of sequences of operations where single operations are still $O(\log N)$. In [8, 11, 17] parallel algorithms are presented to speed up individual operations on a $p$-processor EREW-PRAM. In these approaches $\text{ExtractMin}$ and $\text{Insert}$ require $O(\log \log N)$ parallel time using $p = \log N$ processors.

Parallel priority queue operations with $n > 1$ have been discussed in [1, 3, 4, 7, 10, 12, 14]. Randomised algorithms are presented in [4, 12] whereas tree-based deterministic algorithms are discussed in [3, 10]. This paper is on deterministic algorithms using a tournament tree. Applications of these algorithms are in branch-and-bound problems where it is necessary to select the next $n$ subproblems (i.e., $n$ higher priority valued subproblems) to be solved in turn to generate new subproblems and so on (e.g., parallel discrete event simulation).

The paper is organized as follows. First we review the PRAM and BSP models for parallel computation. The next two sections presents our parallel algorithms for priority queues based in complete binary search trees in both models.

2 Two models for parallel computation

A well-known model of theoretical parallel computation is the parallel random access machine (PRAM) [5]. In a PRAM $p$ processors have concurrent unit time access to a shared memory. The number of processors is unbounded. The processors execute a program synchronised after each clock cycle of the machine. During each clock cycle each processor is capable of performing a local computation, reading two words of data from the global address space and writing one word of data to the global address space. However, no truly scalable PRAM computer has been so far built. Shared memory with unit access time is unrealistic to physically implement for large number of processors with current technology. Nevertheless, PRAMs are the target of an important amount of parallel algorithms [5]. In this paper we use a concurrent read exclusive write (CREW) PRAM where two or more processors may read the same memory cell simultaneously but they may not write the same cell.

In the bulk-synchronous parallel (BSP) model [16, 9] any parallel computer is seen as composed of a set of $p$ processor-memory pairs which communicate each other through messages. A BSP computation is organised as a sequence of supersteps wherein each superstep is delimited by a barrier synchronisation of all of the processors. During a superstep each processor may only perform computations on data locally held and send messages to other processors. However, the messages sent during a superstep are only available to the target processors at the beginning of the next superstep. Within supersteps the work performed by each processor is sequential and asynchronous.

Performance prediction is a central aim in the BSP model. The cost of each superstep is given by the expression $w + h g + \ell$, with (i) $w = \max(w_i)$ where $w_i$ is the cost of computations performed by processor $i$ during the superstep, (ii) $h = \max(h_i^{j\rightarrow}, h_i^{j\rightarrow})$ where $h_i^{j\rightarrow}$ is the total amount of messages received (sent) by processor $i$ (j), and $g$ is the cost of transmitting one word in a situation of continuous traffic in the communication network, and (iii) $\ell$ is the cost of performing a barrier synchronisation of all processors. Note that particular hardware designs are hidden and cost by the parameters $g$ and $\ell$; these values can be empirically determined by running benchmark
programs on each type of machine. The total cost of a BSP computation is then the sum of the
costs of individual supersteps.

An exclusive read exclusive write (EREW) PRAM can be seen as a BSP computer with \( g = \ell = 1 \). Note that under situations of sufficient parallel slackness it is possible to simulate a PRAM
computer upon a BSP computer into 1-optimal efficiency with high probability, see [16] and papers
at [15] (the term slackness refers to the ratio of the problem size \( n \) to the number of processors \( p \)).

3 Tournament tree and basic sequential operations

Every item stored in the PQ consists of a priority value and an identifier. We associate every leaf
of the CBT with one item, and use the internal nodes to maintain a continuous binary tournament
among the items. A match, at internal node \( d \), consists of determining the item with higher priority
(less numerical value) between the two children of \( d \) and writing the identifier of the winner in \( d \).
The tournament is made up of a set of matches played in every internal node located in each
path from the leaves to the root. Each time that the priority associated with a leaf at position \( e \) is changed, the tournament is updated performing matches along the unique path between the
position \( e \) and the root of the tree. We call this operation \( \text{update-cbt}(e) \).

A PQ with \( N \) items is implemented using: (i) an array \( \text{CBT}[1 \ldots 2N - 1] \) of integers to
maintain results of matches among items, (ii) an array \( \text{Prio}[1\ldots N] \) of priority values, and (iii) an
array \( \text{Leaf}[1\ldots N] \) of integers to map between items and leaves. A node at position \( s \) in the array
\( \text{CBT} \) has its children at positions \( 2s \) and \( 2s + 1 \). The parent of a node \( s \) is at position \( \lfloor \frac{s}{2} \rfloor \). All
internal nodes are stored between positions \( 1 \) and \( N - 1 \) of the CBT. The highest priority value in
the PQ is given by \( \text{Prio}[\text{CBT}[1]] \), its identifier is \( i = \text{CBT}[1] \), and its associated leaf is at position
\( e = \text{Leaf}[i] \) of the CBT. To enable a dynamic reusing of item identifiers in the PQ, the array \( \text{Leaf} \) is
also used to maintain a single linked list of available item identifiers (initially this list is empty).

Deletions in the CBT are performed by removing the child with lower priority between the
children of the parent of the rightmost leaf (i.e., the child of internal node at position \( N - 1 \) in
CBT array which has lower priority), and exchanging it with the target leaf \( e \) to be deleted (i.e.
\( e = \text{Leaf}[\text{CBT}[1]] \)). Then \( \text{update-cbt}(e) \) is executed. Insertions are performed by appending a new
rightmost leaf and updating the CBT. This is done by expanding into two leaves the first leaf of
the tree (i.e. the leaf at position \( N \) in CBT array) and executing \( \text{update-cbt}(2N - 1) \).

In the context of the parallel operations described later, it is relevant to know the number of
item identifiers found in any path from the leaves to the root. In the worst case, this number is
\( h \approx \log_2 N \) but on the average case is \( h^* \approx h/2 \) as we show next for the case \( N = 2^k - 1 \).

Any random assignment of results in the matches performed in the CBT can be equivalently
seen as enumerating the nodes from left to right level by level towards the root as follows. We
start from the leaves (which are all at the same level \( k \) since \( N = 2^k - 1 \)) enumerating from 1 to \( N \).
Then we continue in the next level \( k - 1 \) enumerating 1, 3, 5, 7, ..., \( N - 1 \), the next level \( k - 2 \)
enumerating 1, 5, 9, ..., \( N - 3 \), and so on till we enumerate the root with 1. This enumeration