Dartmouth College
Dartmouth Digital Commons

Computer Science Technical Reports | Computer Science

1-1-1991

Optimal Parallel and Sequential Algorithms for the Vertex Updating Problem of a Minimum Spanning Tree

Donald B. Johnson
Dartmouth College

Panagiotis Metaxas
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/cs_tr

Part of the Computer Sciences Commons

Dartmouth Digital Commons Citation

Johnson, Donald B. and Metaxas, Panagiotis, "Optimal Parallel and Sequential Algorithms for the Vertex Updating Problem of a Minimum Spanning Tree" (1991). Computer Science Technical Report PCS-TR91-159. https://digitalcommons.dartmouth.edu/cs_tr/59

This Technical Report is brought to you for free and open access by the Computer Science at Dartmouth Digital Commons. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Optimal Parallel and Sequential Algorithms for the Vertex Updating Problem of a Minimum Spanning Tree

PCS-TR91-159

Donald B. Johnson
Panagiotis Metaxas

Optimal Parallel and Sequential Algorithms for the Vertex Updating Problem of a Minimum Spanning Tree

Donald B. Johnson* Panagiotis Metaxas†
Dartmouth College‡

Technical Report PCS-TR91-159

Abstract

We present a set of rules that can be used to give optimal solutions to the vertex updating problem for a minimum spanning tree: Update a given MST when a new vertex z in introduced, along with weighted edges that connect z with the vertices of the graph. These rules lead to simple parallel algorithms that run in $O(\lg n)$ parallel time using $n/\lg n$ EREW PRAMs. They can also be used to derive simple linear-time sequential algorithms for the same problem. Furthermore, we show how our solution can be used to solve the multiple vertex updating problem.

1 Introduction

COMPLEXITY AND PREVIOUS RESULTS. The vertex updating problem for a minimum spanning tree was first addressed by Spira and Pan in [SP75], where a $O(n)$ sequential algorithm was presented. Another solution using depth-first-search and having the same time complexity was later given by Chin and Houck in [CH78], while Pawagi and Ramakrishnan [PR86] gave a parallel solution to the problem. Their algorithm, which runs in $O(\lg n)$ time [1] using n^2 CREW PRAMs, precomputes all maximum weight edges on paths between any two nodes in the tree, and then breaks

*email address: djohnson@cardigan.dartmouth.edu
†email address: takis@dartmouth.edu
‡Department of Mathematics and Computer Science, Hanover, NH 03755
[1] We denote $\log_2 n$ by $\lg n$.

the $\binom{n}{2}$ cycles simultaneously in constant time. Varman and Doshi [VD86] presented an efficient solution that works in the same parallel time, but uses only n CREW PRAMs. Their solution is a "divide and conquer" algorithm which, using the separator theorem, breaks the input tree in $\sqrt{n}$ subtrees of roughly the same size by removing $\sqrt{n}-1$ edges. Then, it solves the problem recursively in all subtrees and merges the results. Even though their idea is rather simple, the implementation details make the algorithm rather complex. Lately, Jung and Mehlhorn [JM88] gave an optimal solution for the more powerful CRCW PRAM model. However, simulating this algorithm without concurrent writing slows down its performance by a factor of $\lg n$. Their approach reduces the problem to an expression evaluation problem by defining a function on the nodes of the MST which when evaluated effectively computes the new MST.

We present an optimal yet simple solution for the EREW PRAM model which works in $O(\lg n)$ parallel time using $n/\lg n$ processors. Our solution needs a valid tree-contraction schedule. Several methods that provide such schedules have been proposed in the past few years, such as the ones reported in [MR85, ADKP89, KD88, CV88, GR86, GMT88]. All these methods give a schedule for contracting a tree of n nodes into a single node in $O(\lg n)$ parallel time using $n/\lg n$ processors, provided that the *prune* (remove leaves) and the *shortcut* (remove nodes of degree two) operations have been defined.

In section 2 we discuss the background and we give the definition of the problem. Section 3 has an outline of the solution and introduces the invariants and the rules that are used in the problem's binarized version. In section 4 we discuss some of the algorithms that can be derived using the rules, while section 5 describes the binarization technique and gives the main theorem. The related problem of edge-updating is briefly discussed in section 6 and, finally, section 7 shows how the vertex updating algorithms can be used to solve the multiple vertex updating problem in parallel.

2 Definitions

2.1 The parallel Model

The model of parallel computation we will use throughout this paper is the EREW PRAM (exclusive read-exclusive write parallel random access machine) [KR90] in which n processors are employed. These processors communicate through a shared memory. Simultaneous reading from or writing to a memory cell by two or more processors is not allowed. This is the weakest of the PRAM models. Other models include the CREW, in which only simultaneous reading is permitted, and the CRCW,

in which some memory cell can be accessed simultaneously for reading or writing by many processors. These models define a strict hierarchy on the power of the PRAM machines.

2.2 The Problem

We are given a weighted graph $G = (V, E_G)$, along with a minimum spanning tree (MST) $T = (V, E)$ and a new vertex z with n weighted edges connecting z to every vertex in V. (Note: If, in an instance of the problem, z is not connected to some vertex x, we can assume an edge (z, x) having maximum weight.) We want to compute a new MST $T' = (V \cup \{z\}, E')$.

One parallel solution would be to compute the MST from scratch, but this requires time $O(\lg^2 n)$ with $n^2/\lg^2 n$ CREW PRAMs ([CLC82]). Sequential algorithms for computing the MST from scratch on a sparse graph take $O(m \lg \lg n)$ time, where m is the number of edges in the graph ([Yao75, CT76]). The fact that the number of cycles in the input graph is small (there are $O(n^2)$ cycles versus $O(2^n)$ cycles in general graphs) enables us to compute the new MST faster, by breaking the cycles.

Upon introducing the new vertex z along with n weighted edges, $\binom{n}{2}$ cycles are created. If we break these cycles by deleting the maximum weight edge (MWE) that appears in each cycle, the resulting tree will be the new MST. It is easy to see that at most n of these $2n-1$ weighted edges will be included into the new MST. No non-tree edge of the old graph can be included because all of them are already MWEs on some existing cycle in the original graph, so we need not consider any such edge. Moreover, any sequential algorithm that solves the problem must take time $\Omega(n)$: Consider the case when an existing MST forms a path and vertex z is connected to the two ends of the path. Then, any of the $n+1$ edges could be the MWE and a sequential algorithm has to examine all of them.

2.3 Representation

In light of this discussion we may take the input to be a tree T with $n-1$ weighted edges (corresponding to the given MST) and n weighted nodes (corresponding to weights of the newly introduced edges to z). We will call this object a *weighted tree* (figure 1). Note that *a path between two weighted nodes in T corresponds to a cycle in the graph augmented with z*. Such a graph is shown in figure 1b and is implied in figure 1c. Thus we call this object the *implicit graph*.

In the discussion that follows, reference to the weight of a node will mean reference to the corresponding edge in the implicit graph, unless noted otherwise.

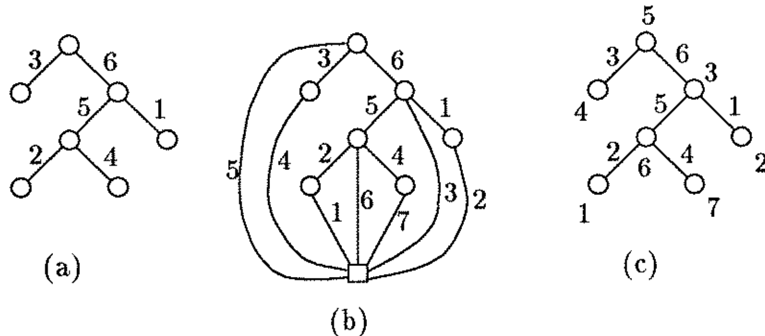

Figure 1: (a) The initial (given) MST, (b) the implicit graph after introducing the new vertex z along with its weighted edges and (c) the corresponding weighted tree.

3 Breaking the Cycles

3.1 Outline of the Algorithm

As we have said, we are given the input in the form of a weighted rooted tree. We assume that each vertex has a pointer to a circular linked list of its children, and the linked lists are stored in an array. This representation of the input is not crucial, since it can be derived in $O(\lg n)$ time using $n/\lg n$ processors from any reasonable representation (see [CV88] for a discussion on the representation).

