A Parallel Priority Queue with Constant Time Operations

Gerth S. Brodal   Jesper L. Träff
Christos D. Zaroliagis

MPI–I–97–1-011   May 1997

A Parallel Priority Queue with Constant Time Operations*

Gerth Stølting Brodal† Jesper Larsson Träff‡ Christos D. Zaroliagis

Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany

Email: {brodal,traff,zaro}@mpi-sb.mpg.de

May 16, 1997

Abstract

We present a parallel priority queue that supports the following operations in constant time: *parallel insertion* of a sequence of elements ordered according to key, *parallel decrease key* for a sequence of elements ordered according to key, *deletion of the minimum key element*, as well as *deletion of an arbitrary element*. Our data structure is the first to support multi insertion and multi decrease key in constant time. The priority queue can be implemented on the EREW PRAM, and can perform any sequence of n operations in $O(n)$ time and $O(m \log n)$ work, m being the total number of keys inserted and/or updated. A main application is a parallel implementation of Dijkstra's algorithm for the single-source shortest path problem, which runs in $O(n)$ time and $O(m \log n)$ work on a CREW PRAM on graphs with n vertices and m edges. This is a logarithmic factor improvement in the running time compared with previous approaches.

Keywords: Parallel data structures, parallel algorithms, graph algorithms, priority queues.

1 Introduction

A priority queue is a sequential data structure which can maintain a set of elements with keys drawn from a totally ordered universe subject to the operations of insertion, deletion,

*This work was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT).

†Supported by the Danish Natural Science Research Council (Grant No. 9400044). This work was done while the author was with BRICS (Basic Research in Computer Science, a Centre of the Danish National Research Foundation), Dept. of Computer Science, University of Aarhus, Denmark.

‡Supported by the DFG project SFB 124-D6 (VLSI Entwurfsmethoden und Parallelität).

decrease key, and find minimum key element. There has been a considerable amount of work on *parallel priority queues*, see for instance [2, 5, 8, 9, 25, 26, 27, 28].

There are two different directions for incorporating parallelism into priority queues. The first is to speed up the individual queue operations that handle a *single* element, using a small number of processors [2, 5, 25, 27, 28]. For instance, the parallel priority queue of Brodal [2] supports find minimum in constant time with one processor, and insertion, deletion, and decrease key operations (as well as other operations) in constant time with $O(\log n_0)$ processors, n_0 being the maximum number of elements allowed in the priority queue. The other direction is to support the simultaneous insertion of k elements and the simultaneous deletion of the k *smallest* elements, k being a constant. Pinotti and Pucci introduced in [26] the notion of k-*bandwidth* parallel priority queue implementations, by giving implementations of k-bandwidth-heaps and k-bandwidth-leftist-heaps for the CREW PRAM. Using the k-bandwidth idea Chen and Hu [5] give an EREW PRAM parallel priority queue supporting multi insert and multi delete (of the k smallest elements) in $O(\log\log \frac{n}{k} + \log k)$ time. Ranade *et al.* [28] show how to apply the k-bandwidth technique to achieve a parallel priority queue implementation for a d-dimensional array of processors, and Pinotti *et al.* [25] and Das *et al.* [8] give implementations for hypercubes. None of the above data structures supports the simultaneous deletion of k arbitrary elements.

In this paper we present a parallel priority queue which supports simultaneous insertion and simultaneous decrease key of an *arbitrary* sequence of elements ordered according to key, in addition to find minimum and single element delete operations. These operations can all be performed in constant time. Our main result is that any sequence of n queue operations involving m elements in total can be performed in $O(n)$ time using $O(m \log n)$ operations on the EREW PRAM. The basic idea in the implementation is to perform a pipelined merging of keys. With the aid of our parallel priority queue we can give a parallel implementation on the CREW PRAM of Dijkstra's single-source shortest path algorithm running in $O(n)$ time and $O(m \log n)$ work on digraphs with n nodes and m edges. This improves the running time of previous implementations [11, 24] by a logarithmic factor, while sacrificing only a logarithmic factor in the work. This is the fastest, work-efficient parallel algorithm for the single-source shortest path problem.

The rest of the paper is organized as follows. In Section 2 we define the operations supported by our parallel priority queue. The main application to Dijkstra's single-source shortest path algorithm is presented in Section 3. In Section 4 we give a simple implementation of the priority queue which illustrates the basic idea of the pipelined merge, but requires $O(n^2 + m \log n)$ work for a sequence of n queue operations. In Section 5 we show how to reduce the work to $O(m \log n)$ by dynamically restructuring the pipeline in a tree like fashion. Further applications are discussed in Section 6. A preliminary version of the paper appeared as [4]. In that version a parallel priority data structure was proposed supporting a somewhat different set of operations, more directly tailored to the parallel implementation of Dijkstra's algorithm.

2 A parallel priority queue

In this section we specify the operations supported by our parallel priority queue. We will be working on the PRAM [18, 19], and for the description of the queue operations and the simple implementation assume that successively numbered processors $P_1, \ldots, P_i, \ldots$, are available as we need them. In Section 5 we will then show how to work with a reduced number of processors.

Consider a set of up to n_0 *elements* $e_1, \ldots, e_{n_0}$, each with a *key* drawn from a totally ordered set. We emphasize that an element e_i has key d_i by writing $e_i(d_i)$. Keys do *not* uniquely identify elements, that is $e(d')$ and $e(d'')$ are different *occurrences* of the *same* element e. The priority queue maintains a set Q of elements subject to the operations described below. At any given instant a set of successively numbered processors $P_1, \ldots, P_i$ will be associated with Q. We use $|Q|$ to denote the number of processors currently associated with Q. The priority queue operations are executed by the available processors in parallel, with the actual work carried out by the $|Q|$ processors associated with the queue. The operations may assign new processors to Q and/or change the way processors are associated with Q. The result (if any) returned by a queue operation is stored at a designated location in the shared memory.

- $\textsc{Init}(Q)$: initializes Q to the empty set.
- $\textsc{Update}(Q, L)$: updates Q with a list $L = e_1(d_1), \ldots, e_k(d_k)$ of (different) elements in non-decreasing key order, *i.e.*, $d_1 \leq \cdots \leq d_k$. If element e_i was not in the queue before the update, e_i is inserted into Q with key d_i. If e_i was already in Q with key d'_i, the key of e_i is changed to d_i if $d_i < d'_i$, otherwise e_i remains in Q with its old key d'_i.
- $\textsc{DeleteMin}(Q)$: deletes and returns the minimum key element from Q in location $\textsc{MinElt}$.
- $\textsc{Delete}(Q, e)$: deletes element e from Q.
- $\textsc{Empty}(Q)$: returns **true** if Q is empty in location $\textsc{Status}$.

