of the computation time associated with the application, the grain, to the time required to access items.

$$\text{Speedup} \leq \frac{T_{applic} + T_{request}}{T_{request}}$$

where $T_{request}$ is the time necessary to complete the appropriate accesses to the Distributed Priority Queue to treat an item, and T_{applic} the grain.

Therefore, in our model, each processor attempts a new access to the priority queue as soon the last access is completed, which means in fact that there is no granularity. Both applications flood the network which leads to a considerable overhead. Such an experiment gives a *lower bound of the reachable concurrency* obtained by the distributed priority queue independently of the application. This provides the most unbiased comparison of DPQ. However, it worth noting that, by definition, such a simulation cannot provide a speed-up. In practice, when the DPQ is associated with an application, the concurrency is typically higher (the speed-up is achieved by dividing the overall load of the application between the processors).

4.4 Results and analysis

Experimental studies are used in early design stages to determine values for effective parameters used in models (impact of degree, diameter, message transfer costs,...).

For the self-scheduling, we run the programs with different maximum sizes of priority queue by modifying the random size of the gap between the initial priority and the maximum priority (the greater the gap is, the greater the size of the queue will be). For instance, all the following Meiko curves are obtained with a gap of 120 with maximum increment of 20. Typically, the maximum DPQ sizes varies from 1372 down to 1342 with the increasing number of processors.

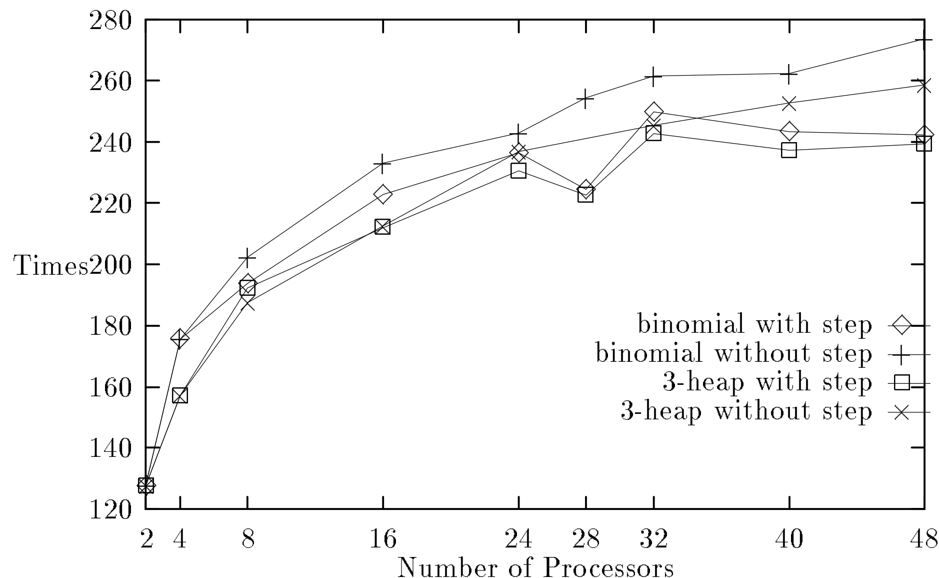

Figure 7: Location Strategy on Meiko-T800 for Self-Scheduling

Figure 7 shows the performances of the DPQ using the load-balancing strategy described earlier in Section 3, the strategy using a step spreads the requests in different subtrees. We show the results on 3-heap and binomial trees (figure 2). The running performance (in seconds) is increased to 15% with 48 processors when using a step not set to 1, and suggests higher benefits with more processors.

To formulate the metric of the sensibility to network degree, we report all timings for Self-Scheduling on the Meiko in figures 7 and 8. The degree of the d-heap have been empirically fixed for the executions with different number of processors.

Gradually increasing the number of processors does not change the number of initial requests to the root but increases the number of messages due to the assignment of the items to a specific processor and the number of messages due to the according maintenance of the heap invariant. A speed-up study does make sense here since there is no granularity to divide in between processors. Here we test the scalability of amount of overhead induced by the heap invariance. In figure 8, the line topology indicates the communication overhead due to the latency of the network. Gradually increasing the distance between the processors decreased the hot-spot impact, though it increases latency. Results curves show that the best running times occur when the degree of the d-heap tree is greater, but close, to the degree of the physical network: 6 from 8 to 32 processors, 7 above. A 6-ary tree with depth 2 includes 8 to 43 processors.