The algorithm consists of a number of phases. During each phase, nodes of degree 1 (i.e. leaves) and nodes of degree 2 (internal nodes having one child) of the weighted tree are being *processed*. Each tree-node is processed once in the entire course of the algorithm. The order in which the nodes are processed in parallel is dictated by a tree-contraction schedule. Two such schedulings – the Shunting and the ACD – are described later. Processing a node means examining the edges composing *small cycles* (cycles of length 3 or 4) that the node is part of, and breaking these cycles by removing the MWE that appears in them, effectively computing the MST of the subgraph induced by the examined edges. This is done by a set of rules which also update neighboring nodes, so that the size of the unprocessed part of the tree decreases without losing any information about the MWEs of larger cycles.

A sequential algorithm needs only to apply the appropriate rule while visiting the nodes of the tree. Thus a depth-first-search visit of the nodes suffices. When working in parallel though, the rules can apply to many nodes at once, provided that

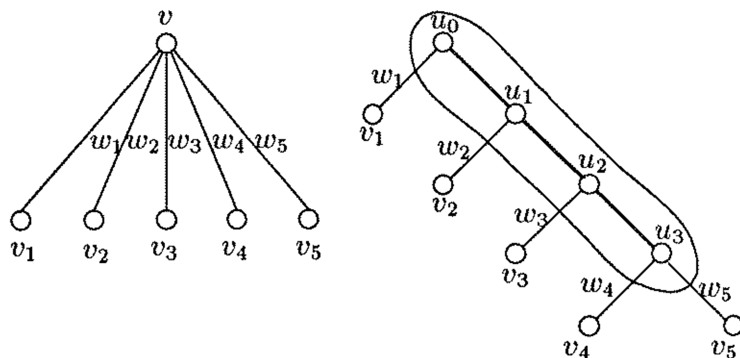

Figure 2: Binarization: A node with more than two children is represented by a right path of unremovable edges.

no confusion arizes from the updating of neighboring nodes. A valid tree contraction schedule suffices to assure that neighboring nodes are not processed at the same time. When all edges have been examined (i.e. after processing all the nodes), the MST of the implicit graph has been computed.

The rules assume a binary tree as input, so some preprocessing is needed to transform the weighted tree to a *binary weighted tree*. This can be achieved in the same time bounds, and is needed only for ordering the processing of a node's children; only the parallel algorithms need this transformation. This transformation is performed by the procedure *binarize*, which we briefly describe here. For the details we refer to section 5. Each node $v = u_0$ with $k > 2$ children is augmented with $k - 2$ fake nodes u_i (see figure 2) to form a right path. Each of the children v_j of node v is attached as a left child of node u_{j-1}, while u_{k-2} has a right child as well. The weights of the edges connecting the u_i's have weights $-\infty$ which makes them unremovable by the MST algorithm. The fake nodes do not have weights. They are introduced only to facilitate the order of processing of v's children. At the end of the algorithm the right path is always included in the MST of the binarized problem, giving a unique obvious solution to the general problem. The binary weighted tree has the same number of cycles, but may have height much larger than the input tree and may contain twice as many nodes. This, though, does not affect the running time of the algorithm, which is logarithmic in the number of tree nodes.

3.2 Invariants and Rules

The rules are divided into two categories: Rules that are applied to nodes of degree 1 (pruning rules), and rules that are applied to nodes of degree 2 (shortcutting rules). For simplicity we will assume that the former is a leaf and the latter is an internal node with one child. This will not always be the case but the treatment is essentially the same.

Each node is examined once for rule application. Then, its incident edges are identified as being either (unconditionally) *included* into the new MST, *excluded* from it, or *conditionally included* in it. A conditionally included edge is one which will be included in the MST unless it is the MWE of another cycle which has not been yet considered. In the figures that accompany this paper, a conditionally included edge is marked with a star (*).

It is useful to observe that *the edge with minimum weight incident to some node will always be included into the MST.* Actually, many sequential and parallel algorithms are based on this observation (i.e. [Pri57, SP75, CLC82]). Edge inclusion makes use of this observation.

Another useful observation is that *whenever some edge is found to correspond to the MWE of some cycle it can be removed from the tree without affecting the computation of the remaining graph.* (Kruskal's MST algorithm makes use of this fact.) Edge exclusion is based on this observation.

Let $w : V \cup E \rightarrow R$ be the weights of the nodes and the edges. As we said earlier, in the beginning of the algorithm the weight of a node v, if it exists, is the weight of the edge connecting v to z. Some nodes (fake or not) may not have an assigned weight. For the purposes of the algorithm a weight of $+\infty$ is assumed on each of them. In this presentation we assume that the weights have distinct values. It is sufficient to consider that the currently processed node has weight larger than its equally weighted neighbor.

Let us define the *provisional graph* to be the graph induced by the examined edges. Note that whenever a leaf v is processed, three edges are examined: $(z, v), (v, p(v))$ and $(p(v), z)$, where $p(v)$ represents the parent of v. Whenever an internal node is processed five edges are examined. Each application of a rule on some node preserves the following three invariants:

Invariant 1 *The minimum spanning tree of the provisional graph has been computed.*

This will be true, because we will consider and break all cycles of the provisional graph by removing the MWE appearing in them. Furthermore, the rules will assure that the non-excluded edges of the provisional graph will constitute a MST. So, it will be the MST of the provisional graph because (i) connectivity is preserved, (ii) no cycles remain and (iii) the cycles have been broken by removing their MWE.

Invariant 2 *The weight $w[v]$, if exists, of some unprocessed node v corresponds to the max weight edge (MWE) on the unique path from z to v either via edge (z, v) or through edges contained in the provisional graph.*

As we said, application of a rule on some node v will examine and break all the small cycles that v is on, by removing their MWE. Some larger cycle sharing the MWE with a broken small cycle will also be broken. The next invariant describes how other larger cycles containing v are affected.

Invariant 3 *Let c be a larger cycle containing node v and let e be its MWE. After the application of a rule on v, then either e has been deleted or there remains another cycle in which e is the MWE.*

These invariants hold in the beginning of the algorithm since there are no processed nodes and thus the provisional graph is empty. We now show how these invariants can be preserved under tree-contraction operations.

3.3 Pruning Rules

Consider a cycle involving leaf v, $p(v)$ and z. Let $w[(v, p(v))] = a$, $w[v] = b$ and $w[p(v)] = c$ (see figure 3). The small cycle of length 3 they form can be broken in such a way that the invariants are preserved by removing one of the three edges involved. We consider the following cases:

$a = \max\{a, b, c\}$: Then, edge $(v, p(v))$ must be excluded from the new MST. When it is removed, the tree is broken into two subtrees (which are connected in the implicit graph via z). Moreover, the edge that corresponds to b has to be included into the MST, since this is the only way that v can be connected with the graph.

$b = \max\{a, b, c\}$: Then, node v can be included in the MST only via $(v, p(v))$. Therefore, this edge is added into the MST while edge (z, v) is excluded from it.

