An $O(\log n)$ Time Common CRCW PRAM Algorithm for Minimum Spanning Tree

Ramesh Subramonian
Division of Computer Science,
University of California, Berkeley,
Berkeley CA 94720
subramon@melody.berkeley.edu

Abstract

We present a probabilistic algorithm for finding the minimum spanning tree of a graph with n vertices and m edges on a Common CRCW PRAM. It uses expected $O(\log n \log^* n)$ time with $(m+n)$ processors and expected $O(\log n)$ time with $(m+n)\log n$ processors. This represents a significant improvement in terms of efficiency over the previous best results for solving this problem on a Common CRCW PRAM and compares favourably with the best result for the Priority CRCW PRAM, a more powerful model. The algorithm presents a novel application of recent results on recursive *-tree data structures [2]. An important contribution of this paper is (i) a strategy to schedule the growth of components in algorithms based on repeated graph-contractions and (ii) an amortized analysis technique to account for the scheduling overhead.

1 The Algorithm

The Minimum Spanning Tree (MST) problem is a classic combinatorial optimization problem and has been studied extensively in the literature [7]. Many almost linear sequential algorithms have been designed for this problem. Various parallel algorithms for the MST problem have been proposed. We give a brief explanation of the models used, details of which can be found in [10]. In the Concurrent Read Exclusive Write (CREW) model, in a step, more than one processor can read from a given location but at most one processor can write to a given location. In the Concurrent Read Concurrent Write (CRCW) Model, in a step, more than one processor can write to a given location in a step. Based on the way write conflicts are resolved, there are various possibilities. In the Common CRCW Model, concurrent writes to the same location must write the same value. In the Arbitrary model, an arbitrary one of the multiple writes to the same location succeeds. In the Priority Model, processors are assigned unique, unchangeable priorities at the start of the computation and the write of the highest priority processor succeeds. Chin *et al* [3] proposed an $O(\log^2 n)$ time algorithm on a CREW PRAM using n^2 processors. Hirschberg [8] proposed an $O(\log n)$ time algorithm on a Common CRCW PRAM using n^3 processors. Awerbuch and Shiloach [1] proposed an $O(\log n)$ time algorithm on a Priority CRCW PRAM using $m + n$ processors. It has been shown that a single step of a Priority CRCW PRAM can be simulated on a Common CRCW PRAM using $O(\frac{\log p}{\log\log p})$ time, where p is the number of processors that each has [6]. This can be used to modify the algorithm of [1] to run on a Common CRCW PRAM in $O(\frac{\log^2 n}{\log\log n})$ time using $m + n$ processors.

In this paper, we devise a Common CRCW PRAM algorithm that requires $EO(\log n)$ [1] time using $(m+n)\log n$ processors and $EO(\log n \log^* n)$ time using $m + n$ processors. This is the first (almost) logarithmic time algorithm for this problem which uses a linear number of processors on the weaker and more realistic Common CRCW PRAM model.

Outline. In Sollin's sequential algorithm [7], which finds the MST in $(m+n)\log n$ time, in each iteration, the smallest edge incident on each node is identified as an MST edge. These edges are used to contract the graph i.e, all nodes connected by MST edges are merged into a single component and all edges between nodes in the same component are made redundant. An edge (u, v) is redundant if $u = v$. This is repeated this until all edges are redundant, which takes $O(\log n)$ iterations. In the sequential algorithm, finding the minimum edge incident on a component is found by merging the heaps of the constituent nodes. In the parallel version, finding the minimum edge is easy initially because each edge list is sorted. To continue to be able to find the smallest edge quickly, the edges belonging to the constituent nodes that merge into a super-node must be in sorted order .

In designing the parallel equivalent, it is clear that we should not form large components because assembling, compacting and sorting the edges of the constituent nodes could be time-consuming. Instead we try to create small components which can be shrunk fast and yet create

[1] $EO(f(n))$ is shorthand for *expected* $O(f(n))$.

many of these components so that the graph as a whole shrinks fast. There is still the possibility that a large component could be created. In that case, we shrink it over many iterations.

Data Structures. Let $G = (V, E)$ be an undirected weighted graph, where V is a set of vertices and E is a set of edges, each $e \in E$ having a weight $w(e)$. Without loss of generality, assume that G is connected and that the weights are distinct.