The $\textsc{Update}(Q, L)$ operation provides for (combined) multi insert and multi decrease key for a sequence of elements ordered according to key. For the implementation, it is important that the sequence be given as a list, enabling one processor to retrieve, starting from the first element, the next element in constant time. We will represent such a list of elements as an *object* with operations $L.\mathsf{first}$, $L.\mathsf{remfirst}$ for accessing and removing the head (first element) of the list, operations $L.\mathsf{curr}$ and $L.\mathsf{advance}$ for returning a current element and advancing to the next element, and $L.\mathsf{remove}(e)$ for removing the element (pointed to by) e. When the end of the list is reached by operation $L.\mathsf{advance}$, $L.\mathsf{curr}$ returns a special element $\perp$. A list object can easily be built to support these operations in constant time (with one processor).

In Sections 4 and 5 we present two different implementations of the priority queue. In particular, we establish the following two main results:

Theorem 1 *The operations* $\textsc{Init}(Q)$ *and* $\textsc{Empty}(Q)$ *take constant time with one processor. The* $\textsc{DeleteMin}(Q)$ *and* $\textsc{Delete}(Q, e)$ *operations can be done in constant time by* $|Q|$

processors. The operation UPDATE(Q, L) *can be done in constant time by* $1+|Q|$ *processors, and assigns one new processor to* Q*. The priority queue can be implemented on the EREW PRAM. Space consumption per processor is* $O(n_0)$*, where* n_0 *is the maximum number of elements allowed in the queue.*

Theorem 2 *The operations* INIT(Q) *and* EMPTY(Q) *take constant time with one processor. After initialization, any sequence of* n *queue operations involving* m *elements in total can be performed in* $O(n)$ *time with* $O(m \log n)$ *work. The priority queue can be implemented on the EREW PRAM. Space consumption per processor is* $O(n_0)$*, where* n_0 *is the maximum number of elements allowed in the queue.*

Before giving the proofs of Theorems 1 and 2 in Sections 4 and 5 respectively we present our main application of the parallel priority queue.

3 The main application

The *single-source shortest path problem* is a notorious example of a problem which despite much effort has resisted a very fast (*i.e.*, NC), work-efficient parallel solution. Let $G = (V, E)$ be an n-vertex, m-edge directed graph with real-valued, non-negative edge weights $c(v, w)$, and let $s \in V$ be a distinguished *source vertex*. The single-source shortest path problem is to compute for all vertices $v \in V$ the length of a shortest path from s to v, where the length of a path is the sum of the weights of the edges on the path.

The best sequential algorithm for the single-source shortest path problem on directed graphs with non-negative real valued edge weights is Dijkstra's algorithm [10]. The algorithm maintains for each vertex $v \in V$ a tentative distance $d(v)$ from the source, and a set of vertices S for which a shortest path has been found. The algorithm iterates over the set of vertices of G, in each iteration selecting a vertex of minimum tentative distance which can be added to S. The algorithm can be implemented to run in $O(m + n \log n)$ operations by using efficient priority queues like Fibonacci heaps [12] for maintaining tentative distances, or other priority queue implementations supporting deletion of the minimum key element in amortized or worst case logarithmic time, and decrease key in amortized or worst case constant time [3, 11, 17].

The single-source shortest path problem is in NC (by virtue of the all-pairs shortest path problem being in NC), and thus a fast parallel algorithm exists, but for general digraphs no *work-efficient* algorithm in NC is known. The best NC algorithm runs in $O(\log^2 n)$ time and performs $O(n^3(\log \log n / \log n)^{1/3})$ work on an EREW PRAM [16]. Moreover, work-efficient algorithms which are (at least) sublinearly fast are also not known for general digraphs.

Dijkstra's algorithm is highly sequential, and can probably not be used as a basis for a fast (NC) parallel algorithm. However, it is easy to give a parallel implementation of the algorithm that runs in $O(n \log n)$ time [24]. The idea is to perform the distance updates within each iteration in parallel by associating a local priority queue with each processor. The vertex of minimum distance for the next iteration is determined (in parallel) as the minimum of the minima in the local priority queues. For this parallelization it is important that the priority queue operations have worst case running time, and therefore the original Fibonacci heap cannot be used to implement the local queues. This was first observed in [11] where a new data structure, called relaxed heaps, was developed to overcome this problem. Using relaxed heaps, an $O(n \log n)$ time and $O(m+n \log n)$ work(-optimal) parallel implementation of Dijkstra's algorithm is obtained. This seems to have been the previously fastest work-efficient parallel algorithm for the single-source shortest path problem. The parallel time spent in each iteration of the above implementation of Dijkstra's algorithm is determined by the (processor local) priority queue operations of finding a vertex of minimum distance and deleting an arbitrary vertex, plus the time to find and broadcast a global minimum among the local minima. Either or both of the priority queue operations take $O(\log n)$ time, as does the parallel minimum computation; for the latter $\Omega(\log n)$ time is required, even on a CREW PRAM [7]. Hence, the approach with processor local priority queues does not seem to make it possible to improve the running time beyond $O(n \log n)$ without resorting to a more powerful PRAM model. This was considered in [24] where two faster (but not work-efficient) implementations of Dijkstra's algorithm were given on a CRCW PRAM: the first algorithm runs in $O(n \log\log n)$ time, and performs $O(n^2)$ work; the second runs in $O(n)$ time and performs $O(n^{2+\epsilon})$ work for $0 < \epsilon < 1$.

An alternative approach would be to use a parallel global priority queue supporting some form of multi-decrease key operation. As mentioned in the introduction none of the parallel priority queues proposed so far support such an operation; they only support a multi-delete operation which assumes that the k elements to be deleted are the k elements with smallest keys in the priority queue. This does not suffice for a faster implementation of Dijkstra's algorithm.