The performance obtained with the binomial tree is satisfactory, although the increasing diameter jeopardizes the effective exploitation of the adaptive degree. The results suggest that the binomial tree is less efficient than an adequate d-heap. The binomial tree structure does

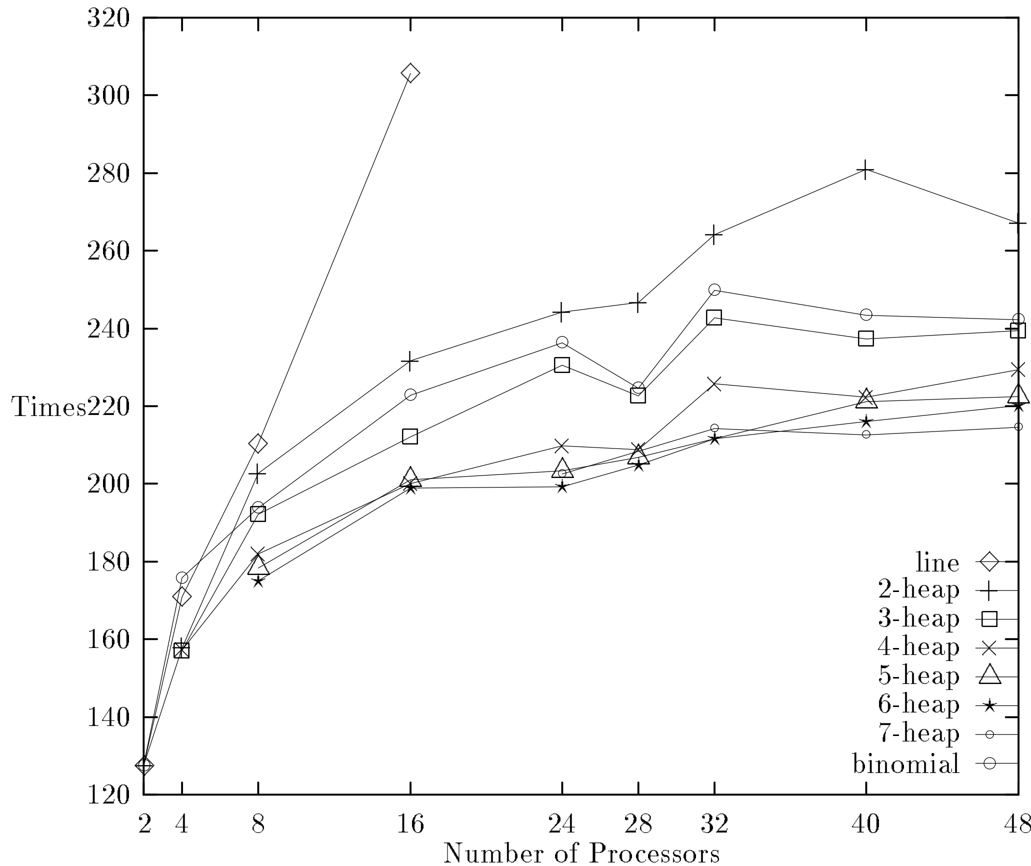

Figure 8: Performances of Topologies for Self-Scheduling on Meiko-T800

not depend on the interconnectivity but rather depends on the number of processors and the maximum degree of a processor is $\lceil \log n \rceil$ but the diameter is within $[\lfloor \log n \rfloor, \lceil \log n \rceil]$. For example, with 40 processors, the binomial tree has maximum degree 6 which is comparable to the suitable degree of the d-heap, but has depth 5 which degrades performances. However, it is worth mentioning that the adaptiveness of the d-heap is not dynamically scalable (in a network with an increasing number of processors); adding some processors extends the 'bottom' of the d-ary tree increasing the number of layers, i.e., the tree's diameter. In the binomial case, adding some processors increases both the diameter and the degree of the tree, scaling the concurrency.

A comparable degradation for the line topology occurs in the Event-Set implementation in figure 9. For these testbeds, the DPQ is initially set with a 2000-items and the priority are arbitrary generated in a $[0, 3000]$ domain. The processors executed 5000 (deletemin-insert) events in total: exactly $5000/n$ each.

To clearly understand the trends of running times while increasing the number of processors, in figure 10 we show the impact of the degree. The efficiency of a specific d-heap implementation falls when a threshold of the number of processors is exceeded: 7 for 28 processors, 7 for 32, 6 for 40, and 8 for 48. The elucidation of the phenomenon of this perturbed sequence required knowledge on the physical partitioning of the Meiko installation and is out of the scope of the present study.