$c = \max\{a, b, c\}$: In this case the best way to include $p(v)$ is *not* via $(z, p(v))$, so this edge is excluded. It could be best to include $p(v)$ via v, but if so, we do not know it yet. But in this case the MWE that connects $p(v)$ with the provisional tree will have weight $\max\{a, b\}$. The edge corresponding to $\min\{a, b\}$ has to be included in the MST. In the figure, this is indicated by labeling v with $\min\{a, b\}$. The edge corresponding to $\max\{a, b\}$ is conditionally included. We update $w[p(v)] := \max\{a, b\}$ to preserve Invariant 2. The effect of the update is the following: If $w[p(v)]$ is found to be a MWE of some other cycle later on, the

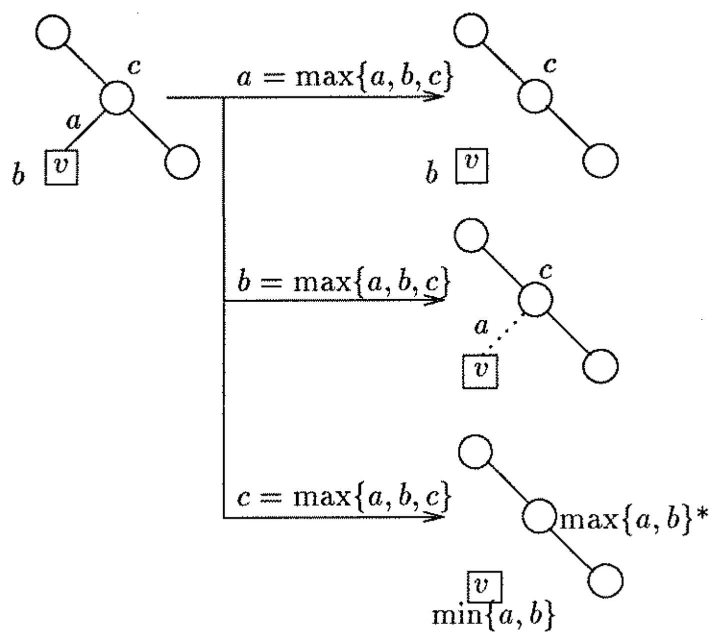

Figure 3: Pruning Rules for a Leaf. In this and in following pictures, a conditionally included edge is marked with a star (*), an (unconditionally) included edge is dotted and we keep its weight letter. We erase an excluded edge along with its weight letter.

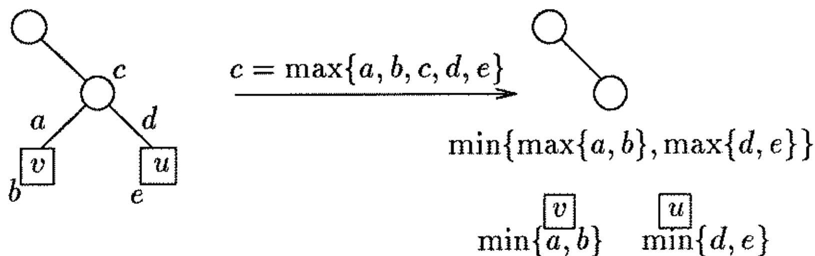

Figure 4: When v's sibling is a leaf.

edge corresponding to $\max\{a, b\}$ will not be included in the MST. Otherwise, if upon termination of the algorithm no rule has excluded $w[p(v)]$, it will be included.

Some special care must be taken in the case that v's sibling, say u, is also a leaf (figure 4) and is processed at the same time. This, for example, can happen when the ACD scheduling method is used, which schedules v and u to be processed at the same time. Let $w[u] = e$ and $w[(u, p(v))] = d$. If $c = \max\{a, b, c, d, e\}$ then $(z, p(v))$ must be excluded and some precaution must be taken to avoid simultaneous reading or writing on $p(v)$ by the two processors. Moreover this cycle of length 4 $(z, v, p(v), u, z)$ must be broken, by excluding one of the four edges. Also $w[p(v)]$ must be updated to either $\max\{a, b\}$ or $\max\{d, e\}$, depending on which child was connected to the excluded edge. In particular, if $\max\{a, b\} > \max\{d, e\}$ then $\max\{a, b\}$ is excluded and $w[p(v)] := \max\{d, e\}$. Else if $\max\{d, e\} > \max\{a, b\}$ then $\max\{d, e\}$ is excluded and $w[p(v)] := \max\{a, b\}$.

3.4 Shortcutting Rules

Let's consider a situation where v has only one child u. There are two possible small cycles involving v: A "lower" one (z, v, u, z) and an "upper" one $(z, v, p(v), z)$. We will describe how to break these cycles in such a way that the invariants are preserved. Let $w[v] = a$, $w[(v, u)] = b$, $w[u] = c$, $w[(v, p(v))] = d$ and $w[p(v)] = e$ (see figure 5).

$a = \max\{a, b, c\}$ or $a = \max\{a, d, e\}$: Then, the best way to include v in the MST is either via $(v, p(v))$ or via (v, u). Thus, the edge with weight $\min\{b, d\}$ must be included to the MST. The other, equal to $\max\{b, d\}$ will be conditionally included, i.e. it will be included if and only if it is not the MWE in another cycle

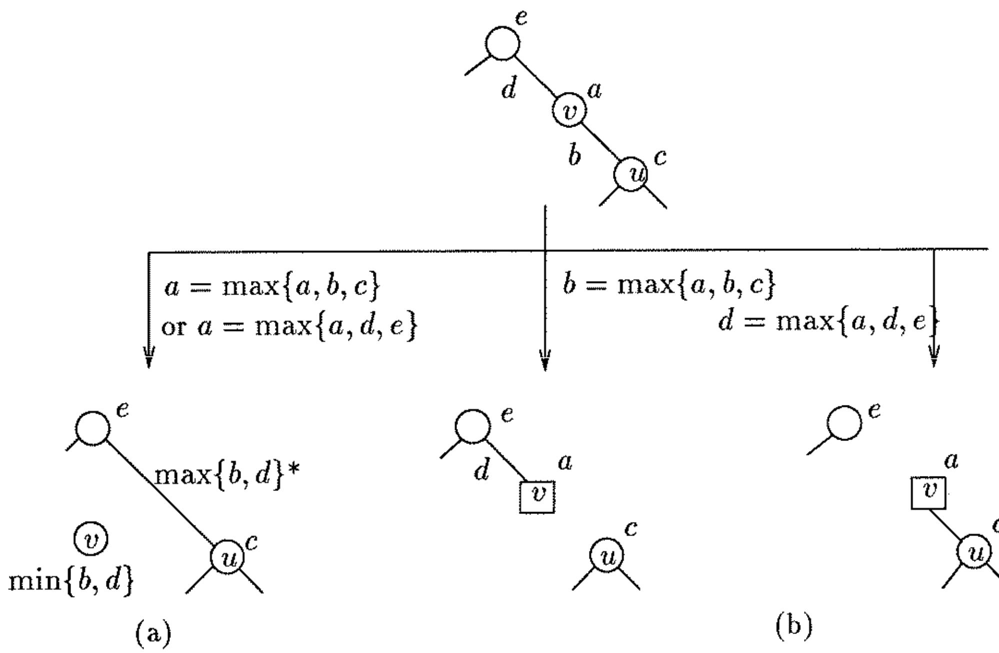

Figure 5: Shortcutting Rules: The first two cases.

involving $z, u, p(v)$, and possibly other nodes. Therefore we can preserve the MWE-related information about these cycles by introducing a new shortcutting edge $(u, p(v))$ corresponding to the edge with weight $\max\{b, d\}$.

$b = \max\{a, b, c\}$ or $d = \max\{a, d, e\}$: In the first case edge (v, u) must be excluded from the new MST and can be deleted. In the second case edge $(v, p(v))$ must be excluded. In either case v becomes a "leaf" and it then can be pruned using the rules for pruning. (Of course in the second case we use the term "leaf" loosely: Node v is the parent of u but nevertheless it can be pruned as a leaf because it has degree one.)

$c = \max\{a, b, c\}$ and $e = \max\{a, d, e\}$: Then, (z, u) and $(z, p(v))$ must be excluded from the MST and the edge corresponding to $\min\{a, b, d\}$ must be included. In figure 6 this case is presented, with the further assumption that $b > d$ for a simplified picture. We have to update $p(v)$ and u so that they reflect this fact preserving the invariants. We observe that there may exist (i) cycles involving edges (z, v), $(v, p(v))$ and edges in the upper part of the tree, (ii) cycles involving (z, v), (v, u) and edges in the lower part of the tree, and finally (iii) cycles going through $u, v, p(v)$ that do not involve (z, v). The update should preserve information about these cycles as dictated by Invariant 3. We update $w[p(v)] := \max\{a, d\}$ to account for cycles of the first type, and $w[u] := \max\{a, b\}$ to account for cycles of the second type, both corresponding to conditionally included edges. Finally, we introduce a conditionally included shortcutting edge $(p(v), u)$ with weight $\max\{b, d\}$ to account for cycles of the third type.