Using our new parallel priority queue, we can give a linear time parallel implementation of Dijkstra's algorithm. Finding the vertex of minimum distance and decreasing the distances of its adjacent vertices can obviously be done by the priority queue, but preventing that a vertex, once selected and added to the set S of correct vertices, is ever selected again requires a little extra work. The problem is that when vertex v is selected by the find minimum operation, some of its adjacent vertices may have been selected at a previous iteration. If care is not taken, our parallel update operation would reinsert such vertices into the priority queue, which would then lead to more than n iterations. Hence, we must make sure that we can remove such vertices from the adjacency list of v in constant time upon selection of v. We first sort the adjacency list of each vertex $v \in V$ according to the weight of its adjacent edges. Using a sublinear time work-optimal mergesort algorithm [6, 15] this is done with $O(m \log n)$ work on the EREW PRAM. This suffices to ensure that priority queue updates are performed on lists of vertices of non-decreasing tentative distance. To make it possible to remove in constant time any vertex w from the adjacency list of v we make the sorted adjacency lists doubly linked. For each $v \in V$ we also construct an array consisting of the vertices w to which v is adjacent, $(w, v) \in E$, together with a pointer to the position of v in the sorted adjacency list of w. This preprocessing can easily be carried out in $O(\log n)$ time using linear work on the EREW PRAM.

Let L_v be the sorted, doubly linked adjacency list of vertex $v \in V$, and I_v the array of vertices to which v is adjacent. As required in the specification of the priority queue, we represent each L_v as an object with operations L_v.first, L_v.remfirst, L_v.curr, L_v.advance and L_v.remove(e). In the iteration where v is selected the object for L_v will be initialized with a

Algorithm Parallel-Dijkstra
/* Initialization */
Sort the adjacency lists of G after edge weight, and make doubly linked lists L_v;
For each v build array I_v of vertices to which v is adjacent;
$\textsc{Init}(Q)$;
$d(s) \leftarrow 0$; $S \leftarrow \{s\}$;
$\textsc{Update}(Q, L_s(0))$;
/* Main loop */
while $\neg\textsc{Empty}(Q)$ **do**
    $v(d) \leftarrow \textsc{DeleteMin}(Q)$;
    $d(v) \leftarrow d$; $S \leftarrow S \cup \{v\}$;
    $\textsc{Update}(Q, L_v(d))$;
    forall $w \in I_v$ **pardo**
        if $w \notin S$ **then** remove v from L_w **fi**;
    odpar
od

Figure 1: An $O(n)$ time parallel implementation of Dijkstra's algorithm.

constant value d representing the distance of v from the source. We denote this initialization of the object by $L_v(d)$. The operations L_v.first and L_v.curr return a vertex w on the sorted adjacency list of v (first, respectively current) with key $c(v, w)$ offset by the value d, *i.e.* $d + c(v, w)$. The initialization step is completed by initializing the priority queue Q, and inserting the object $L_s(0)$ representing the vertices adjacent to the source vertex s with offset 0 into Q.

We now iterate as in the sequential algorithm until the priority queue becomes empty, in each iteration deleting a vertex v with minimum key (tentative distance) from Q. As in the sequential algorithm the distance d of v will be equal to the length of a shortest path from s to v, so v is added to S and should never be considered again. The adjacency list object $L_v(d)$ representing the vertices adjacent to v offset with v's distance from s is inserted into the priority queue, and will in turn produce the tentative distances $d + c(v, w_i)$ of v's adjacent vertices w_i. Since the adjacency lists were initially sorted, the tentative distances produced by the L_v object will appear in non-decreasing order as required in the specification of the priority queue. To guarantee that the selected vertex v is never selected again, v must be removed from the adjacency lists of all vertices $w \notin S$. This can be done in parallel in constant time by using the array I_v to remove v from the adjacency lists L_w of vertices to which v is adjacent for all $w \notin S$. Note that this step requires concurrent reading, since the $|I_v|$ processors have to know the starting address of the I_v array of the selected vertex v. However, the concurrent reading required is of the restricted sort of broadcasting the same constant-size information to a set of processors. A less informal description of the above implementation of Dijkstra's algorithm is given in Figure 1.

Theorem 3 *The parallel Dijkstra algorithm runs: (i) in $O(n)$ time and $O(m \log n)$ work on the CREW PRAM; (ii) in $O(n \log(m/n))$ time and $O(m \log n)$ work on the EREW PRAM.*

Proof. (i) The initialization takes sublinear time and $O(m \log n)$ work on an EREW PRAM, depending on the choice of parallel sorting algorithm. Since one vertex is put into S in each iteration, at most $n-1$ iterations of the **while** loop are required. Each iteration (excluding the priority queue operations) can obviously be done in constant time with a total amount of work bounded by $O(\sum_{v \in V} |L_v| + \sum_{v \in V} |I_v|) = O(m)$. We have a total of n priority queue operations involving a total number of elements equal to $\sum_{v \in V} |L_v| = m$. Now, the bounds of part (i) follow from Theorem 2.

(ii) Concurrent reading was needed only for removing the selected vertex v from the adjacency lists of vertices $w \notin S$ using the I_v array. This step can be done on an EREW PRAM if we broadcast the information that v was selected to $|I_v|$ processors. In each iteration this can be done in $O(\log |I_v|)$ time and $O(|I_v|)$ work. Summing over all iterations gives $O(\sum_{v \in V} \log |I_v|) = O(\log(\Pi_{v \in V} |I_v|) = O(n \log(m/n))$ time and $O(m)$ work. □

4 Linear pipeline implementation

In this section we present a simple implementation of the priority queue as a linear pipeline of processors, thereby giving a proof of Theorem 1.

At any given instant the processors associated with Q are organized in a linear pipeline. When an UPDATE(Q, L) operation is performed a new processor becomes *associated* with Q and is put at the front of the pipeline. Elements of the list L may already occur in Q, possibly with different keys; it is the task of the implementation to ensure that only the occurrences with the smallest keys are output by the DELETEMIN(Q) operation. An array is used to associate processors with Q. Let P_i denote the ith processor to become associated with Q. The task of P_i will be to perform a stepwise merging of the elements of the list $L = e_1(d_1), \ldots, e_k(d_k)$ with the output from the previous processor P_{i-1} in the pipeline (where $i > 1$). Since L becomes associated with P_i at the UPDATE(Q, L) call, we shall refer to it as the element list L_i of P_i when we describe actions at P_i. Processor P_i produces output to an *output queue* Q_i; Q_i is either read by the next processor, or, if P_i is the last processor in the pipeline, Q_i contains the output to be returned by the next DELETEMIN(Q) operation. The pipeline after 4 UPDATE(Q, L) operations is shown in Figure 2. Each Q_i is a standard FIFO queue with operations Q_i.first, which returns the first element of Q_i, Q_i.remfirst, which deletes the first element of Q_i, and Q_i.append(e), which appends the element e to the rear of Q_i. Furthermore, each Q_i must support deletion of an element (pointed to by) e in constant time, Q_i.remove(e). Implementation of each Q_i as a doubly linked list suffices.