Figure 11 gives some run times on the Cray-T3D. The processors executed 70000 (deletemin-insert) events in total: exactly $70000/n$ each. In these results, because of user time limitation, it was not possible to conduct experiments with a larger number of processors. It is confirmed that the best performances are for degree 5 for 8 to 32 processors.

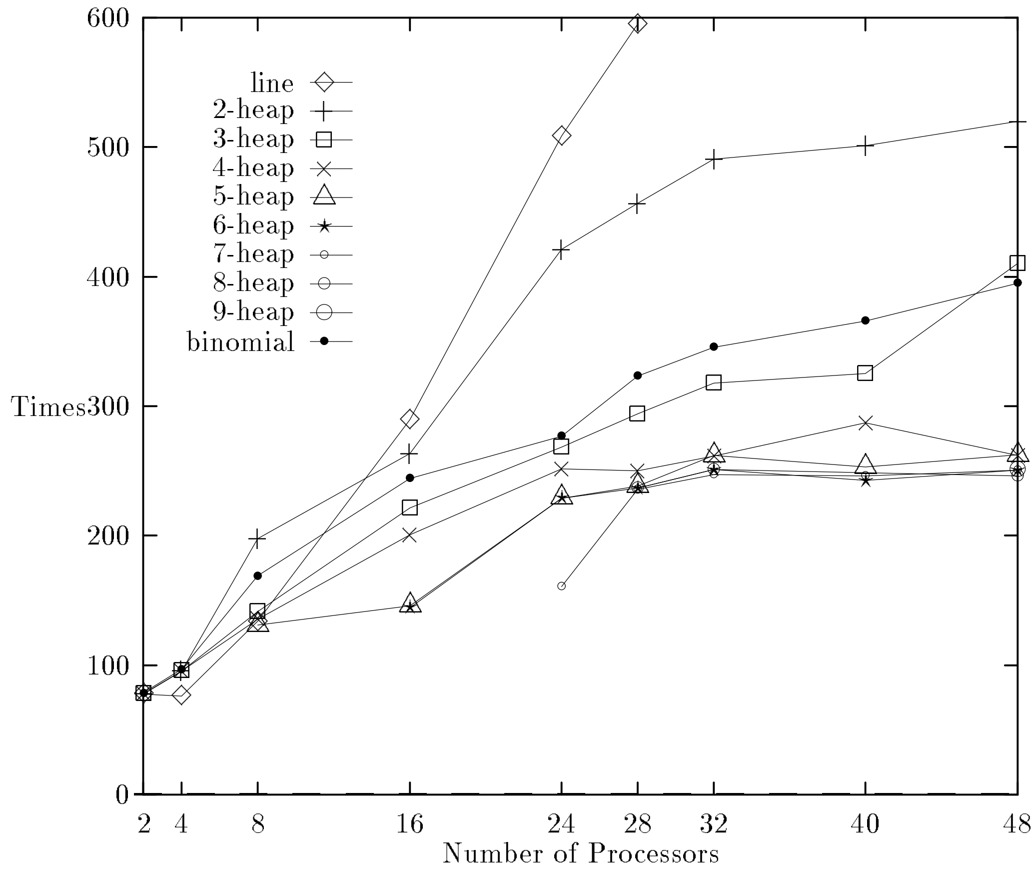

Figure 9: Performances of Topologies for Event-Set on Meiko-T800

These experiments confirm the trade-off between the interconnectivity of networks and the effective use of the number of processors, and suggest the adoption of degree-based DPQ. Our current (and preliminary) results show that the best running times occur when the degree of the logical tree is close to the degree of the physical network: 6 on the T3D and 4 on the Meiko-T800. However, a second result shows that, in order to be scalable, the degree must increase accordingly to the diameter even if it is greater than the physical degree.