It is worth noting that in the last shortcutting rule not all three weight-updates are needed for the correctness of the algorithm. One of them can be omitted as redundant (figure 6): If $a = \min\{a, b, d\}$, then we don't need the shortcutting edge because some MWE occurring in cycles of the third type will also be MWE in some other cycle of the first or the second type. So, updating $w[p(v)] := d^*$ and $w[u] := b^*$ is enough. Otherwise, if $a \neq \min\{a, b, d\}$, and say $b > d$, then updating of u is not needed, because information about cycles of all three types is preserved (see figure 7). This observation can lead to a simpler implementation of the algorithm. Again, a weightless node is treated as if it was holding the maximum value.

3.5 Memory Access Conflicts

The fact that shortcutting may update both parent and child nodes creates a possibility of a write conflict. Consider the following situation: Let (see figure 8a) nodes x, v, y satisfy $p(y) = v$ and $p(v) = x$, and assume that nodes x and y are to be

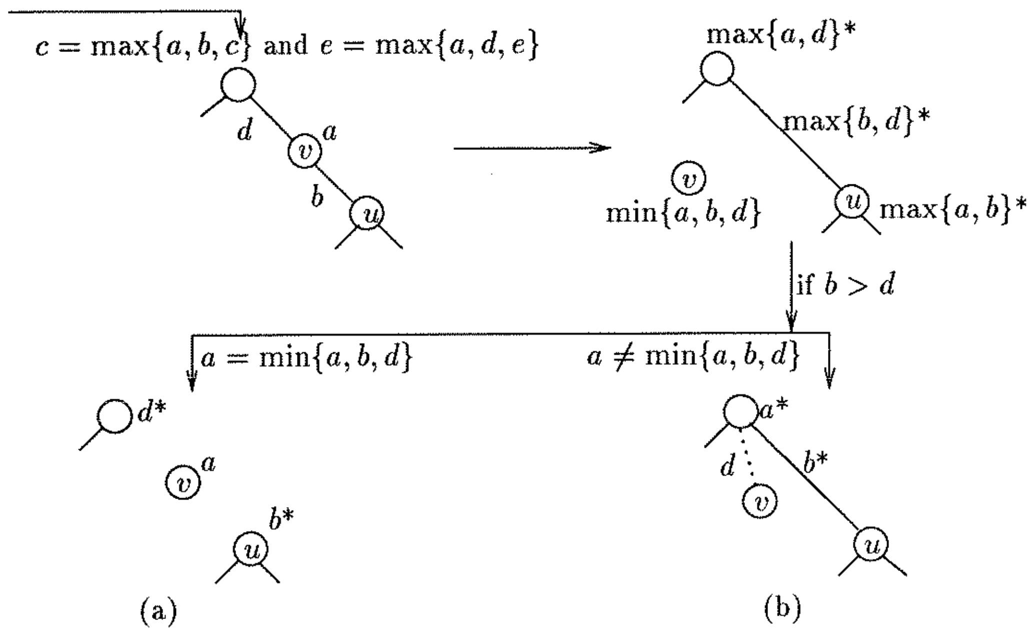

Figure 6: Shortcutting Rules: The third case. In (a) and (b) we show the simplified updating assuming $b > d$.

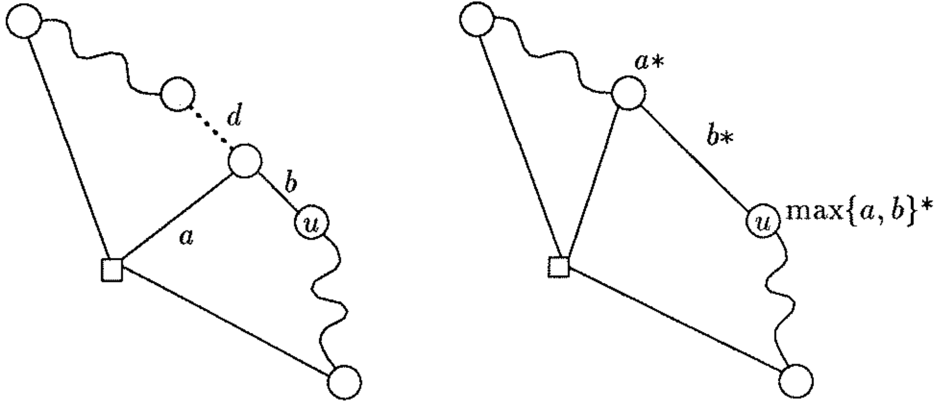

Figure 7: Assuming $d = \min\{a, d, b\}$: By omitting u's updating we have still preserved the MWE's of all three kinds of cycles.

processed simultaneously. Moreover, let's assume that shortcutting node x calls for updating child node v with $w[v] := a$, and shortcutting node y calls for updating parent node v with $w[v] := b$.

Recall that, according to Invariant 2, the weight of a node represents the MWE on the path from this node to z via processed neighbors. The write conflict actually represents a cycle involving nodes z, x, v, y, and possibly x's parent and/or y's child. Since two processors may try to write on the same memory cell, the conflict can be avoided and the cycle behind it should be broken by removing the edge $\max\{a, b\}$ while updating $w[v] := \min\{a, b\}$.

This is the only kind of access conflict that can be created by the shunting schedule we describe in the next section. There are other schedules though, which create a slightly more complicated conflict when updating a node with three values (see figure 8b), say a, b, and c. Again, this can be resolved by breaking the MWEs of the three cycles behind the conflict. Therefore, in this case the algorithm should update $w[v] := \min\{a, b, c\}$ and should exclude the edges that correspond to the other two values.[2]

The above discussion verifies that application of the rules preserve the invariants, proving the following Lemma:

[2]The write conflict that occurs when two sibling leaves are pruned simultaneously (figure 4) can be viewed as a special case of this conflict.

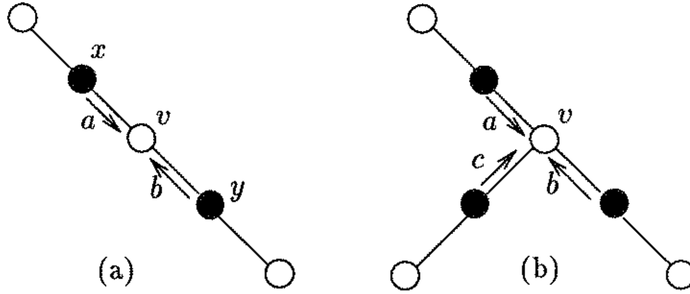

Figure 8: Memory Access Conflicts and how to resolve them.Two (a) and three (b) processors attempt to update v.

Lemma 1 *Application of a pruning or a shortcutting rule on some node v of the weighted graph preserves the invariants.*

□

3.6 Correctness Lemma

We have described a set of rules that define a *prune* operation which removes the leaves of a tree, and a *shortcut* operation which removes nodes of degree two from the tree. Note that individual prunings and shortcuttings take $O(1)$ time to be performed.

A *valid tree-contraction schedule* is one which schedules the nodes of the binary tree for pruning and shortcutting in such a way that (i) when a node is operated upon it has degree one or two, and (ii) neighboring nodes are not operated upon simultaneously.

Lemma 2 *When the rules are applied on the nodes of a binary weighted tree at times given by any valid tree contraction scheduling, they correctly produce the updated minimum spanning tree.*

Proof. As we said, a valid contraction schedule defines processing times on nodes having degree 1 or 2. So, only the prune and shortcut operations are needed, and they are provided by the rules. Moreover neighboring nodes are not scheduled at the same time, and any node may be accessed for updating its weight simultaneously by at most three processors. This conflict can be resolved easily as mentioned in the previous subsection. Therefore the shortcutting and pruning operations can be

done without confusion, and their application, according to Lemma 1, preserves the invariants. Thus, at the end of the schedule the MST of the provisional graph, and therefore the updated MST of the implicit graph, has been computed. □

REMARK 1. Several researchers, who have given solutions to other tree contraction problems, have used a variety of names to denote the "removal of a leaf" and "removal of a node with degree two" operations. *Rake* has been used as a synonym for prune, *compress* and *by-pass* as synonyms for shortcut. Finally, *shunt* and *rake* have been used to denote the application of a prune followed by a shortcut.