The INIT(Q) operation marks all processors as not associated with Q, and can therefore be done in constant time by initializing the association array. The operations DELETEMIN(Q), DELETE(Q, e) and UPDATE(Q, L) are all implemented by a procedure MERGESTEP(Q), which, for each processor P_i associated with Q, performs one step of a merge of the element list L_i of P_i and the elements in the output queue Q_{i-1} of the previous processor.

Let $Q(i)$ denote the contents of the priority queue after the ith UPDATE(Q, L) operation. The purpose of procedure MERGESTEP(Q) is to output, for each processor P_i associated

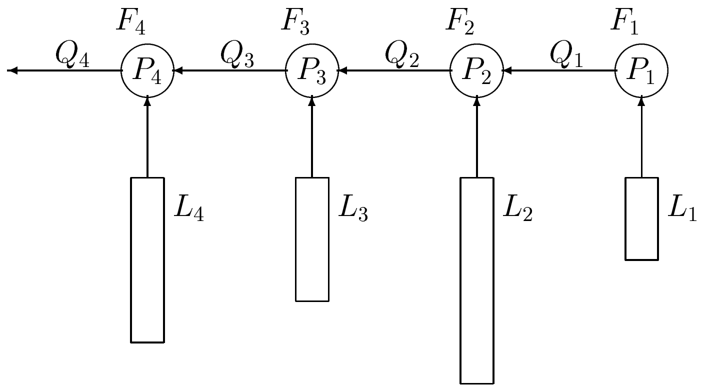

Figure 2: The linear processor pipeline with associated data structures.

with Q, the next element of $Q(i)$, if non-empty, in non-decreasing order to the output queue Q_i. This is achieved as follows. Recall that L_i is a list of elements, some of which may already have been in the priority queue before the ith update (possibly with different keys). The elements output to each output queue will be in non-decreasing order, so the merge step of processor P_i simply consists in choosing the first element of either Q_{i-1} or L_i, whichever is smaller (with ties broken arbitrarily), deleting this element and outputting it to Q_i. The merge step must also ensure that an element is output from Q at most once. There can be occurrences of an element in different lists L_i corresponding to a number of updates on this element; the occurrence with the smallest key must be output. In order to guarantee that an element is output from $Q(i)$ (by P_i) at most once, an element is marked as *forbidden* by processor P_i once it is output. Each processor maintains a set F_i of forbidden elements, represented as a Boolean array indexed by elements: $F_i[e] = \textbf{true}$ iff e has been output and made forbidden by P_i. This ensures that each Q_i always contains different elements. In order that the merge step can be performed in constant time, it must furthermore hold that neither Q_{i-1} nor L_i contain elements that are forbidden for P_i. We maintain the invariants that

$$F_i \cap Q_{i-1} = \emptyset \text{ and } F_i \cap L_i = \emptyset.$$

The merge step for processor P_i now proceeds as follows: the smaller element is chosen from either Q_{i-1} or L_i (with ties broken arbitrarily), presuming neither is empty. If either Q_{i-1} or L_i is empty the element is taken unconditionally from the other sequence, and when both are empty, P_i has no more work to do. If the chosen element is not forbidden for the next processor P_{i+1}, it is output to Q_i, and made forbidden for P_i. If it also occurs in either Q_{i-1} or L_i it must be deleted so that the above invariants are maintained. To this end, each processor maintains two arrays of pointers $\overline{Q}_i$ and $\overline{L}_i$ into Q_i and L_i, respectively, indexed by the elements. When an element e is inserted into Q_i, a pointer $\overline{Q}_i[e]$ to e in Q_i is created; when e is removed from Q_i (by processor P_{i+1}) $\overline{Q}_i[e]$ is reset to $\perp$. The pointers $\overline{L}_i[e]$ into L_i should be set, conceptually, when the update $\textsc{Update}(Q, L)$ is performed. However, this would require concurrent reading, so instead we initialize the $\overline{L}_i$ pointer array in a pipelined fashion. Since at most one element from L_i is "consumed" at each merge step, it suffices to

Procedure MergeStep(Q)
forall P_i, $i \in |Q|$ **pardo** /* for all processors associated with Q */
    if L_i.curr $\neq \perp$ **then** /* lazy $\overline{L}_i$ pointer update */
        if $F_i[L_i.\mathsf{curr}] =$ **true then** L_i.remove(L_i.curr); /* current element is forbidden */
        else $\overline{L}_i[L_i.\mathsf{curr}] \leftarrow L_i$.curr; **fi**;
        L_i.advance; /* advance to next element */
    fi;
    $e'(d') \leftarrow Q_{i-1}$.first;
    $e''(d'') \leftarrow L_i$.first;
    if $d'' < d'$ **then**
        $e'(d') \leftarrow e''(d'')$;
        L_i.remfirst;
        /* remove e' from Q_{i-1} using $\overline{Q}_{i-1}[e']$ */
        if $\overline{Q}_{i-1}[e'] \neq \perp$ **then** Q_{i-1}.remove($\overline{Q}_{i-1}[e']$);
    else
        Q_i.remfirst;
        /* remove e' from L_i using $\overline{L}_i[e']$ */
        if $\overline{L}_i[e'] \neq \perp$ **then** L_i.remove($\overline{L}_i[e']$);
    fi;
    $F_i[e'] \leftarrow$ **true**;
    if $\neg F_{i+1}[e']$ **then**
        Q_i.append($e'(d')$);
        Update $\overline{Q}_i[e']$ to the position of e' in Q_i;
    fi;
odpar;
End of Procedure

Figure 3: The MergeStep(Q) procedure.

let each merge step initialize the pointer for the next element of L_i. When an element e is chosen from Q_{i-1} and has to be deleted from L_i, either $\overline{L}_i[e]$ already points to e's position in L_i, or it has not yet been set. In the latter case e is deleted later when reached by the corresponding merge step because $F_i[e] =$ **true**. The MergeStep(Q) procedure is shown in Figure 3.

It is now easy to implement the remainder of the priority queue operations. The operations Update(Q, L) should associate a new processor P_i with the pipeline whose task is to merge L with the elements already in the queue. In order to guarantee that the new processor has something to merge, a MergeStep(Q) is performed to bring at least one new element into Q_{i-1}. The new processor then associates itself with the pipeline, and initializes the set of forbidden elements and the pointer arrays $\overline{Q}_i$ and $\overline{L}_i$. The operation is shown in Figure 4.

A DeleteMin(Q) is even easier. A call to MergeStep(Q) brings a new element into

Procedure UPDATE(Q, L)
MERGESTEP(Q); /* perform a merge step to ensure that last queue is non-empty */;
Associate a new processor P_i with Q and connect it to the pipeline;
$L_i \leftarrow L$;
Initialize F_i, $\overline{L}_i$ and $\overline{Q}_i$;
End of Procedure

Figure 4: The UPDATE(Q, L) operation.