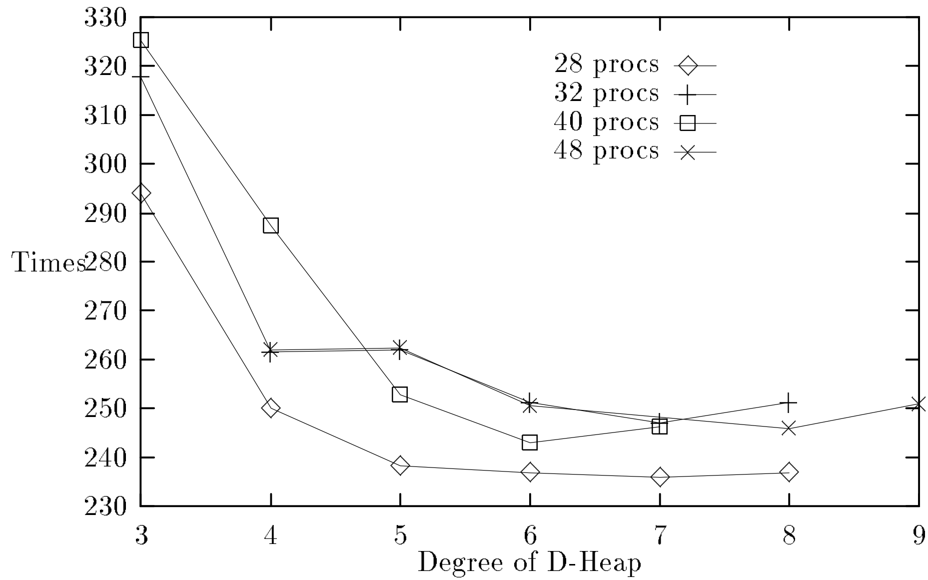

Figure 10: Impact of degree of d-Heap for Event-Set on Meiko-T800

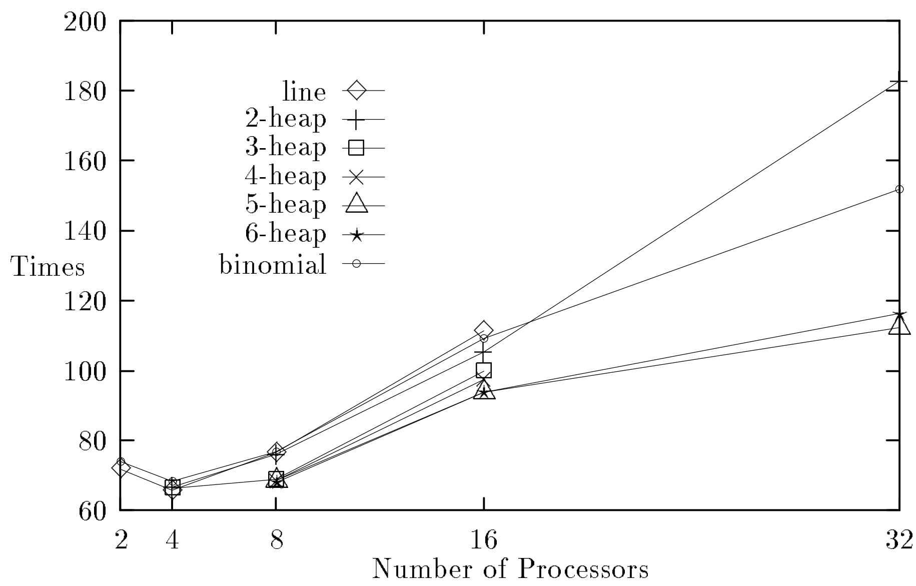

Figure 11: Performances of Topologies for Event-Set on Cray-T3D

5 Concluding Remarks and Perspectives

This discussion illustrates some of the challenges of network computing where processors communicate only through message-passing.

They experiments show that although reasonable concurrency is possible using tree topology, the appropriate degree and diameter is dependent on the interconnectivity and the number of processors. The logical tree degree is not limited only by the interconnection network degree. For message-passing networks, the limits of concurrency are defined by the trade-off between the interconnectivity of the network and the effective use of the number of processors.

These results raise practical and theoretical perspectives. First, an experimentation will include a real implementation (e.g., Branch and Bound solving the Traveling Salesman Problem, Quadratic Assignment Problem [18],...). Second, Fault-tolerance by the introduction of replication and partitioning of the network must be considered. Third, despite an inherent bottleneck due to the nature of the deletemin operation, the control of the request by a unique processor can be reduced, but the solution requires more complicated code. Simultaneously, it is easy to remark that this work, restricted to priority queue operations, can be immediately extended to min-max-median priority queues, as defined in [30].

Acknowledgments. This research supported by James Cook University Merit Research Allocation Grant *"Portable Distributed Priority Queue"* and in part by the TRACS program (Training and Research on Advanced Computing Systems) for Human Capital and Mobility Programme of the European Community, by the Edinburgh Parallel Computing Centre, Edinburgh University, and hosted at the Computing and Electrical Engineering dept., Heriot-Watt University, Edinburgh, United Kingdom.