We assume that G is presented as an adjacency list with the edges incident on a particular node in contiguous locations. Each edge appears twice, once as (i, j) and once as (j, i). $\forall v \in V$: the component v belongs to is stored in $P[v]$. Initially, each node is in a component by itself i.e, $\forall v \in V, P[v] \leftarrow v$. When we merge i into j, we set i to point to j ($P[i] \leftarrow j$). Two nodes belong to the same component if they point to the same node. A leader points to itself ($P[v] = v$). We will maintain the invariant that each node points to a *leader* i.e., $\forall v : P[P[v]] = v$. (We will use the terms *node* and *component* interchangeably. Strictly, a component, C_v, is a set of nodes which point to the same leader, v ($C_v = \{i | P[i] = v\}$).) In each iteration, the minimum edge incident on a node, i, is recorded in $Min\_edge[i]$. $MST[1..n]$ will eventually contain the $n - 1$ edges of the minimum spanning tree. For each component i, we have a list C_i which is a list of the vertices in it. Initially, $\forall i, C_i = \{i\}$. For each component i, we have a list L_i which consists of a subset of the edges from each of the vertices that comprise component i. Initially, $\forall i, L_i =$ smallest edge incident on i. We defer details of C_i and L_i to Section 1.2.

1.1 Details

For each *leader*, i, we repeat the following steps until all edges are redundant.

1. Find smallest edge incident on i. $Min\_edge[i] =$ left most non-redundant edge in L_i. (We justify restricting attention to the edges in L_i in Lemma 2.4.) This is done in $O(1)$ time using $|L_i|$ processors using the algorithm cited in Lemma 2.1.

2. $Colour[i] \leftarrow\in_R \{Black, White\}$ [2].

3. If i is $White$ and its minimum edge is to a $Black$ node ($Min\_edge[i] = j$ and $Colour[j] = Black$), then indicate that i wants to merge with j. This is done by: $\forall (j, i)$: if $Colour[i] = White$ and $Colour[j] = Black$ and $Min\_edge[i] = j$, then $Marked[(j, i)] \leftarrow true$.

4. Each black node determines whether the number of its edges that are marked is 0, 1, or greater. This is done with a minor variation of the algorithm cited in Lemma 2.1.

5. Every white node, i, that desires to merge does so if it is the only such node (i merges with j if $Colour[i] = White$, $Min\_edge[i] = j$, $Colour[j] = Black$ and $|\{(j, k) | Marked[(j, k)] = true\}| = 1$).

   Merging involves

[2] $x \leftarrow\in_R X$ means x is assigned a value from X uniformly at random.

(a) $P[i] \leftarrow j$,

(b) $MST[i] \leftarrow Min\_edge[i]$. (Note that $MST[i]$ is assigned a value at most once since after Step 7, there will be no edge with i as an end-point.)

(c) combining edge lists of both nodes by $L_j \leftarrow L_j \cup L_i$,

(d) combining constituent nodes of both nodes by $C_j \leftarrow C_j \cup C_i$.

A node is said to be *merged* when all the above four steps have been performed for it. Periodically, C_j and L_j are compacted. (Details in Section 1.2 and 1.3.)

6. **Special case.** This arises when more than one white node wants to merge with a black node. If $\exists b, \exists W : b$ is black and $|W| > 1$ and $\forall w \in W$, w is white and $Min\_edge[w] = b (\equiv Marked[(b,w)] = true)$, then we have found a large component. We shrunk it only partially in this iteration by pairing up nodes in W and merging these pairs. We will mark all nodes in this component, $(W \cup \{b\})$, as passive until the contraction is complete, at which stage b is reset to active. A passive component executes only Steps 6(b)-6(d).

   Rationale for marking nodes passive. When a component is formed, the minimum edge incident on it is the minimum of the edges incident on each constituent node. Since the component is too large for us to assemble all the edges of the constituent nodes in $O(1)$ time, we mark them as passive until such time as the contraction is complete.

   Remarks. First, as a consequence of Step 6(b) and Step 7, no edge will have a white, passive node as an endpoint. Second, while being passive prevents b from merging into another node, j, it does not prevent j from merging into b.

