Simple parallel algorithms for the replacement edge problem and related problems on spanning trees

Jyrki Katajainen and Jesper Larsson Träff
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark

Abstract

Let G be a connected, undirected, and weighted graph with n vertices and m edges. Further, let S be a spanning tree of G and T a minimum spanning tree of G. In this paper we present parallel algorithms for solving the following problems:

(a) Given G and T, for each edge e of T, determine a replacement edge f by which e should be replaced to create a new minimum spanning tree if e is removed from G.

(b) Given G and T, find a most vital edge of G with respect to T, that is, an edge whose removal has the largest impact on the weight of T.

(c) Given G and S, determine whether S is a minimum spanning tree.

(d) Given G and T, compute, for each edge e of G, by how much the weight of e can change without affecting the minimality of T.

The algorithms run in $O(\log n)$ time and $O(m)$ space with m processors. The problems (a), (b), and (d) are solved on a MINIMUM CRCW PRAM, whereas the problem (c) is solved on a CREW PRAM. The algorithms utilizes a simple technique for computing functions defined on paths in rooted trees that might be of independent interest.

1 Introduction

We consider the following problem on minimum spanning trees of weighted graphs. Given a minimum spanning tree (MST), for each tree edge, determine a non-tree edge such that whenever the tree edge is removed and replaced with its corresponding non-tree edge, a new MST results. A very efficient sequential algorithm exists for this and closely related problems [5, 13, 14]. In this paper a new, fast parallel algorithm for this replacement edge problem is presented. The algorithm is based on a simple preprocessing of the spanning tree which allows an efficient computation of functions defined on paths joining a pair of vertices.

Actually, we can solve a slightly more general problem, where the minimum replacement edge is computed for every edge of a specific spanning tree. The formal definition of the problem is as follows. Let $G = (V, E)$ be a connceted, undirected, and weighted graph with vertex set V, edge set E and weight-function $weight{:}E \rightarrow I\!R$. Let n and m denote the number of vertices and edges of G, respectively. We assume that $m \geq n$. Let $S = (V, E_S)$ be a spanning tree of G. The *replacement edge problem* is to determine, for each tree edge $e \in E_S$, a non-tree edge $f \in E \setminus E_S$ of least weight among the non-tree edges with the property that, if e is replaced with f in S, a spanning tree in the graph $G' = (V, E \setminus \{e\})$ results. If such an edge exists, f is called a (minimum) *replacement edge* of e. As showed by Tarjan [13], the replacement edge problem can be solved sequentially in $O(m\alpha(m, n))$ time and space (where α is a functional inverse of Ackermann's function) by using path compression techniques. In this paper we present a fast, parallel algorithm for the replacement edge problem. Our algorithm runs in $O(\log n)$ time and $O(m)$ space with m processors on a CRCW PRAM.

The heart of our solution is a simple algorithm for computing path products on a tree. Given a tree T, any two vertices v and w of T induce a unique path $T(u, v)$ in T. For any vertex pair $\{v, w\}$, their *path product* with respect to some associative operation is the product of labels of the edges on path $T(v, w)$. In our case we are given a collection of vertex pairs and the task is to compute all the products along the corresponding paths efficiently in parallel. A related problem studied by us is that of performing path labellings. Here we are given a collection of vertex pairs and the task is to label the edges on the corresponding paths with some label so that, if an edge gets more than one label, only the minimum one is preserved.

An algorithm for the replacement edge problem for spanning trees obviously solves the replacement edge problem for MSTs. By using path product and path labelling techniques, a number of related problems concerning MSTs can be solved as efficiently as the replacement edge problem. In this paper we consider the problems of

(a) finding the most vital edge(s) of a graph with respect to an MST,

(b) verifying that a given spanning tree is minimum,

(c) performing sensitivity analysis of an MST.