Procedure DELETEMIN(Q)
MERGESTEP(Q);
if P_i is the last processor **then**
  MINELT $\leftarrow Q_i$.first; Q_i.remfirst;
fi;
End of Procedure

Procedure DELETE(Q, e)
MERGESTEP(Q);
if P_i is the last processor **then**
  $F_i[e] \leftarrow$ **true**;
  /* Remove e from Q_i using $\overline{Q}_i[e]$ */
  if $\overline{Q}_i[e] \neq \perp$ **then** Q_i.remove$(\overline{Q}_i[e])$;
fi;
End of Procedure

Figure 5: The DELETEMIN(Q) and DELETE(Q, e) operations.

the output queue of the last processor P_i. The smallest element of Q is the first element of Q_i which is removed and copied to the return cell MINELT. The operation DELETE(Q, e) just makes e forbidden for the last processor. To ensure that the last output queue does not become empty, one call to MERGESTEP is performed. The code for these operations is shown in Figure 5. The final operation EMPTY(Q) simply queries the output queue of the last processor, and writes **true** into the STATUS cell if empty.

For the correctness of the queue operations it only remains to show that processor P_{i+1} only runs out of elements to merge when $Q(i)$, the queue after the ith update, has become empty. We establish the following:

Lemma 1 *The output queue Q_i of processor P_i is non-empty, unless $Q(i)$ is empty.*

Proof. It suffices to show that as long as $Q(i)$ is non-empty, the invariant $|Q_i| > |F_{i+1} \setminus F_i| \geq 0$ holds. Consider the work of processor P_{i+1} at some MERGESTEP. Either $|F_{i+1} \setminus F_i|$ is increased by one, or $|Q_i|$ is decreased by one, but not both, since in the case where P_{i+1}

outputs an element from Q_i this element has been put into F_i at some previous operation, and in the case where P_{i+1} outputs an element from L_{i+1} which was also in Q_i, this element has again been put into F_i at some previous operation. In both cases $|F_{i+1} \setminus F_i|$ does not change when P_{i+1} puts the element into F_{i+1}. The work of P_{i+1} therefore maintains $|Q_i| \geq |F_{i+1} \setminus F_i|$; strict inequality is reestablished by considering the work of P_i which either increases $|Q_i|$ or, in the case where P_i is not allowed to put its element e into Q_i (because $e \in F_{i+1}$), decreases $|F_{i+1} \setminus F_i|$ (because e is inserted into F_i). $\square$

In procedure UPDATE(Q, L) we need to initialize the array of forbidden elements F_i to **false** for all elements, and each of the pointer arrays $\overline{L}_i$ and $\overline{Q}_i$ to $\perp$. There is a well-known solution to do this sequentially in constant time (see for instance [21, pp. 289–290]). The initialization is done in a lazy fashion, and requires two extra arrays of the same size for each array, respectively. The first array will be treated as an array of pointers into the second array which will be a stack of array indices actually "seen". The stack is initially made empty by setting a stack pointer to the bottom of the stack array. Apart from this nothing has to be done for either of the three arrays. When an element is accessed, we can check as follows whether it is encountered for the first time and thus has to be initialized. We look up the pointer of the element in the pointer array. If this is not a valid pointer into the stack, the element has not been seen before (likewise, if the pointer points to a position in the stack above the stack pointer, the element has not been seen before). In this case the element is initialized, its index pushed on the stack, and the pointer in the pointer array set to this position. Only in the case where the pointer points to a valid position in the stack below the stack pointer may the element have been encountered before, and this is the case only if the index which is on the stack actually equals the index of the element.

This completes the proof of Theorem 1.

4.1 A linear space implementation

For the linear pipeline implementation as described above it is possible to reduce the $O(n_0)$ space required per processor for the forbidden sets and the arrays of pointers into the output queues to a total of $O(n_0+m)$ for a sequence of queue operations involving m elements, if we allow concurrent reading. Instead of maintaining the forbidden sets F_i and arrays $\overline{L}_i$ and $\overline{Q}_i$ explicitly, we let each occurrence of an element in the priority queue carry information about its position (whether in some queue Q_j or in L_i), whether it has been forbidden and if so, by which processor. Maintaining for each element a doubly linked list of its occurrences in the data structure makes it possible for processor P_i to determine in constant time whether a given element has been forbidden for processor P_{i+1}, and to remove it in constant time from Q_{i-1} whenever it is output from L_i, and from L_i whenever it is output from Q_{i-1}. In order to insert new elements on these *occurrence lists* in constant time an array of size $O(n_0)$ is needed. For element e this array will point to the most recently inserted occurrence of e (which is still in Q). Occurrences of e appear in the occurrence list of e in the order in which update operations involving e were performed.

At the ith UPDATE(Q, L) operation, each element e of L_i (i.e., of L which is now associated with the new processor P_i) is linked to the front of the occurrence list of e with a label that it belongs to L_i and pointers which allows it to be removed from L_i in constant time. Let us now consider a merge step of processor P_i. When an element e is chosen from Q_{i-1}, P_i looks at the next occurrence of e in e's occurrence list. If this occurrence is in L_i, it is removed, both from L_i and from the list of e-occurrences. This eliminates the need for the $\overline{L}_i$ array. It is now checked whether e is forbidden for P_{i+1} by looking at the next occurrence of e; if it not marked as *forbidden by* P_{i+1}, e is output to Q_i, marked as *forbidden by* P_i. If e was forbidden by P_{i+1} this occurrence is still marked as forbidden by P_i, but not output. If e is chosen from L_i, P_i looks at the previous occurrence of e. If this is in Q_{i-1} it is removed from both Q_{i-1} and from the list of e-occurrences. This eliminates both the need for forbidden sets and the pointer arrays $\overline{Q}_i$. It should be clear that consecutive occurrences of e are never removed in the same merge step, so the doubly linked lists of occurrences are properly maintained also when different processors work on different occurrences of e. Note, however, that all elements in L_i have to be linked into their respective occurrence lists before subsequent merge steps are performed, so concurrent reading is needed. This gives the following variant of the priority queue.