References

[1] F.B. Bastani, S.S. Iyengar, and I-Ling Yen. Concurrent maintenance of data structures in a distributed system. *Computer Journal*, 21(2):165–174, 1988.

[2] M.R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7(3):298–319, August 1978.

[3] J. Calhoun and R. Ford. Concurrency control mechanisms and the serializability of concurrent tree algorithms. In *Proc. of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo, Ontario, April 1984.

[4] C.A. Crane. Linear lists and priority queues as balanced binary trees. Technical Report 259, Stanford, CA, 1972.

[5] S.K. Das and W.-B. Horng. Managing a parallel heap efficiently. In *Proc. of Conf. on Parallel Architectures and Languages Europe (PARLE'91)*, pages 270–288, 1991.

[6] N. Deo and S. Prasad. Parallel heap. In *International Conference on Parallel Processing*, volume III, pages 169–172, 1990.

[7] C.S. Ellis. Distributed data structures: A case study. *IEEE Transactions on Computers*, C-34(12):1178–1185, December 1985.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, October 1994.

[9] T. Johnson and P. Krishna. Lazy updates for distributed search structure. *SIGMOD record*, 22(2):337–346, 1993. Proceedings of ACM-SIGMOD International Conference on Management of Data, 1993.

[10] D.W. Jones. An empirical comparaison of priority queue and event-set implementation. *Comm. ACM*, 29(4):191–194, April 1986.

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

[12] D.E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[13] B. Kröll and P. Widmayer. Distributing a search tree among a growing number of processors. *SIGMOD record*, 23(2):265–276, 1994. proceedings of ACM-SIGMOD International Conference on Management of Data, Minneapolis, 05-1994.

[14] R. Ladin, B. Liskov, and L. Shira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the 9th Symposium on Principles of Distributed Computing (PODC'90)*, pages 43–57, 1990.

[15] B. Le Cun, B. Mans, and C. Roucairol. Comparison of concurrent priority queues for branch and bound algorithms. Technical Report RR-MASI-92-65, MASI, Université Paris 6, 1992.

[16] P. Lehman and S. Yao. Efficient locking for concurrent operation on b-tree. *ACM Trans. on Database Systems*, 6(4):650–670, December 1981.

[17] B. Mans. Portable distributed priority queues with mpi. In S. Sahni, V. Prasanna, and V. Bhatkar, editors, *Proc. of the IEEE International Conference on High Performance Computing*, pages 16–21, New Delhi, India, 27-30 December 1995. Tata-Mcgraw Hill.

[18] B. Mans, T. Mautor, and C. Roucairol. A parallel depth first search branch and bound for the quadratic assignment problem. *European Journal of Operational Research*, 81(3):617–628, 1995.

[19] B. Mans and C. Roucairol. Performances of parallel branch and bound algorithms with best-first search. *Discrete Applied Mathematics*, to appear, 1996. (accepted October 1994).

[20] G. Matsliach and O. Shmueli. An efficient method for distributing search structures. In *Proceedings of the IEEE First Int. Conf. on Parallel and Distributed Information Systems*, pages 159–166, 1991.

[21] Message Passing Interface Forum. Mpi: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance computing*, 8(3-4), 1994.

[22] Message Passing Interface Forum. Mpi frequently asked questions. http://www.mcs.anl.gov/mpi, 1994.

[23] M.R. Meybodi. Concurrent data structures for hypercube machine. In *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'92)*, pages 703–724, Paris, June 1993.

[24] D. Peleg. Complexity considerations for distributed data structures. Technical Report CS-89-31, Weizmann Institute of Science, Israel, 1989.

[25] D. Peleg. Distributed data structures: a complexity-oriented view. In *Proceedings of the 4th International Workshop on Distributed Algorithms, (WDAG'90)*, pages 71–89. Springer-Verlag, 1990.

[26] V.N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37(12):1657–1665, December 1988.

[27] N. Santoro and R. Khatib. Labeling and implicit routing in networks. *The Computer Journal*, 28:5–8, 1985.

[28] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):235–246, July 1985.

[29] D.D. Sleator and R.E Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, February 1986.

[30] J. Van Leeuven and D. Wood. Interval heaps. *The Computer Journal*, 36(3):209–216, 1993.

[31] J.W.J. Williams. Algorithm 232: heapsort. *CACM*, 7:347–348, 1964.