The *most vital edge problem*, which consists of finding an edge of a graph G whose removal has the largest impact on the weight of any MST of G (the weight of a tree being the sum the weights of its edges), has recently been considered in isolation [7, 8, 11]. It is worth noting that the algorithms proposed implicitly solve the replacement edge problem for MSTs, and that the replacement edge problem is a bottleneck in these solutions. Our new algorithm provides a considerably faster solution to the most vital edge problem than previous parallel algorithms [8, 11]. For a comparison, the two parallel algorithms given by Hsu *et al.* [8] run in time $O(n^{1+\varepsilon})$ with $n^{1-\varepsilon}$ processors, for $0 < \varepsilon < 1$, and $O(m \log(m/N)/N + n\alpha(m,n)\log(m/n))$ with $N \leq (m \log m)/(n\alpha(m,n)\log(m/n))$ processors, respectively. These algorithms are neither fast nor very parallel. An even slower algorithm was given by Suraweera and Maheshwari [11], running in $O(m)$ time using $m+n$ processors.

Given a graph G and a spanning tree S of G, the *MST verification problem* is that of determining whether S is an MST of G. For the verification of MSTs, a linear-time algorithm was presented by Dixon *et al.* [5]. However, the best known parallel algorithm uses the naive approach in which the whole MST is re-computed. The fast parallel algorithms for finding MSTs (see, *e.g.*, [3, 10] seem to require a CRCW PRAM. Our verification algorithm runs on a CREW PRAM with the same efficiency.

Given a graph G and its MST T, the *MST sensitivity analysis problem* is to compute, for each edge e of G, by how much the weight of e can change without affecting the minimality of T. For the sensitivity analysis of MSTs, an optimal sequential algorithm (with an unknown complexity) was given by Dixon *et al.* [5]. As far as we know, our algorithm is the first logarithmic algorithm for this problem.

When solving the replacement edge, the most vital edge, and the MST sentitivity analysis problems, we use a MINIMUM CRCW PRAM as our model of computation. In this model write conflicts are resolved as follows: If two or more processors concurrently try to write a value to the same memory location, the smallest value is written into that location. From a practical point of view this model is (in all likelihood) unrealistically strong. By the well-known simulation theorem (see, *e.g.*, [6]), the MINIMUM CRCW PRAM model, like the CREW PRAM model, can be simulated with only logarithmic time overhead by the EREW PRAM model. Our results can therefore immediately be translated to an EREW PRAM with a logarithmic decrease in efficiency. The use of the MINIMUM CRCW PRAM model considerably simplifies the exposition, and is justified, since we have been unable to give a faster algorithms running directly in the EREW PRAM model.

The rest of the paper is organized as follows. In Section 2 a simple technique for computing path products on (rooted) trees is introduced. In Section 3 a similar technique for labelling paths on (rooted) trees is presented. These techniques are needed in Section 4, where a solution to the replacement edge problem is given. Some obvious applications of both the replacement edge algorithm and the tree preprocessing techniques are also given in Section 4. In Section 5 open problems and possible improvements are discussed.

2 Computing path products on trees

Given a tree T of n vertices, one is often interested in computing some function defined on the labels of edges on the unique path joining two vertices, like summing the weights of the edges or finding an edge of minimum weight. In our case we are given a collection of m vertex pairs, and we want to compute in parallel the product, with respect to any associative operation, of the edge labels on the path joining the vertices of each pair. In Subsection 2.1 we show how all these products can be computed in $O(\log n)$ time with m processors after $O(\log n)$ preprocessing with n processors on a CREW PRAM. The drawback of this solution is that it requires $O(n \log n + m)$ space. In Subsection 2.2 we reduce the space requirement to $O(n + m)$ without affecting the time performance. Our approach is reminiscent of that for preprocessing and answering queries on Euler tours used in [15].

2.1 A solution by using non-linear space

Let $T = (V, E)$ be a rooted tree with root $r \in V$. We assume that the representation T consists of the *adjacency lists* of the vertices such that

(a) each vertex is represented by a symbolic integer name, *i.e.*, $V = \{1, 2, \ldots, n\}$,

(b) each undirected edge $\{v, w\} \in E$ is present twice in the representation of T, once as arc (v, w) in the adjacency list of v and once as arc (w, v) in the adjancency list of w, and

(c) we have a direct access from arc (v, w) to arc (w, v) and *vice versa.*

When this representation is available, *parent* $p(v)$ of each vertex v; *level* $\ell(v)$ of each vertex v, *i.e.*, its distance from the root r; as well as *lowest common ancestor* $lca(v, w)$ of two vertices v and w can be computed efficiently (see, for example, [9, Chapter 3]).