Lemma 2 *The operations* INIT(Q) *and* EMPTY(Q) *take constant time with one processor. The* DELETEMIN(Q) *and* DELETE(Q, e) *operations can be done in constant time by* $|Q|$ *processors. The operation* UPDATE(Q, L) *can be done in constant time by* $1 + |Q| + |L|$ *processors, and assigns one new processor to* Q*. The priority queue can be implemented on the CREW PRAM. The total space consumption is* $O(n_0 + m)$*, where* n_0 *is the maximum number of elements allowed in the queue, and* m *the total number of elements updated.*

5 Dynamically restructuring tree pipeline

In this section we describe how to decrease the work done by the algorithm in Section 4 such that we achieve the result stated in Theorem 2. Before describing the modified data structure, we first make an observation about the work done in Section 4.

Intuitively, the work done by processor P_i is to output elements by incrementally merging its list L_i with the queue Q_{i-1} of elements output by processor P_{i-1}. Processor P_i terminates when nothing is left to be merged. An alternative bound on the work done is the sum of the *distance* each element $e(d)$ belonging to a list L_i travels, where we define the distance to be the number of processors that output $e(d)$. Since the elements $e(d)$ in L_i can be output only by a prefix of the processors $P_i, P_{i+1}, \ldots, P_n$, the distance $e(d)$ travels is at most $n - i + 1$. This gives a total bound on the work done by the processors of $O(mn)$. The work can actually be bounded by $O(n^2)$ due to the fact that elements get annihilated by forbidden sets.

In this section we describe a variation of the data structure in Section 4 that intuitively bounds the distance an element can travel by $O(\log n)$, *i.e.*, bounds the work by $O(m \log n)$. The main idea is to replace the linear pipeline of processors by a binary tree pipeline of processors of height $O(\log n)$.

We start by describing how to arrange the processors in a tree and how to dynamically restructure this tree while adding new processors for each UPDATE-operation. We then describe how the work can be bounded by $O(m \log n)$ and finally how to perform the required processor scheduling.

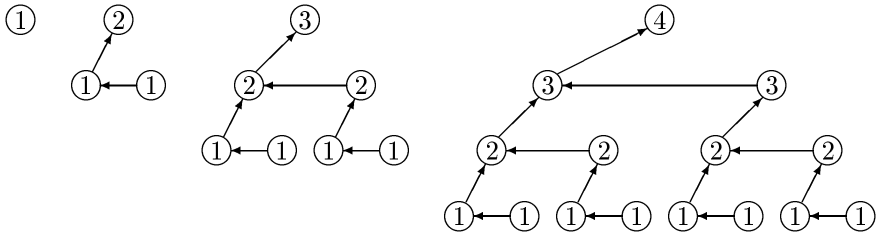

Figure 6: The tree arrangement of processors. Numbers denote processor ranks.

5.1 Tree structured processor connections

To arrange the processors in a tree we slightly modify the information stored at each processor. The details of how to handle the queues and the forbidden sets are given in Section 5.2. Each processor P_i still maintains a list L_i and a set of forbidden elements F_i. The output of processor P_i is still inserted into the processor's output queue Q_i, but P_i now receives input from two processors instead of one processor. As for the linear pipeline we associate two arrays $\overline{Q}_i$ and $\overline{L}_i$ with the queue Q_i and list L_i. The initialization of the array $\overline{L}_i$ is done in the same pipelined fashion as for the linear pipeline.

The processors are arranged as a sequence of perfect binary trees. We represent the trees as shown in Figure 6. A left child has an outgoing edge to its parent, and a right child an edge to its left sibling. The incoming edges of a node v come from the left child of v and the right sibling of v. Figure 6 shows trees of size 1, 3, 7 and 15. Each node corresponds to a processor and the unique outgoing edge of a node corresponds to the output queue of the processor (and an input queue of the parent processor). The rank of a node is the height of the node in the perfect binary tree and the rank of a tree is the rank of the root of the tree. A tree of rank $r+1$ can be constructed from two trees of rank r plus a single node, by connecting the two roots with the new node. It follows by induction that a tree of rank r has size $2^r - 1$.

The processors are arranged in a sequence of trees of rank $r_k, r_{k-1}, \ldots, r_1$, where the ith root is connected to the $i+1$st root as shown in Figure 7. For the sequence of trees we maintain the invariant that

$$r_k \leq r_{k-1} < r_{k-2} < \cdots < r_2 < r_1. \tag{1}$$

When performing an UPDATE-operation a new processor is initialized. If $r_k < r_{k-1}$ the new processor is inserted as a new rank one tree at the front of the sequence of trees as for the linear pipeline. That (1) is satisfied follows from $1 \leq r_k < r_{k-1} < \cdots < r_1$. If $r_k = r_{k-1}$ we link the kth and $k-1$st tree with the node corresponding to the new processor to form a tree of rank $1 + r_{k-1}$. That (1) is satisfied follows from $1 + r_{k-1} \leq r_{k-2} < r_{k-3} < \cdots < r_1$. Figure 7 illustrates the relinking for the case where $r_k = r_{k-1} = 2$ and $r_{k-2} = 4$. Note that the only restructuring of the pipeline required is to make the edge e an incoming edge of the new node w.

The described approach for relinking has been applied in a different context to construct purely functional random-access lists [23]. In [23] it is proved that a sequence of trees

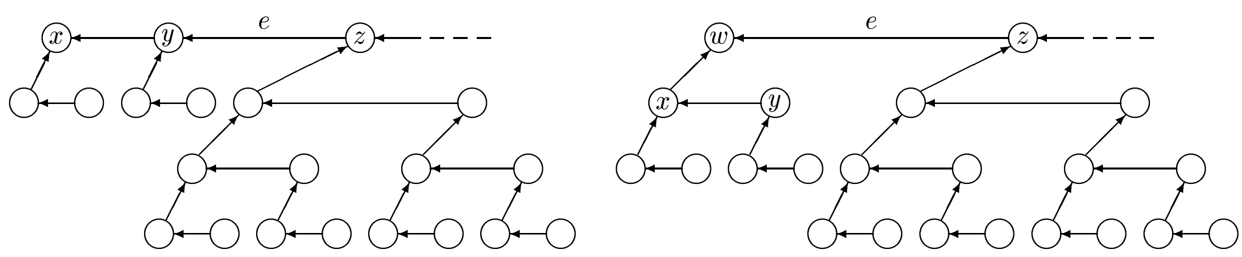

Figure 7: How to restructure the tree when performing UPDATE.

satisfying (1) is unique for a given number of nodes.

5.2 Queues and forbidden sets

We now give the details of how to handle the output queues and the forbidden sets and how to implement the MERGESTEP-operation. Let P_j be a processor connected to a processor P_i, $i > j$, by the queue Q_j. For the tree pipeline processor P_j is only guaranteed to output a subset of the elements in $Q(j)$ in non-decreasing order. For the linear pipeline processor P_j outputs exactly the set $Q(j)$.