4 The algorithms

As we said earlier, the vertex updating problem has a lower bound of $\Omega(n)$ sequential time. We say that a sequential algorithm for some problem of size n is *optimal* if it runs in time that matches a lower bound for the problem to within a constant factor.

Let $t(n)$ denote the parallel running time for some parallel algorithm, and $p(n)$ denote the number of processors employed by the algorithm. Then, $w(n) = t(n) \cdot p(n)$ denotes the work performed by the algorithm. A parallel algorithm for some problem is said to be *optimal* if it has polylogarithmic parallel running time and the work $w(n)$ performed by the algorithm is $O(T(n))$, where $T(n)$ is the running time of the best known sequential algorithm for the same problem. If, instead, $w(n)$ is within a polylogarithmic factor of the optimal speedup the algorithm is called *efficient.*

As we will show, the sequential and parallel algorithms we present here are all optimal. We discuss now the sequential and parallel algorithms that can be derived using the rules.

Theorem 1 *There are optimal sequential and parallel algorithms that solve the MST vertex updating problem based on the rules presented above. The sequential algorithms run in $O(n)$ time and the parallel algorithms run on a binary weighted tree in $O(n/p)$ time using $p \leq n/\lg n$ EREW PRAM processors.*

Proof. A. THE OPTIMAL PARALLEL ALGORITHMS. We first mention a lower bound for any valid tree-contraction schedule resulting from the fact that at most $\lfloor n/2 \rfloor + 1$ nodes can be removed simultaneously:

Lemma 3 *Any valid tree-contraction schedule must have length at least* $\lg n$.

There are, actually, several valid tree contraction schedules that can achieve this lower bound. First, [MR85] proposed such a schedule which was constructed on the fly by an optimal randomized algorithm[3]. The problem and its applications drew

[3]A newer version of their paper was published in [MR89]

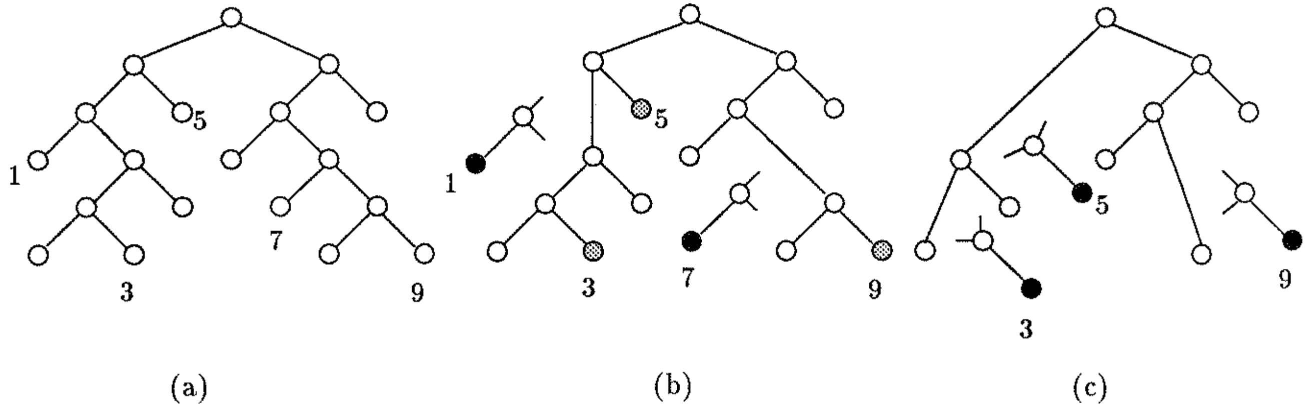

Figure 9: The Shunting Schedule: The first phase. (a) Numbering of the leaves. (b) Step 2: Shunting of odd numbered left children. (c) Step 3: Shunting of odd numbered right children.

the attention of researchers, and soon several optimal deterministic algorithms were presented ([ADKP89, KD88, CV88, GR86, GMT88]). For a discussion on the history of the parallel tree-contraction algorithms see [KR90].

Shunting. We will briefly describe here the simplest of these schedules which was proposed independently by [ADKP89] and [KD88]. The algorithm is composed of a number of phases, each containing the following steps (figure 9):

1. Number the leaves of the tree from left-to-right. Here, the input is supposed to be a *regular* binary tree, i.e. a binary tree in which every internal node has exactly two children. The numbering can be done using the eulerian tour technique ([TV85]) within the desired bounds.

2. Prune the odd-numbered leaves that are the left children of their parent. Then, shortcut their parent. This is called the *shunt* or *rake* operation.

3. Shunt the odd-numbered leaves that are the right children of their parent.

4. Shift out the last bit of the remaining leaves and repeat steps 2 to 4 until the whole tree has been contracted.

Lemma 4 ([ADKP89]) *The previous algorithm computes a valid contraction schedule which has length $O(\lg n)$.*

If we had a processor assigned to each node, we could contract the tree using n processors in the desired time. But the processing time for each leaf can be computed beforehand and placed in an array of length $\lceil n/2 \rceil + 1$. The array is filled with pointers to leaves having numbers 1,3,5,7,..., then to leaves having numbers 2,6,10,14,..., etc. In general there are $O(\lg n)$ phases numbered $0 \leq i \leq \lceil \lg(n-2) \rceil$, and in each of them, leaves numbered $2^i, 3 \cdot 2^i, 5 \cdot 2^i, 7 \cdot 2^i, \ldots$ are shunted. Thus, having the array, it takes time $O(n/p)$ using $p \leq n/\lg n$ processors to do the contraction. Optimality is achieved for $p = n/\lg n$.

ACD. The accelerated centroid decomposition (ACD) technique was proposed by [CV88] and also provides an optimal valid scheduling. Using their technique another optimal algorithm for the vertex updating problem is acquired.

Though the ACD method is rather complex and lengthy to fully describe, we'll give here a brief outline of the scheduling it produces. We first define the centroid decomposition of a tree T. Let *size* of a node v be the number of nodes that are contained in the subtree rooted at v, and let *centroid level* of node v be $\lceil \lg size(v) \rceil$. Then, the *centroid path* of v is the longest directed path which is passing through v and is composed of tree edges and nodes having the same centroid level with v. *Centroid decomposition* of a tree T is the partition of the tree nodes into centroid paths. Note that there are at most $\lceil \lg n \rceil$ centroid levels. A node v is the *tail* of its path, if it has no child with the same centroid level as itself.

According to the ACD schedule some node v is removed as follows: If v is the tail of a centroid path with level i, it is pruned at time $2i + 1 = 2K_v + 1$, else it is shortcuted at time $2K_v + 2$, where K_v is the index of the most significant bit in which the bit representation of A_v and $A_{cp(v)}$ differ. Here, $cp(v)$ is the centroid parent of v (that is, the parent of v if it belongs to the same centroid path), and A_v is the sum of the sizes of the non-centroid children of all centroid ancestors of v in its centroid path.

Since in either scheduling at most three processors may attempt to operate on the same data item at a certain time, we can schedule the operations in such a way that no read or write conflicts occur. This observation along with Lemma 2 completes the proof for the parallel case.

B. THE OPTIMAL SEQUENTIAL ALGORITHMS. The rules we presented do not depend on the particular order in which the nodes of the tree are removed. So, different removal sequences of the nodes yield different algorithms and, in particular, we can derive sequential algorithms from the parallel ones. The running time of these algorithms differ only by a constant. We present here some of these algorithms:

Remove on the fly. Use depth first search to visit the nodes of the tree. Every time a node of degree 1 or 2 is encountered, process it using pruning or shortcutting rule, respectively. Each node will be visited at most twice (on the way down the tree and on the way up the tree), so its running time is $O(n)$.