Let $T(v, w)$ denote the path in T from vertices v to w; that is, $T(v, w)$ is the *ordered* collection vertices and edges met when traversing from v to w. Further, assume that the edges of T are labelled and let $\odot$ be any associative operation defined on the labels. Our goal is to preprocess T such that the product of edge labels on any path $T(v, w)$ can be computed efficiently. Let *product*$(T(v, w))$ denote this product; that is, if path $T(v, w)$ contains vertices $v = v_0, v_1, \ldots, v_k = w$, *product*$(T(v, w)) = \bigodot_{i=0}^{k-1} \mathit{label}(v_i, v_{i+1})$.

It is obvious that *product*$(T(v, w)) =$ *product*$(T(v, u)) \odot$ *product*$(T(u, w))$, if $u = lca(v, w)$. By using this formula, the product for any path on T is readily computed from the partial products. Let $p^k(v)$ denote the ancestor of vertex $v \in V$ that lies k levels from v. Next we show how the products upwards, *i.e.*, for the paths of the form $T(v, p^k(v))$, $l > 0$, can be computed. By using an analogous construction, which is left for an interested reader, the products downwards are obtained.

Now we associate with each vertex $v \in V$ two arrays P_v and S_v, each of size $\lfloor \log n \rfloor$. For $0 \leq i < \lfloor \log n \rfloor$, $\mathrm{P}_v[i]$ will be a pointer 2^i levels upwards in the tree, and $\mathrm{S}_v[i]$ will be the product of edge labels along the path from v to the vertex pointed by $\mathrm{P}_v[i]$. This information can be computed by using pointer jumping on rooted tree T. The procedure for computing P_v and S_v, for all vertices $v \in V$, looks as follows.

```
**procedure** PREPROCESS
**par** $v \in V$ **do**
    $\mathrm{P}_v[0], \mathrm{S}_v[0] \leftarrow p(v), \mathit{label}(v, p(v))$
    **for** $i = 1$ **to** $\lfloor \log n \rfloor$ **do**
        $w \leftarrow \mathrm{P}_v[i-1]$
        $\mathrm{P}_v[i], \mathrm{S}_v[i] \leftarrow \mathrm{P}_w[i-1], \mathrm{S}_v[i-1] \odot \mathrm{S}_w[i-1]$
```

Using n processors the procedure obviously runs in $O(\log n)$ time. Only concurrent reading is needed, so the preprocessing can be accomplished on a CREW PRAM.

Using the information in P_v and S_v, we can answer any path-product query in logarithmic time. The procedure below computes the product for path $T(v, u)$, in which u is the ancestor of v that lies k levels from v, $k > 0$.

```
**function** MULTIPLY($T(v, p^k(v))$:path):label
$i, j \leftarrow 0, k$
**while** even($j$) **do** $i, j \leftarrow i + 1, j$ **div** 2
$\mathrm{Q} \leftarrow \mathrm{S}_v[i]$; $w \leftarrow \mathrm{P}_v[i]$; $i, j \leftarrow i + 1, j$ **div** 2
**while** $j > 0$ **do**
    **if** odd($j$) **then** $\mathrm{Q}, w \leftarrow \mathrm{Q} \odot \mathrm{S}_w[i], \mathrm{P}_w[i]$
    $i, j \leftarrow i + 1, j$ **div** 2
**return**(Q)
```

The time complexity of procedure MULTIPLY is proportional to the number bits in the binary representation of k, which is less than $\lceil \log n \rceil$. With concurrent read capabilities, many processors can simultaneously compute a path product in logarithmic time.

2.2 A linear-space solution

In the data structure of the previous subsection we associated an array of size $O(\log n)$ with each vertex of the rooted tree $T = (V, E)$. Since T contains n vertices, the space requirements are in total $O(n \log n)$. The main idea, when reducing this to $O(n)$, is to associate a logarithmic array with only $O(n/\log n)$ vertices. However, these vertices have to be chosen carefully so that the time requirements will not be increased asymptotically.