   Define $pred[w] = v$ if $Min\_edge[w] = Min\_edge[v] = b$ and (b,v) is the rightmost edge to the left of (b,w) in b's edge list. $\forall w \in W$, compute $pred[w]$. For the tail of the list (the white node, w_0, corresponding to the left-most marked edge in b's edge list), we set $pred[w_0] = b$. Thus, we create a linked list of the nodes in W. We pair up the nodes in W and merge these pairs.

   (a) $\forall w \in W$, set w to *passive*. Set b to *passive*. b will be reset to active only when it has no marked edge (no node want to merge with it).

   (b) $\forall w \in W$, compute $pred[w]$.

   (c) $\forall w \in W$, $Sex[w] \leftarrow \in_R \{Male, Female\}$. By definition, $Sex[b] = Male$.

   (d) If $Sex[w] = Female$ and $Sex[pred[w]] = Male$, then merge w into $pred[w]$

7. To ensure that all nodes point to a leader and that all edges are between leaders, we do "short-cutting". $\forall u \in V$, $P[u] \leftarrow P[P[u]]$ and $\forall (u,v) \in E$, $(u,v) \leftarrow (P[u], P[v])$.

1.2 Partial Compaction and Sorting (PCS)

Intuition. When we merge components, we must ensure that the edges of this component are in contiguous memory locations and still sorted. Also, we would like to discard redundant and duplicate edges. However, to do so after every iteration would be too time-consuming. So, our strategy (PCS) is to construct an edge list with just sufficient information from each of the constituent nodes as is necessary. Now, to find the minimum edge incident on the component, we can restrict our attention to the selected edges. This allows us to treat the component as if it were a single node until the next PCS.

Consider a node which has just become part of a component of size s. Till the time it becomes a part of a component of size s^2, it could lose at most $s^2 - 1$ edges, one to each member of its component. Therefore, at most the smallest s^2 of its edges were relevant to this period of the computation.

Details The question is how often and how much of PCS to perform ? Our rationale is to perform as much PCS as possible, without affecting the speed of the computation (Lemma 1.2). We will show that this is precisely the amount of PCS we need to perform in order to find the minimum edge incident on a node in $O(1)$ time (Lemma 2.4).

Let I be the number of iterations performed. We perform PCS when I doubles. Consider the PCS when $I = k$. The number of iterations since the last PCS is $\frac{k}{2}$ Therefore, as long as PCS takes $O(k)$ time, it does not increase the asymptotic time. For each component, we create an edge list comprising 2^{2k} edges from each constituent node, eliminate redundant edges using parallel prefix and sort the remaining edges.

Lemma 1.1 *After k iterations, the size of the largest component is $\leq 2^k$.*

Proof. Since at most two nodes merge into one in an iteration, this follows directly. □

Lemma 1.2 *PCS requires $O(1)$ amortized time per step.*

Proof. Consider the PCS performed when the number of iterations is k. The size of the edge list to be compacted is $\leq 2^k \times 2^{2k} = 2^{3k}$ edges since, by Lemma 1.1, the maximum size of a component after k steps is 2^k. Compacting and eliminating redundant edges is done using parallel prefix which requires $\log L \leq O(k)$ time for a list of size $L = 2^{3k}$ using L processors [11]. Sorting requires $\log L \leq O(k)$ time for a list of size $L = 2^{3k}$ using L processors [4]. Since the number of iterations since the previous PCS is $k/2$, the amortized cost of PCS is $O(1)$ per step. □

1.3 Complete Compaction and Sorting (CCS)

However, PCS is insufficient as we will now show. After a PCS has been performed, we can find the smallest edge incident on each component in $O(1)$ time. When we merge components, we do not compact and re-sort the edges of the constituent nodes until the next PCS. So, to find the smallest edge incident on the composite component, we find the smallest edge incident on each of the constituent components and then find the minimum of these minima. We can do so in $O(1)$ time (Lemma 2.2) provided that the number of processors available is greater than the square of the number of constituent components. If this is not true, we perform CCS, in which we merge *all* the edges of the constituent components, eliminate redundant edges and re-sort them. Thereafter, the entire component can be treated as a single sorted node.