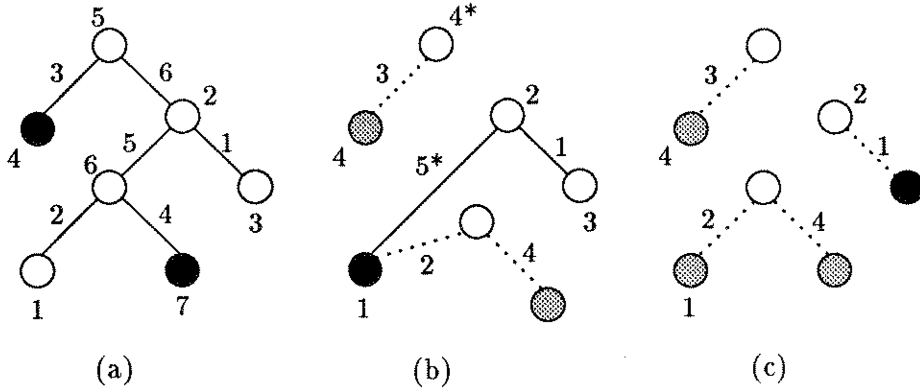

Figure 10: Run of the Parallel Algorithm using SHUNTING on the tree of Figure 1. Dotted edges are included into the MST, deleted edges are excluded from it. (a) In black are the first two processed vertices. (b) Third step. (c) Fourth and final step.

Postorder. Another algorithm simpler to implement but with the same time-bound is the following: Visit the nodes of the weighted tree in postorder (using, for example, depth-first-search). A node is processed after all its children have been processed, so only the pruning rules are needed. Since each node is processed at most once, we have an $O(n)$ sequential algorithm. [4]

Shunting. A third algorithm follows from the parallel scheduling of [ADKP89]. Number the leaves of the tree in a left-to-right manner and place their addresses in an array of length $\lfloor n/2 \rfloor + 1$ as dictated by the schedule. Then visit the array performing shunts on the nodes of the array.

ACD. Visit the leaves of the tree computing their centroid levels and performing centroid decomposition [CV88]. Bucket-sort the scheduled numbers, place them in an array of length n and then process the nodes of the array. □

REMARK 2. This list does not exhaust the possible sequential and parallel algorithms that can be based on the rules presented, but includes only the simpler ones. We should note that other tree-contraction techniques (like the ones presented in [GR86] and [GMT88]) lead to different algorithms with the same bounds. (The shunting scheduling is shown in figure 10).

REMARK 3. The shunting method described in [ADKP89] requires that the root

[4] As a matter of fact, this algorithm is similar to the one given in [CH78], but the pruning rules make it easier to describe and understand.

of the tree not be shunted until the end. This is needed mainly for the expression tree evaluation algorithm. In our case, shunting the root is permitted since there is no top-to-bottom information to be preserved.

REMARK 4. Tree contraction algorithms usually require a second phase after the contraction which is called "expansion phase". The algorithms we present here do not need a second phase because of Invariant 1: At the end of the tree contraction phase the new MST has been computed.

5 Binarization

We now discuss how a general tree can be transformed into a binary tree using the procedure *binarize*:

Each node v having k children $v_1, \cdots, v_k$ is represented by a *right path* (figure 2) composed of $v = u_0$ and $k-2$ fake nodes $u_1, \cdots, u_{k-2}$, so that node u_j is the right child of u_{j-1} and the parent of u_{j+1}. Node v_i, $i = 1, \cdots, k-1$ becomes the left child of node u_{i-1} and v_k the right child of u_{k-2}. We assign weights of $-\infty$ to the edges of the right path so they can not be excluded by the rules. The fake nodes have no assigned weight (which is treated as maximum value weight by the algorithm) because they are only introduced to facilitate the processing order of v's children. Of course, the real nodes v and the v_i's keep their weights.

Recall that the shunting scheduling accepts as input a regular binary tree. When using this technique the binarization should be extended to handle nodes v with only one child in the input tree. For each of them a second child v' is introduced, for which $w[v'] = \infty$ and $w[(v, v')] = -\infty$.

Theorem 2 *There are logarithmic-time optimal parallel algorithms solving the MST vertex updating problem on a rooted tree.*

Proof: The binarized graph has exactly the same number of cycles as the given graph, and at the end of the algorithm the edges composing the right path are always included into the new MST. Therefore, the solution of the binarized problem shows a corresponding unique and unambiguous solution to the general problem. □

Similar binarization techniques to the one described have been used in [VD86, CV88, ADKP89]. Another technique ([JM88]) "plants" a balanced binary tree over the v_i's with v as the root. The internal nodes and the internal edges have weights as those in the right path in the previously described technique. Both constructions require the *list ranking* algorithm [CV86] which runs within the desired bounds. (Actually, in [VD86] the eulerian tour technique is used which has the list ranking procedure as a subroutine.)

6 A Note on Edge Updating Problem

The edge updating problem of a MST can be defined as follows: We are given a graph $G = (V, E)$ and its MST $T = (V, E')$ as input along with the weight function $w : E \to R$. Also we know that one of the edges e changes weight, and we want to compute the new MST T'. A sequential algorithm for this problem is given in [Fre83] with $O(\sqrt{m})$ running time.

It is obvious that if e is a tree edge and decreases its value or if e is not a tree edge and increases its value, then $T' = T$. In the other two cases we have:

A TREE EDGE INCREASES. A simple solution is to consider the two connected components that e divides the graph into, and to find the minimum weight edge connecting them. So, a simple algorithm for it, is:

1. Preorder the tree using the eulerian tour technique [TV85].

2. Using the preorder numbering divide V in two subsets V_1 and V_2 created by removing e from T.

3. Put all edges connecting some vertex in V_1 to some vertex in V_2 in an array of length $m = |E|$ and compute the minimum.

The running time of this algorithm is $O(\lg n)$ using $m/\lg n$ EREW PRAMs.

A NONTREE EDGE DECREASES. Consider the cycle it creates in the tree when is added, and remove the MWE in it using a variation of the list ranking algorithm in which the rank of a list element l is defined to be the maximum of the values appearing in the elements following l in the list. The running time is $O(\lg n)$ using $n/\lg n$ EREW PRAMs [CV86].

This problem is a special case of the *edge insertion in a MST* problem for which [CH78] have shown a simple reduction to the vertex insertion problem. Our algorithm can be used to solve this problem in parallel as well.

7 On the Multiple Vertex Updates Problem

7.1 Introduction

We define the problem of *multiple vertex updates* of a MST as follows: Let $G = (V_G, E_G)$ be a weighted graph on n vertices and m weighted edges and $T = (V_G, E_T)$ be its MST. Suppose G is augmented with $k = |V_k|$ new vertices that are connected to V_G by $kn = |E_k|$ new weighted edges, but they are not connected among themselves. We are asked to compute the new MST T'.

We will prove the following Theorem:

Theorem 3 *The multiple updates MST problem can be solved in parallel in time* $O(\lg n \lg k)$ *using* $nk/\lg n \lg k$ *EREW PRAM processors.*

The problem of multiple vertex updates was considered in [Paw89] where a parallel algorithm is presented. It runs in $O(\lg n \lg k)$ time using nk CREW PRAM processors. We will show how our solution for the (single) vertex update problem can be used to achieve a better solution for the multiple updates problem.

7.2 Optimality

A sequential algorithm can solve the problem in time $O(kn)$, by solving k single update problems sequentially. Another approach would be to compute the MST of the augmented graph $G' = (V_G \cup V_k, E_T \cup E_k)$ from scratch. Again, the edges that are not in the given MST do not need to be considered.

The augmented graph G' can be sparse or dense depending on the value of k. The best algorithms to compute the MST of a graph $G = (V, E)$ sequentially run in time $O(|E| \lg\lg |V|)$ for sparse graphs ([Yao75, CT76]) and in time $O(n^2)$ for dense graphs (using Prim's well known algorithm [Tar83]). Assuming that the number of edges connecting the new vertices to the tree is $O(kn)$, the k single updates solution is preferable since it is simpler and, for sparse graphs, asymptotically faster.

The solution we present solves the problem in $O(\lg n \lg k)$ time using $nk/\lg n \lg k$ EREW PRAM processors. It is therefore optimal for graphs having $O(kn)$ edges and also uses a weaker model of parallel computation.

7.3 The algorithm

Our solution follows in general the solution presented in [Paw89] but in certain parts uses different implementation techniques to achieve the tighter time and processor bounds.

The algorithm consists essentially of three parts.

1. Make k copies of T and solve k update MST problems in parallel.

2. Combine the MSTs of the k solutions into a new graph G_z. This graph may contain cycles. Transform it to an equivalent bipartite graph G_b.