Assume that each vertex v of T knows its level $\ell(v)$ in T. Now we mark each vertex with the number $\ell(v) \bmod \lceil \log n \rceil$. We say that the vertices with the same number belong to the same *class*. Then we sort the vertices with respect to their numbers. This sorting of small integers can be done in $O(\log n)$ time with $O(n/\log n)$ processors [4]. After sorting, it is an easy matter to determine the class with the smallest number of vertices. Let us mark all the vertices in this particular class and also the root of T, if not marked already. Let V_x denote the set of marked vertices of V. Observe that the cardinality of V_x is less than $n/\log n + 1$.

Next we build an auxiliary tree $T' = (V_x, E_x)$ above tree T. Hence, the vertices of T' are just the vertices in V_x. Tree T' is built such that there is a direct access (pointer) from a vertex v in T to the corresponding vertex in T' and *vice versa*. Edge $\{v, u\}$ is included in E_x whenever path $T(v, u)$ exists and it does not contain any marked vertices. The edges of E_x are labelled with *product*$(T(v, u))$. The construction of T' is easily done in $O(\log n)$ time with n processors. We simply allocate one processor for each marked vertex of T, and let those to search for their closest marked ancestor. The corresponding products can be computed at the same time. Due to the construction of V_x, each processor will finish its search after $O(\log n)$ steps. The full representation of T' is obtained by sorting (cf. [12, 15]). After this, tree T' is preprocessed as described in Subsection 2.1. When doing this, we also store with each pointer $P_v[i]$ the last edge on the path from v to the ancestor pointed by $P_v[i]$.

To answer path-product queries, we partition each path $[T(v, p^k(v))$ into three pieces: $T(v, x)$, $T(x, y)$, and $T(y, p^k(v))$, where x is the closest marked ancestor of v and y is the closest marked descendant of $p^k(v)$ on path $T(v, p^k(v))$. (Observe that some of these paths might be of the form $T(u, u)$, but these can be ingnored from further considerations.) Vertex x is easily found by traversing T upwards until a marked vertex is encountered. When x and z, which is the closest marked ancestor of $p^k(v)$, are available y can be found by using T': find the logarithmic pointer chain from x to z and pick up y from the last pointer of this chain. The product of $T(y, p^k(v))$ is again computed by traversing T upwards. The product of $T(x, y)$ is computed in T'. When the products of these subpaths are computed, the product of the whole path is obtained in constant time.

It is easily seen that the processing of a path-product query requires $O(\log n)$ time with n processors. In tree T we need only parent pointers.

The full data structure of Subsection 2.1 is needed for tree T', but since it contains $O(n/\log n)$ vertices, the space consumption is reduced to $O(n)$.

The discussion of this section is summarized in the following

Theorem 1: Given a tree T of n vertices and a collection $\{p_1, p_2, \ldots, p_m\}$ of m paths on T. All path products $\mathit{product}(p_i)$, $i = 1, \ldots, m$, can be computed in $O(\log n)$ time with m processors, after $O(\log n)$ preprocessing with n processors on a CREW PRAM. The total amount of space required for these computations is $O(n + m)$.

Proof: The rooted version of T can be constructed in $O(\log n)$ time and $O(n)$ space with $O(n/\log n)$ processors (see [9, Chapter 3]). This construction gives us the parent pointer $p(v)$ for each vertex v. We need also the functions $\ell(v)$, giving the level of vertex v, and $\mathit{lca}(v, w)$, giving the lowest common ancestor of two vertices v and w. These functions can be evaluated in constant time, after a proper preprocessing taking $O(\log n)$ time and $O(n)$ space with $O(n/\log n)$ processors (see [9, Chapter 3]). The other parts of the claim follow from the above discussion. $\square$

3 Performing path labellings on trees

In this section we consider the problem of labelling paths on a tree of n vertices. We are given a collection C of m paths on the tree and a label associated with each path. The labels are drawn from some totally ordered universe. We use ∞ to denote the largest element of this universe. For the sake of simplicity, we assume that all the path labels are distinct. The problem is to label each edge on the given paths with the corresponding label such that, if an edge will get many labels, only the smallest one is preserved with it. Our purpose in this section is to show how the labels can be assigned in $O(\log n)$ time with m processors.