Assume that processors P_i and P_j were created as a result of the ith and jth, respectively, UPDATE operations. Let J_j denote the set of elements deleted by DELETE and DELETEMIN operations between the jth and ith UPDATE operations. The important property of J_j is that J_j are the elements that can be output by P_j but are illegal as input to P_i, because they already have been deleted prior to the creation of P_i. We represent each J_j as a Boolean array. How to handle J_j when restructuring the pipeline is described later in this section. To guarantee that Q_j does not contain any illegal input to P_i we maintain the invariant

$$Q_j \cap (F_i \cup J_j) = \emptyset. \tag{2}$$

Our main invariant for the connection between processors P_j and P_i while processor P_j still has input left to be considered is (3), which intuitively states that P_j has output more elements than the number of elements output by P_i plus the elements deleted before the connection between P_j and P_i is created.

$$|(F_i \cup J_j) \setminus F_j| < |F_j \setminus (F_i \cup J_j)|. \tag{3}$$

We now describe how to implement the MERGESTEP-operation such that the invariants (3) and (2) remain satisfied. The basic idea of the implementation is the same as for the linear pipeline. Processor P_j first selects the element v with smallest key in L_j and the input queues of P_j in constant time. If no v exists processor P_j terminates. Otherwise, all occurrences of v are removed from L_j and the input queues Q_ℓ of P_j using the arrays $\overline{L}_j$ and $\overline{Q}_\ell$ respectively, and v is added to F_j. If Q_j is an input queue of P_i and $v \notin F_i \cup J_j$, then v is inserted in Q_j. If $v \in F_i \cup J_j$, then v is not inserted into Q_j, since otherwise (2) would be violated. If $v \in F_i$, then v has already been output by processor P_i and we can safely annihilate v. If $v \in J_j$, then v has been deleted from $Q(k)$, $j \le k < i$, and we can again safely annihilate v. That (3) is satisfied after a MERGESTEP-operation follows from an argument similar to the one given in the proof of Lemma 1 for the linear pipeline: the work done by processor P_i, when inserting a new element into F_i, either increases the left-hand side of (3) by one or decreases the right-hand side of (3) by one and thereby makes the inequality $\leq$. The $<$ is reestablished by processor P_j which inserts a new element into F_j (this either decreases the left-hand side of (3) by one or increases the right-hand side of (3) by one).

Invariant (3) allows us to let Q_j become empty throughout a MERGESTEP-operation, without violating the correctness of the operation and without P_j being terminated. The reason is that $F_j \setminus (F_i \cup J_j) \neq \emptyset$ implies that there exists an element v that has been output by P_j ($v \in F_j$) that neither has been deleted from the data structure before P_i was created ($v \notin J_j$) nor has been output by P_i ($v \notin F_i$). If Q_j becomes empty, v can only be stored in an output queue of a processor in the subtree rooted at P_i due to how the dynamic relinking is performed, *i.e.*, v appears in a Q_k, $j < k < i$. It follows that v has to be output by P_i (perhaps with a smaller key because v gets annihilated by an appearance of v with a smaller key) before the next element to be output by P_j can be output by P_i. This means that P_i can safely skip to consider input from the empty input queue Q_j, even if Q_j later can become non-empty. Note that (3) guarantees that a queue between P_{i-1} and P_i always is non-empty.

We now describe how to implement the UPDATE-operation. The implementation is as for the linear pipeline, except for the dynamic relinking of a single connection (edge e in Figure 7) which is done after the MERGESTEP-operation and the initialization of the new processor. Assume that P_i is the newly created processor. That Q_{i-1} satisfies (3) and (2) follows from the fact that $J_{i-1} \subset F_{i-1}$ (the MERGESTEP-operation at the beginning of the UPDATE-operation implies that at least one element output by P_{i-1} has not been deleted) and $F_i = \emptyset$. What remains to be shown is how to satisfy the invariants for the node P_j when Q_j, $j < i$, is relinked to become an input queue of P_i.

When Q_j is relinked, P_j has output at least $|J_j|+1$ elements in total ($|J_j|$ for delete operations and one from the MERGESTEP-operation at the beginning of the UPDATE-operation). Because $F_i = \emptyset$ and $i > j$, it follows that (3) is satisfied after the relinking. To guarantee that (2) is satisfied we have to update J_j according to the definition and to update the queue Q_j as follows

$$Q_j \leftarrow Q_j \setminus J_j.$$

Since Q_j and J_j can be arbitrary sets it seems hard to do this updating in constant time without some kind of precomputation. Note that the only connections which can be relinked are the connections between the tree roots. Our solution to this problem is as follows: For each MERGESTEP-operation, we mark the deleted element v as *dirty* in all the output queues Q_j where P_j is a root and mark v dirty in J_j. Whenever a queue Q_j is relinked we just need to be able to delete all elements marked dirty from Q_j in constant time. When inserting a new element into a queue Q_j it can be checked if it is dirty or not by examining if the element is in J_j.

A reasonably simple solution to the marking problem, as well as to the insertion of the dirty elements in J_j, is the following. First, note that each time Q_j is relinked it is connected to a node having rank one higher, *i.e.*, we can use this rank as a time stamp t. We represent a queue Q_j as a linked list of vertices, where each vertex v has two time stamped links to vertices in each direction from v. The link with the highest time stamp $\leq t$ is the current link in a direction. A link with time stamp $t+1$ is a link that will become active when Q_j is relinked, *i.e.*, we implicitly maintain two versions of the queue: The current version and the version where all the dirty vertices have been removed. The implementation of the marking procedure is straightforward. To handle the Boolean array J_j, it is sufficient for each **true** entry to associate a time stamp. A time stamp equal to $t+1$ implies that the entry in J_j is dirty. As described here the marking of dirty vertices requires concurrent read to know the deleted element, but by pipelining the dirty marking process along the tree roots from left to right, concurrent read can be avoided. This is possible because the relinking of the tree pipeline only affects the three leftmost roots in the tree pipeline.

We now argue that the described data structure achieves the time bounds claimed in Theorem 2, *i.e.*, that the work done by the processors for the MergeStep-operations is $O(m \log n)$. Elements can travel a distance of at most $2 \log n$ in a tree (in the sense mentioned in the beginning of this section) before they reach the root of the tree. The problem is that the root processors move elements to lower ranked nodes, but the total distance to travel increases at most by $2 \log n$ for each of the n MergeStep-operations. This is true, because the increase in the total distance to travel along the root path results in the telescoping sum

$$2(r_1 - r_2) + 2(r_2 - r_3) + \cdots + 2(r_{k-1} - r_k)$$

which is bounded by $2 \log n$. We conclude that the actual merging work is bounded by $O(2m \log n + 2n \log n)$, *i.e.*, $O(m \log n)$.