3. Solve the bipartite MST problem on the graph G_b.

We will show that each of these parts can be implemented within the desired time-processor bounds.

7.3.1 Solving k Updating Problems

Making k copies of T can be done efficiently as an application of the *scheduling principle* [KR90], which was pointed out by Brent [Bre74]:

Lemma 5 *Suppose that an algorithm is composed of m computational operations executed in t parallel steps, and let $m_i, 1 \leq i \leq t$ be the operations executed at step i. Then it can be implemented in $O(\frac{m}{p} + t)$ parallel steps using p processors.*

This scheduling principle assumes that *processor allocation* is not a problem. This is true when (a) the m_i's can be computed and (b) we can group the m_i, $1 \leq i \leq t$ operations in groups of cardinality p. Making k copies of T requires $O(kn)$ operations and it can be done in constant time if kn processors are available. Therefore, it can be done in $O(\lg n \lg k)$ time using $kn/(\lg n \lg k)$ processors.

According to Theorem 1, a single updating of a MST can be done in $O(n/p)$ time when p processors are available. Here we have k problems to solve, each of size n. Allocating $n/\lg n \lg k$ processors per problem, it takes $O(\lg n \lg k)$ time to solve each problem in parallel.

7.3.2 Creating the Bipartite Graph

Next, we have to combine the k solutions found in the first part, into a new graph G_z which in turn is transformed to an equivalent bipartite graph G_b. By *equivalent* here, we mean that *there is a cycle in G_b if and only if there is a cycle in G_z*. Graph G_z will never be explicitly created. It is only defined for the sake of description.

Consider the MST of some solution $T_i = (V \cup \{z_i\}, E_i)$. Then, E_i consists of edges (v, w) that belonged to the old MST T, along with new edges of the form (z_i, v). An important observation is that *if some edge (v, w) of the old MST does not appear in all k solutions, it will not be included into the final MST*. This is so, because edge (v, w) was the MWE of some cycle in one of the subproblems and thus must be excluded. So, $G_z = (V \cup \{z_1, \cdots, z_k\}, E_z)$ is composed of original edges (v, w) that appear in *all* k solutions, along with edges of the form (z_i, v), $\forall i \in \{1, \cdots, k\}$. It is easy to see that the formation of G_z can be done within the desired bounds, because there are at most $n - 1$ such edges per solution to examine.

We can view G_z as a collection of subtrees C_j of the old MST that are held together by the z_i's (see figure 11). Every cycle in G_z can be viewed as starting at some z_i, then entering subtree C_j at a node v_e and visiting some of its nodes, then exiting through a node v'_e and visiting z_l, etc, until returning back to z_i (figure 12). The nodes v_e that are adjacent to some z_i are called *e-nodes*.

The transformed graph $G_b = (V_b, \{z_1, \cdots z_k\}, E_b)$ has a set of vertices V_b which contains one vertex v for each e-node v_e of G_z. Consider a path from z_i to some

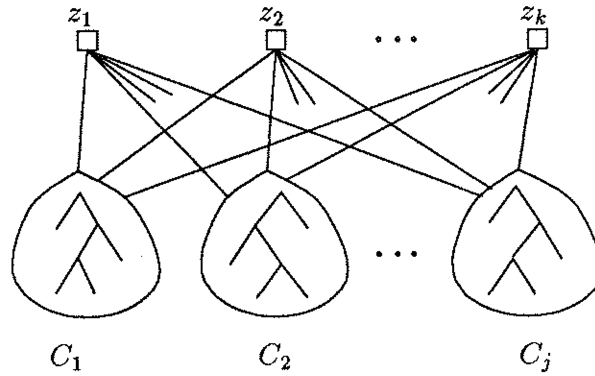

Figure 11: The graph G_z results from putting together the k solutions. The figure points out G_z's bipartite nature.

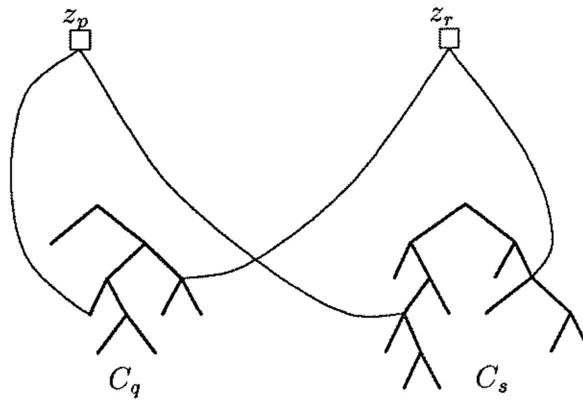

Figure 12: Cycles in G_z. They are composed of alternative visits to z_i's and to tree components. Vertices that are connected to z_i's are called *e-vertices.* In this picture a cycle of length 4 is shown.

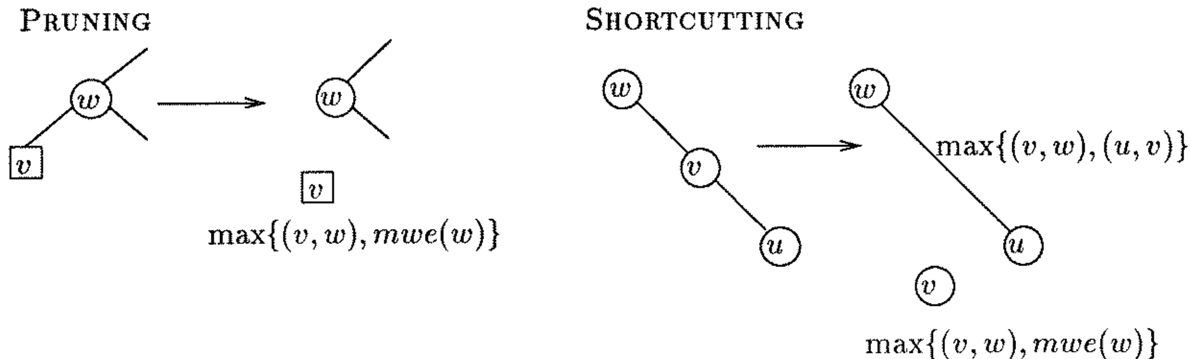

Figure 13: Prunning and Shortcutting Rules for the CADR problem.

e-node v_e and let x be the MWE on this path. Then edge $(z_i, v) \in E_b$ corresponds to this path and has cost equal to x's cost. The algorithm needs therefore to compute all MWE on all paths from z_i to e-nodes v_e. This is done as follows:

First, all e-nodes v_e are recognized and then we root each of the T_i's at z_i. Both of these operations can be done within the desired bounds. Now we have to compute the MWEs on the paths between the z_i's and the e-nodes. So, we have to solve k instances of the following problem:

ALL DISTANCES TO ROOT (ADR). Given a (regular binary rooted) tree with n nodes having weights associated with its edges, find the MWE for each path from a node to the root in $O(n/p)$ time using $p \leq n/\lg n$ EREW PRAM processors.

A sequential algorithm for the problem uses depth first search and runs in linear time. The parallel algorithm reduces the problem to tree-contraction as follows: Associate a function $mwe(v)$ for each node v, where $mwe(v) = \max\{(v, p(v)), mwe(p(v))\}$ if v is not the root and $mwe(root) = \emptyset$. Use tree contraction to contract the tree and then tree expansion[5] to compute $mwe(v)$ for all v. We can come up with simple rules which are given in figure 13.

THE MULTIPLE ADR PROBLEM. Given k (regular binary rooted) weighted trees each one with n nodes, compute the ADR problem on each of them in time $O(\lg n \lg k)$ using $nk/\lg n \lg k$ processors.

The solution is a simple application of the previous algorithm. We associate $n/\lg n \lg k$ processors per tree and compute the problem in time $O(\lg n \lg k)$.

[5]The expansion phase is needed because the definition of the function is top-down.

7.3.3 The Bipartite MST problem

For the third part of the algorithm we need the following definition of the bipartite-MST problem: Let $G = (V_k, V_n, E)$ be a weighted bipartite graph, where $|V_k| = k$ and $|V_n| = n$, $k \leq n$. We want to compute its MST.