As in Section 2, it is enough to solve the problem for rooted trees. Therefore, let T be a rooted tree represented by the adjacency lists (with the additional pointers). We consider here only how the path labellings are performed for the paths of the form $T(v, p^k(v))$. The general problem is solved by dividing each path $T(v, w)$ into two pieces by using the lowest common ancestor of v and w.

Assume that T has been preprocessed such that each vertex v has the array P_v of pointers upwards in the tree. Associate with each vertex v one more array E_v of size $\lfloor \log n \rfloor$ to be used for labelling (sub)paths upwards in the tree starting at v. More specifically, for each vertex v, some label in $\mathrm{E}_v[i]$, $0 \leq i < \lfloor \log n \rfloor$ will denote that all edges on the path from v to the ancestor pointed by $P_v[i]$ are to be labelled by that label. Initially, we give the label ∞ to $\mathrm{E}_v[i]$, for all $0 \leq i < \lfloor \log n \rfloor$ and for all $v \in V$. This initialization takes $O(\log n)$ time with n processors.

In the algorithm to be presented next, many processors perform labellings in parallel. When a processor tries to update a label, it uses the rule that the update is performed only in the case that the written value is smaller that the current value. At some position, two or more processors might try to update the same label. Here we rely on the write conflict resolution rule of the MINIMUM CRCW PRAM which guarantees that the label with the minimum value will be stored in that place. These two rules guarantee that the correctness of our algorithm.

To perform the path labellings, we allocate first one processor for each path in C. Each processor decomposes its path $T(v, p^k(v))$ into subpaths whose size is a power of 2. This computation is guided by the 1-bits in the binary representation of l, as done in procedure PRODUCT in Subsection 2.1. The following procedure is executed for all paths of C in parallel.

```
**procedure** DECOMPOSE($T(v, p^k(v))$:path, $a$: label)
$w, i, j \leftarrow v, 0, k$
**while** $j > 0$ **do**
    **if** odd($j$) **then** $\mathrm{E}_w[i]$, $w \leftarrow a$, $\mathrm{P}_w[i]$
    $i, j \leftarrow i + 1, j$ **div** 2
```

To push down the labellings performed so far, we allocate one processor for each vertex of T and process the paths of length 2^i in $\lfloor \log n \rfloor$ rounds, starting from the longest. At each round the longest paths are halved and each of them gives its label to the shorter subpaths. Due to concurrent writing, the smallest label assigned to any subpath will survive for the next round. More formally, the procedure for pushing down the labels is as follows:

```
**procedure** PUSHDOWN
**par** $v \in V$ **do**
    **for** $i = \lfloor \log n \rfloor$ **downto** 1 **do**
        **if** $\mathrm{E}_v[i] < \mathrm{E}_v[i-1]$ **then** $\mathrm{E}_v[i-1] \leftarrow \mathrm{E}_v[i]$
        $w \leftarrow \mathrm{P}_v[i-1]$
        **if** $\mathrm{E}_v[i] < \mathrm{E}_w[i-1]$ **then** $\mathrm{E}_w[i-1] \leftarrow \mathrm{E}_v[i]$
```

In the **for**-loop of procedure PUSHDOWN above the following invariant is maintained:

> Label $E_v[i]$ has the least value among all paths containing v and starting 2^j, $j > i$, levels downwards in the tree.

It is easy to prove by induction on $\lfloor \log n \rfloor - i$ that the invariant holds. Hence, upon termination, $\mathrm{E}_v[0]$ contains the label for each tree edge $\{v, p(v)\}$. It is also clear that this push down operation takes $O(\log n)$ time with n processors.

By using a similar technique as in the previous section, the amount of storage space needed for the path labellings can be reduced from $O(n \log n + m)$ to $O(n + m)$. First, the auxiliary tree T' is constructed. Second, each path $T(v, p^k(v))$ is divided into three parts $T(v, x)$, $T(x, y)$, and $T(y, p^k(v))$, where x and y are as in Subsection 2.2. Third, the path labellings for the paths $T(v, x)$ and $T(y, p^k(v))$ are done by using the parent pointers in T, whereas $T(x, y)$ is labelled in T' as described above. Finally, the labels in T' are pushed down to T by assigning one processor for each edge $\{x, y\}$ of T'. These processors simply follow the parent pointers from x to y in T.