5.3 Processor scheduling

What remains is to divide the $O(m \log n)$ work among the available processors on an EREW PRAM. Assuming that $O(\frac{m \log n}{n})$ processors are available, the idea is to simulate the tree structured pipeline for $O(\log n)$ time steps, after which we stop the simulation and in $O(\log n)$ time eliminate the (simulated) terminated processors, and reschedule. By this scheme a terminated processor is kept alive for only $O(\log n)$ time steps, and hence no superfluous work is done. In total the simulation takes linear time.

6 Further applications and discussion

The improved single-source shortest path algorithm immediately gives rise to corresponding improvements in algorithms in which the single-source shortest path problem occurs as a subproblem. We mention here the assignment problem, the minimum-cost flow problem, (for definitions see [1]), and the single-source shortest path problem in planar digraphs. As usual, n and m denote the number of vertices and edges of the input graph, respectively. Note that the minimum-cost flow problem is P-complete [14] (i.e., it is very unlikely that it has a very fast parallel solution), while the assignment problem is not known to be in NC (only an RNC algorithm is known in the special case of unary weights [20, 22], and a weakly polynomial CRCW PRAM algorithm that runs in $O(n^{2/3} \log^2 n \log(nC))$ time with $O(n^{11/3} \log^2 n \log(nC))$ work [13] in the case of integer edge weights in the range $[-C, C]$).

The assignment problem can be solved by n calls to Dijkstra's algorithm (see e.g. [1, Section 12.4]), while the solution of the minimum-cost flow problem is reduced to $O(m \log n)$

calls to Dijkstra's algorithm (see e.g. [1, Section 10.7]). The best previous (strongly polynomial) algorithms for these problems are given in [11]. They run on an EREW PRAM and are based on their implementation of Dijkstra's algorithm: the algorithm for the assignment problem runs in $O(n^2 \log n)$ time using $O(nm + n^2 \log n)$ work, while the algorithm for the minimum-cost flow problem runs in $O(nm \log^2 n)$ time using $O(m^2 \log n + nm \log^2 n)$ work. Using the implementation of Dijkstra's algorithm presented in this paper, we can speedup the above results on a CREW PRAM. More specifically, we have a parallel algorithm for the assignment problem that runs in $O(n^2)$ time using $O(nm \log n)$ work, and a parallel algorithm for the minimum-cost flow problem that runs in $O(nm \log n)$ time and $O(m^2 \log^2 n)$ work.

Greater parallelism for the single-source shortest path problem in the case of planar digraphs can be achieved by plugging our implementation of Dijkstra's algorithm (Theorem 3(ii)) into the algorithm of [29] resulting in an algorithm which runs in $O(n^{2\epsilon} + n^{1-\epsilon})$ time and performs $O(n^{1+\epsilon})$ work on an EREW PRAM, for any $0 < \epsilon < 1/2$. With respect to work, this gives the best (deterministic) parallel algorithm known for the single-source shortest path problem in planar digraphs that runs in sublinear time.

Acknowledgements. We are grateful to Volker Priebe for his careful reading of the paper and his insightful comments.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice-Hall, 1993.

[2] Gerth Stølting Brodal. Priority queues on parallel machines. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1097 of *Lecture Notes in Computer Science*, pages 416–427. Springer Verlag, Berlin, 1996.

[3] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 52–58, 1996.

[4] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority data structure with applications. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, pages 689–693, 1997.

[5] Danny Z. Chen and Xiaobo Hu. Fast and efficient operations on parallel priority queues (preliminary version). In *Algorithms and Computation: 5th International Symposium, ISAAC '93*, volume 834 of *Lecture Notes in Computer Science*, pages 279–287. Springer Verlag, Berlin, 1994.

[6] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[7] Stephen Cook, Cynthia Dwork, and Rüdiger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[8] Sajal K. Das, Maria C. Pinotti, and Falguni Sarkar. Optimal and load balanced mapping of parallel priority queues in hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 7:555–564, 1996. Correction ibid. p. 896.

[9] Paul F. Dietz and Rajeev Raman. Very fast optimal parallel algorithms for heap construction. In *Proc. 6th Symposium on Parallel and Distributed Processing*, pages 514–521, 1994.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[12] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[13] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. *Journal of Algorithms*, 14(2):180–213, 1993.

[14] L.M. Goldschlager, R. Shaw, and J. Staples. The maximum flow problem is LOGSPACE complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[15] Torben Hagerup and Christine Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.

[16] Y. Han, V. Pan, and J. Reif. Algorithms for computing all pair shortest paths in directed graphs. In *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 353–362, 1992.

[17] Peter Høyer. A general technique for implementation of efficient priority queues. In *Proc. 3rd Israel Symposium on Theory of Computing and Systems*, pages 57–66, 1995.

[18] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[19] Richard M. Karp and Vijaya Ramachandran. *Parallel Algorithms for Shared-Memory Machines*, volume A of *Handbook of Theoretical Computer Science*, chapter 17, pages 869–942. Elsevier, 1990.

[20] R. Karp, E. Upfal, and A. Wigderson. Constructing a maximum matching is in Random NC. *Combinatorica*, 6:35–38, 1986.

[21] Kurt Mehlhorn. *Data Structures and Algorithms*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1984.

[22] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[23] Chris Okasaki. Purely functional random-access lists. In *Functional Programming Languages and Computer Architecture*, pages 86–95, 1995.

[24] Richard C. Paige and Clyde P. Kruskal. Parallel algorithms for shortest path problems. In *Int. Conference on Parallel Processing*, pages 14–20, 1985.

[25] Maria Cristina Pinotti, Sajal K. Das, and Vincenzo A. Crupi. Parallel and distributed meldable priority queues based on binomial heaps. In *Int. Conference on Parallel Processing*, 1996.

[26] Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40:33–40, 1991.

[27] Maria Cristina Pinotti and Geppino Pucci. Parallel algorithms for priority queue operations. *Theoretical Computer Science*, 148(1):171–180, 1995.

[28] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *Proc. 6th Symposium on Parallel and Distributed Processing*, pages 490–496, 1994.

[29] Jesper L. Träff and Christos D. Zaroliagis. Simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *A Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*, volume 1117 of *Lecture Notes in Computer Science*, pages 183–194. Springer Verlag, Berlin, 1996.