Lemma 6 *The bipartite-MST problem can be solved in $O(\lg n \lg k)$ parallel time using $kn/(\lg n \lg k)$ processors.*

Proof. The algorithm that we use is a well-known algorithm whose main idea is attributed to Borůvka [Tar83] and was described in its parallel form in [CLC82]. The analysis though and the time-processors bounds for the bipartite-MST problem are new.

First, let us give some definitions. A *pseudotree* is a digraph in which each node has outdegree one. A pseudotree has at most one (simple) cycle. A *pseudoforest* is a graph whose components are pseudotrees. The algorithm consists of a number of stages. In each stage, each vertex v selects the minimum weight edge (v, w) incident to it. This creates a pseudoforest of vertices connected via the selected edges. In this case the cycle of each pseudotree involves only two vertices, so it can be easily transformed into a tree. Next, each tree is contracted to a star using pointer-doubling, and vertices in the same component are identified with the root of the star.

A crucial observation here is that after the first stage there will be no more than k vertices in the resulting graph and the problem can be solved in $O(\lg^2 k)$ time using $k^2/\lg^2 k$ processors. So, we only have to show that the first stage can be performed within the desired bounds.

As we said, the first stage consists of finding the minima of $O(n)$ sets of vertices, each with cardinality $O(k)$ and then to reduce the $O(k)$ resulting pseudotrees of height $O(n)$ to stars. For the first part Brent's technique applies. For the second part we use the optimal list-ranking technique of [CV86].

8 Conclusion and Open Questions

It is interesting to find other tree problems for which the tree contraction technique applies. Moreover, an interesting question is the following: Can we come up with a general way of defining pruning and shortcutting rules for tree-contraction in parallel, for those tree problems that are solved sequentially using depth-first-search? Also, developing simple and fast SCAN algorithms for tree contraction would lead to implementation of several PRAM algorithms on machines which are described by the SCAN model.

Acknowledgement. The authors would like to thank Professors Sam Bent and Jim Driscoll for their helpful comments in the presentation of the paper.

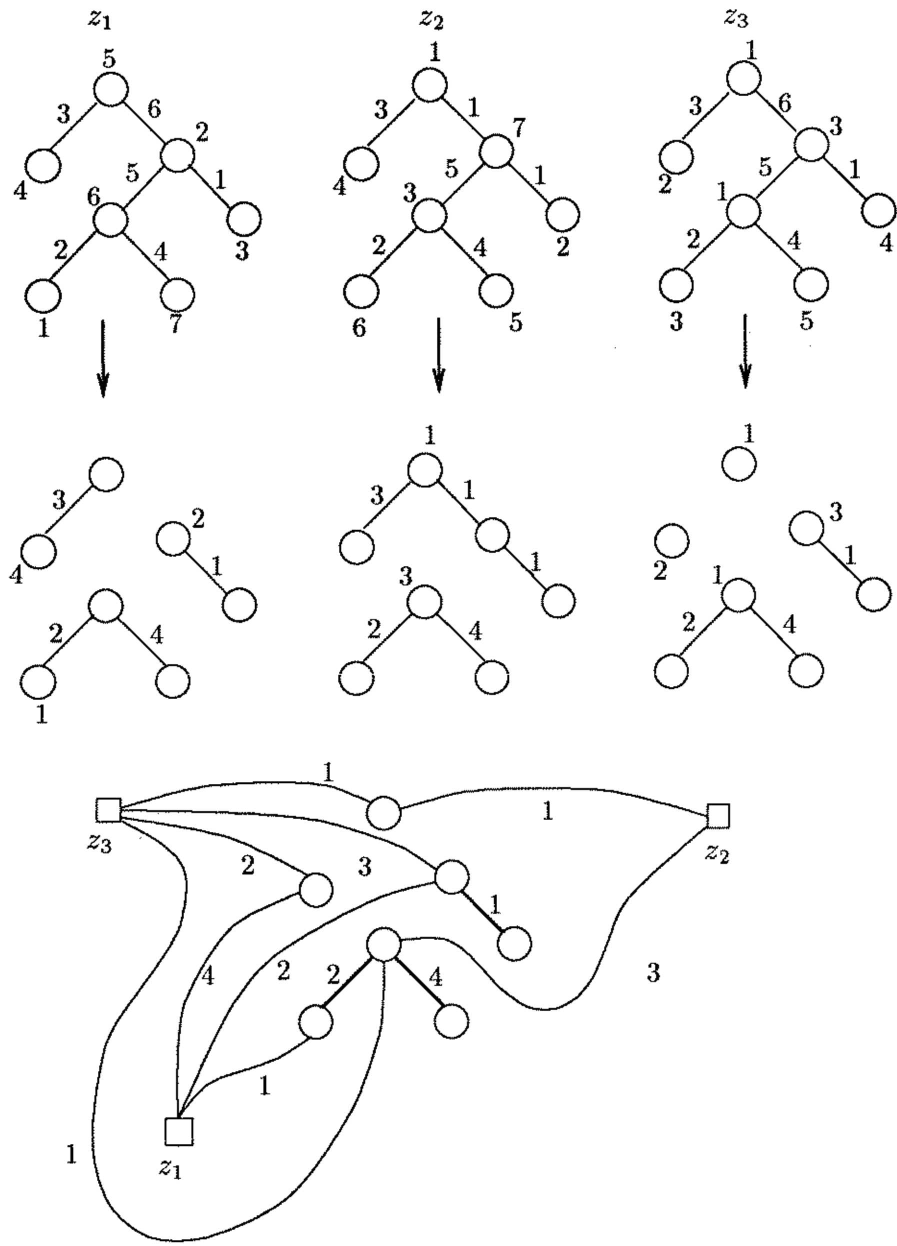

Figure 14: Updating of a MST with 3 new vertices. In the lower part of the figure the graph G_z is shown.

References

[ADKP89] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[Bre74] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal Assoc. Comput. Mach.*, 21(2):201–206, 1974.

[CH78] F. Chin and D. Houck. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences*, 16(12):333–344, 1978.

[CLC82] F.Y. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 25(9):659–665, September 1982.

[CT76] D. Cheriton and R.E. Tarjan. Finding minimum spanning trees. *SIAM Journal Computing*, 5:724–742, 1976.

[CV86] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[CV88] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[Fre83] G. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proc. of the 15th Annual ACM Symp. on Theory of Comput.*, pages 252–257, 1983.

[GMT88] H. Gazit, G. L. Miller, and S. H. Teng. Optimal tree contraction in the EREW model. In *Concurrent Computations: Algorithms, Architecture, and Technology*, New York, 1988. Tewksbury and Dickinson and Schwartz (editors), Plenum Press.

[GR86] A. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. In *Symposium on Foundations of Software Technology and Theoretical Computer Science*, volume 6, pages 453–469. Springer Verlag, 1986.

[JM88] H. Jung and K. Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Information Processing Letters*, 27(5):227–236, April, 28 1988.

[KD88] S. Rao Kosaraju and Arthur L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. *Aegean Workshop on Computing*, pages 101–110, 1988.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook for Theoretical Computer Science*, 1:869–941, 1990.

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pages 478–488, 1985.

[MR89] G. L. Miller and J. H. Reif. Parallel tree contraction. Part 1: Fundamentals. *Advances in Computing Research*, 5:47–72, 1989. JAI Press.

[Paw89] S. Pawagi. A parallel algorithm for multiple updates of minimum spanning trees. In *International Conference on Parallel Processing*, volume III, pages 9–15, 1989.

[PR86] S. Pawagi and I.V. Ramakrishnan. An $O(\log n)$ algorithm for parallel update of minimum spanning trees. *Information Processing Letters*, 22(5):223–229, April, 28 1986.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *Tech. Journal, Bell Labs*, 36:1389–1401, 1957.

[SP75] P.M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4(3):375–380, September 1975.

[Tar83] R.E. Tarjan. *Data Structures and Network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania 19103, 1983.

[TV85] R.E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal Comput.*, 14:862–874, 1985.

[VD86] P. Varman and K. Doshi. A parallel vertex insertion algorithm for minimum spanning trees. In *13th ICALP, Lecture Notes*, volume 226, pages 424–433, 1986.

[Yao75] A. Yao. An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4:21–23, 1975.