To sum up, we have

Theorem 2: Given a tree T of n vertices, a collection $\{p_1, p_2, \ldots, p_m\}$ of m paths on T, and a label $\mathit{label}(p_i)$ associated with each path p_i, $i = 1, \ldots, m$. Each edge e of T can be labelled with the minimum label among those in $\{\mathit{label}(p_j) \mid e \in p_j, j \in \{1, \ldots, m\}\}$ in $O(\log n)$ time and $O(n + m)$ space with $n + m$ processors on a MINIMUM CRCW PRAM.

4 Applications

In this section we present various applications for the path computations given in Sections 2 and 3. We consider here the replacement edge problem

(Subsection 4.1), the most vital edge problem (Subsection 4.2), the MST verification problem (Subsection 4.3), and the MST sensitivity analysis problem (Subsection 4.4). For all the problems we obtain parallel algorithms which run in time $O(\log n)$ with m processors, where n is the number of vertices and m the number of edges in the input graph.

In order to avoid repetitions, we assume throughout of this section that $G = (V, E)$ is a connected, undirected, and weighted graph, $S = (V, E_S)$ a spanning tree of G, and $T = (V, E_T)$ a minimum spanning tree of G. The edges of G are given in an array so that we have a contant time access to them. The trees S and T should be represented by the adjacency lists of the vertices as described in Subsection 2.1.

4.1 Finding a replacement for each spanning tree edge

Recall that the replacement edge problem for a spanning tree S in a graph G is that of finding a minimum replacement for every edge e of S such that, if e is removed from S, a new spanning tree results. Our algorithm for finding all minimum replacement edges is based on the following

Fact 1 (cf. [13, Lemma 6]): Let $S = (V, E_S)$ be a spanning tree of a graph $G = (V, E)$. For each tree edge $e \in E_S$, let the set R_e contain any non-tree edge $\{v, w\} \in E \setminus E_S$ if and only if path $S(v, w)$ in S contains e. If R_e is non-empty, then a suitable replacement edge for e is an edge with the minimum weight in R_e. If R_e is empty, then e has no replacement edge.

The fact gives rise to a simple algorithm. For each non-tree edge $\{v, w\}$, we label path $S(v, w)$ in S with the triple $(\mathit{weight}(v, w), \min(v, w), \max(v, w))$. By performing these labelling in parallel as described in Section 3, each tree edge will get its proper replacement. Observe that the labels are all distinct as required and that the triples must be compared lexicographically.

The number of non-tree edges is $m - n + 1$ and the number of vertices in T is n. Therefore, the following result follows directly from Theorem 2.

Theorem 3: Given a graph G with n vertices and m edges, and any of its spanning tree S. For every edge of S, its minimum replacement edge in G can be found in $O(\log n)$ time and $O(m)$ space with m processors on a MINIMUM CRCW PRAM.

4.2 Finding a most vital edge of a graph

Recall that a *most vital edge* (which is unique if all edge weights are different) of a graph G *with respect to a given MST* T is an edge of G whose removal has the largest effect on the weight of T. If G contains bridges (edges whose removal disconnects G), a most vital edge of G can be taken to be either undefined or one of the bridges of G.

For a graph $G = (V, E)$, we use $G - e$ to denote the graph $(V, E \setminus \{e\}$ and $weight(G)$ to denote the weight of a MST of G. By the following fact, the most vital edge problem reduces to that of finding replacement edges for minimum spanning trees.

Fact 2 (cf. [7, Lemma 1]): Let G be a bridgeless graph and T a minimum spanning tree of G. Then one of the edges of T is a most vital edge of G with respect to T.

Proof: A removal of a non-tree edge has no effect on the weight of T. On the other hand, for each tree edge e of T, $weight(G - e)$ is always greater than or equal to the weight of T. So a most vital edge can be chosen among the edges of T. $\square$

The algorithm for finding a most vital edge of a graph G with respect to an MST T is as follows.