Lemma 1.3 *CCS requires $O(1)$ amortized time per step.*

Proof. Assume PCS has just been performed. Before the next PCS is performed, consider the formation of a component of k sub-components, with a total of E edges. We can use the algorithm cited in Lemma 2.2 if $E \geq k^2$. When k^2 exceeds E, we perform CCS. This requires $\log E$ steps, since it requires a parallel prefix and sorting. But $\log E < 2\log k$ since $E < k^2$. We know by Lemma 1.1 that at least $\log k$ steps must have elapsed since the last PCS for the component to have acquired k sub-components. Hence, the amortized cost of CCS is $O(1)$ per step. □

2 Analysis

Lemma 2.1 *[6] Given an array of n elements which are either 0 or 1, the left most 1 can be found in O(1) time using n processors on a Common CRCW PRAM.*

Lemma 2.2 *[13] Given an array of n elements and n^2 processors, the minimum element can be found in O(1) time on a Common CRCW PRAM.*

Lemma 2.3 *[2] Given an array of size n, the elements of which are 0 or 1, the time for each element of the array to find the rightmost 1 to its left is $O(\log^* n)$ using n processors or O(1) time using $n \log n$ processors.*

Lemma 2.4 *The smallest edge incident on c_i is always the left-most non-redundant edge in L_i.*

Proof. The invariant holds true initially, because we start by sorting the edge list of each node and L_i is a singleton set containing the smallest edge. We show that if it is true after $I = k$ iterations and PCS has just been performed, then it must be true after $2k$ iterations, when the next PCS will be performed.

For the PCS when $I = k$, L_i contains the smallest 2^{2k} non-redundant edges from each of its constituent nodes. This is because we picked the first 2^{2k} edges from each of their edge lists (assuming they exist), which were sorted to start with. We then purge redundant edges and sort this collection of $O(2^k \times 2^{2k})$ edges.

At the next PCS, $I = 2k$, each node could become a part of a component of size $\leq 2^{2k}$ by Lemma1.1. So, it could have lost at most $2^{2k} - 1$ edges as a result of their becoming redundant. But L_i was formed using 2^{2k} edges from each constituent node. Hence, at the next PCS, there must be at least 1 non-redundant edge in L_i from each of the constituent nodes, if one exists. Since the edges were picked from a sorted list, there can be no smaller edge from the constituent nodes. Since L_i is sorted, the smallest edge must be the left-most. □

Lemma 2.5 *The minimum edge incident on a component can always be found in $O(1)$ time.*

Proof. When we merge components, we can find the minimum edge incident on each of the constituent components (Lemma 2.4) in $O(1)$ time. The question is: "How do we find the minimum of these minima?" By Lemma 2.2, we can do so in $O(1)$ time provided that the number of processors available is greater than the square of the number of constituent components. If not, we perform CCS after which finding the minimum edge is straight-forward.

Lemma 2.6 *All edges are between leaders. Precisely, $\forall e = (u, v)$, $P[u] = u$ and $P[v] = v$.*

Proof. Direct consequence of Step 7. □

Lemma 2.7 *Let n be the number of unmerged nodes at the start of a step and let $h(n)$ be the number of unmerged nodes at the end of the step. Then, $E[h(n)] \leq \frac{7n}{8}$.*

Proof. There are three types of nodes which we shall consider separately.
(i) Leaders: Each leader has an edge to some other leader, since, by assumption, the graph is connected and, by Lemma 2.6, any edge that a leader has is to another leader. Hence, for each leader, i, $Min\_edge[i] = j$, for some leader j. i is merged if $Colour[i] = White$ and $Colour[j] = Black$. Hence, P[i is merged] $= \frac{1}{4}$.
(ii) Passive white nodes: A passive white node, w_1, either has a pointer to another white node w_2 ($pred[w_1] = w_2$) or to a passive black node, b. w_1 is merged if $Sex[w_1] = Female$ and $Sex[w_2] = Male$. (Recall that $Sex[b] = Male$ by definition.)
(iii) Passive black nodes: These cannot merge. However, the number of passive black nodes can be at most $\frac{1}{2}$ the total number of nodes because each passive black node, b, must have at least one passive white node $w : Marked[(b, w)] = true \wedge Min\_edge[w] = b$.