1. For all edges of T, find their replacement edges in G. Use here the algorithm of Subsection 4.1.

2. Check whether there exists a tree edge that does not have a replacement. If this is the case, report that a most vital edge is undefined.

3. Otherwise report as a most vital edge, the tree edge e whose replacement f maximizes $weight(f) - weight(e)$. This maximum computation can be carried out in time $O(\log\log n)$ with $O(n/\log\log n)$ processors on a CRCW PRAM (see, *e.g.*, [2]).

If no bridges are found in Step 2, G is bridgeless. After constructing a spanning tree, the computation of all replacement edges therefore provides an alternative (but not very appealing) way of detecting the bridges of a graph (cf. [12]).

To sum up, we have

Theorem 4: Given a graph G with n vertices and m edges, and a minimum spanning tree T of G. A most vital edge of G with respect to T can be found in $O(\log n)$ time and $O(m)$ space with m processors on a MINIMUM CRCW PRAM.

Sometimes the most vital edge problem is formulated such that an MST is not given as a part of input. An edge e is then said to be a *most vital edge* of a graph G *with respect to all MSTs of* G if $weight(G-e) \geq weight(G-f)$, for every edge f of G. To solve this more general problem we use

Fact 3: Let G be a graph and let T be any of its MST. An edge of T is a most vital edge of G with respect to all MSTs of G.

Proof: Assume on the contrary that no edge of T is a most vital edge og G. Let e be an edge of T for which $weight(G-e)$ is maximized. According to our assumption, for each tree edge of T and therefore also for e, there exists an edge f of G for which $weight(G-e) < weight(G-f)$.

Now we divide the consideration into two cases.
Case 1. f is a non-tree edge. Since $weight(G) \leq weight(G-e)$, we have that $weight(G) < weight(G-f)$. However, this is impossible since the weight of any MST of $G-f$ should be equal to the weight of T. Therefore, such an edge f cannot exist.
Case 2. f is a tree edge. Due to our choice of e such a tree edge cannot exist, since $weight(G-e)$ was assumed to the maximum, for all tree edges.
Since both cases lead to a contradiction, the claim must be correct. □

By using the above fact and the existing algorithms for finding an MST of a graph, we can solve the general most vital edge problem.

Corollary 1: Given a graph G with n vertices and m edges. A most vital edge of G with respect to all MSTs of G can be determined in $O(\log n)$ time and $O(m)$ space with m processors on a MINIMUM CRCW PRAM.

Proof: By Fact 3, any MST T of G can be used to find a most vital edge of G. An MST of G can be found in time $O(\log n)$ with m processors on a CRCW PRAM [3, 10]. Thus, Theorem 4 implies the claim. □

4.3 Verification of minimum spanning trees

For the verification, whether a spanning tree is an MST, we use the following

Fact 4 [13, Lemma 5]: Let $G = (V, E)$ be a graph and let $S = (V, E_S)$ be a spnning tree of G. A spanning tree S is minimum if and only if, for each edge $\{v, w\} \in E \backslash E_S$, $weight(v, w) \geq \max\{weight(x, y) \mid \{x, y\}$ is on the path $S(v, w)$ in $S\}$.

Hence, the MST verification problem reduces to that of computing $m - n + 1$ path products with respect to the maximum operation. The algorithms developed in Section 2 can be used here. After computing the path products, it is easy to chech whether the weight of each non-tree edge is actually larger than the corresponding path product. Basically, this is nothing but an AND-computation, that can be carried out in $O(1)$ time with m processors on a CRCW PRAM (see, *e.g.*, [2]). Thus, by Theorem 1 we have

Theorem 5: Given a graph G with n vertices and m edges, and any of its spanning tree S. The verification, whether S is an MST of G, can be accomplished in $O(\log n)$ time and $O(m)$ space with m processors on a CREW PRAM.

4.4 Sensitivity analysis of minimum spanning trees

As a final application we consider the sensitivity analysis of minimum spanning trees. Recall that the problem is to determine, for each edge e of a graph G, how much the weight of e can change before the (weight of) any MST of G changes. A result from [14, Corollary 1] gives a solution to the problem in terms of replacement edges and path queries.

Fact 5 [14, Corollary 1]: Let $T = (V, E_T)$ be an MST of a graph $G = (V, E)$. For each tree edge $e \in E_T$, tree T remains a MST until the weight of e is increased by more than $weight(f) - weight(e)$, where f is a replacement edge of e. For each non-tree edge $g \in E \backslash E_T$, the tree remains a minimum spanning tree until the weight of g is decreased by more than $weight(g) - weight(h)$, where h is a tree edge which is maximum on the tree path induced by g.

Given an MST T, the algorithm below computes the maximum change Δ_e for each edge e of G.

1. For every tree edge e of T, find its replacement edge f in G (cf. Subsection 4.1) and let $\Delta_e = weight(f) - weight(e)$.

2. Preprocess T such that the maximum edge on any path in T can be found efficiently (cf. Section 2).

3. For each non-tree edge $g = \{x, y\}$, find a tree edge h with the maximum weight on path $T(x, y)$ in T (cf. Section 2) and let $\Delta_g = weight(g) - weight(h)$.

By Theorems 1 and 2, the following result is immediate.

Theorem 5: Given a graph G with n vertices and m edges, and an MST T of G. The sensitivity analysis of T in G can be carried out in $O(\log n)$ time and $O(m)$ space with m processors on a MINIMUM CRCW PRAM.

5 Concluding remarks

Parallel algorithms for solving the replacement edge problem for spanning trees and a number of related problems on (minimum) spanning trees were presented. Our algorithms run in $O(\log n)$ time and $O(m)$ space with m processors, n being the number of vertices and m the number of edges in the input graph.

Best sequential algorithms for the same problems run in $O(m\alpha(m, n))$ time or faster, so our parallel algorithms are not work-optimal. By using a considerably more sophisticated preprocessing of the spanning tree, work-optimal parallel algorithms for the problems studied might be possible.

Our parallel algorithms use the MINIMUM CRCW PRAM model. This seems difficult to avoid. Some improvements (with respect to efficiency) are possible by implementing the algorithms directly on an EREW PRAM and using more efficient algorithms known for integer sorting (see, *e.g.*, [1]), instead of general sorting as implicit in the standard simulation.

Acknowledgments

We would like to thank Jens Clausen and Mikkel Thorup for helpful discussions.

References

[1] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 43–472, 1992.

[2] Yrjö Auramo, Jyrki Katajainen, and Juhani Kulmala. Finding the maximum in parallel random access machines. *Bulletin of the EATCS* 52 (1994) 315–334.

[3] Baruch Awerbuch and Yossi Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers* **C-36** (1987) 1258–1263.

[4] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control* **70** (1986) 32–53.

[5] Brandon Dixon, Monika Rauch, and Robert E. Tarjan. Verification and sensitrivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing* **21** (1992) 1184–1192.

[6] David Eppstein and Zvi Galil. Parallel algorithmic techniques for combinatorial computation. *Annual Review of Computer Science* **3** (1988) 233–283.

[7] Lih-Hsing Hsu, Rong-Hong Jan, Yu-Che Lee, Chun-Nan Hung, and Maw-Sheng Chern. Finding the most vital edge with respect to minimum spanning tree in weighted graphs. *Information Processing Letters* **39** (1991) 227–281.

[8] Lih-Hsing Hsu, Peng-Fei Wang, and Chu-Tao Wu. Parallel algorithms for finding the most vital edge with respect to minimum spanning tree. *Parallel Computing* **18** (1992) 1143–1155.

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

[10] Francis Suraweera and Prabir Bhattacharya. An $O(\log m)$ parallel algorithm for the minimum spanning tree problem. *Information Processing Letters* **45** (1993) 159–163.

[11] Francis Suraweera and Piyush Maheshwari. A parallel algorithm for the most vital edge problem on the CRCW-SIMD computational model. In *Proceedings of the 17th Annual Computer Science Conference*, pages 757–766, 1994.

[12] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing* **14** (1985) 862–874.

[13] Robert E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM* **26** (1979) 690–715.

[14] Robert E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters* **14** (1982) 30–33.

[15] Uzi Vishkin. On efficient strong orientation. *Information Processing Letters* **20** (1985) 235–240.