Therefore, for at least $\frac{1}{2}$ the nodes, the probability of merging is $\frac{1}{4}$. Hence, $E[h(n)] \leq \frac{7n}{8}$. □

Lemma 2.8 *Let $T(n)$ = number of steps before all edges are redundant. Then, $T(n) = EO(\log n)$*

Proof. From Lemma 2.7, $T(n) = 1 + T(h(n))$, $E[(h(n)] \leq \frac{7n}{8}$ By Theorem 3 of [9], it follows that $P[T(n) \geq w + 1 + \lfloor \log_{8/7} n \rfloor] \leq \frac{7}{8}^{(w-1)} \frac{n}{8/7^{(\lfloor \log_{8/7} n \rfloor + 1)}}$, for a constant w. (We use essentially the same on Randomized Tree Contraction in [9]). A little manipulation yields $P[T(n) \geq (w+2)(\lfloor \log_{8/7} n + 1 \rfloor)] \leq \frac{1}{n^w}$ The proof follows. □

Theorem 2.1 *The Minimum Spanning Tree of a weighted undirected graph $G = (V, E)$ where $|V| = n$ and $|E| = m$ can be computed in $EO(\log n)$ time using $(m+n)\log n$ processors and in $EO(\log n \log^* n)$ time using $(m+n)$ processors on a Common CRCW PRAM.*

Proof. By Lemma 2.8, the algorithm requires $EO(\log n)$ iterations. Steps 1, 2, 3, 5(a),5(b), 6(a), 6(c) and 7 take $O(1)$ time using $m+n$ processors. Using the algorithm cited in Lemma 2.1, Step 4 requires $O(1)$ time. Using the algorithm cited in Lemma 2.3, Step 6(b) requires $O(1)$ time using $(m+n)\log n$ processors or $O(\log^* n)$ time using $(m+n)$ processors. Step 6(d) is merging which requires the same time as Step 5. Steps 5(c) and 5(d) require $O(1)$ time because we are merging exactly 2 node lists and edge lists respectively. By Lemmas 1.2 and 1.3, PCS and CCS take $O(1)$ amortized time per step. □

References

[1] Awerbuch, B. and Shiloach, Y. "New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM" *IEEE Trans. on Computers* Vol. 36, 1987, pp.1258-1263

[2] Berkman, O. and Vishkin, U. "Recursive *-Tree Parallel data Structures" *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science* 1989, pp. 196-202

[3] Chin, F.Y., Lam, J. ad Chen, I. "Efficient Parallel Algorithms for some Graph Problems" *Communications of the ACM* Vol. 25, 1982, pp. 659-665.

[4] Cole, R. "Parallel Merge Sort" *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science* 1986, pp. 511-516.

[5] Cole, R. and Vishkin, U. "Optimal parallel algorithms for expression tree evaluation and list ranking". *Proceedings of the Aegean Workshop on Computing*, 1988, pp. 91-100.

[6] Fich, F.E., Ragde, P.L., and Wigderson, A. "Relations between concurrent write models of parallel computation" *Proceedings of the 3rd Annual Symposium on Principles of Distributed Computing* 1984, pp. 179-189

[7] Graham, R.L. and Hell, P. "On the History of the Minimum Spanning Tree Problem" *Annals of the History of Computing* Vol. 7, 1985, pp. 43-57.

[8] Hirschberg, D.S. "Parallel Graph Algorithms Without Memory Conflicts" *Proceedings of the 29th Allerton Conference on Communications, Control and Computing* 1982, pp. 257-263

[9] Karp, R.M. "Probabilistic Recurrence Relations" *Proc. 23rd Annual IEEE Symposium on Theory of Computing* 1991, pp. 190-197.

[10] "A survey of parallel algorithms for shared memory machines" *Theoretical Computer Science*, North Holland, 1990.

[11] Ladner, R.E., and Fischer, M.J. "Parallel prefix computations" *JACM* Vol. 27, 1980, pp. 831-838.

[12] Miller, G.L., and Reif, J.H. "Parallel tree Contraction and its Applications" *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp.478-489.

[13] Shiloach, Y. and Vishkin, U. "Finding the maximum, merging and sorting in a parallel computation model" *Journal of Algorithms* Vol. 2, 1981, pp. 88-102.