Concurrent Threads and Optimal Parallel Minimum Spanning Trees Algorithm †

Ka Wong Chong‡ Yijie Han§ Tak Wah Lam‡

July 4, 2000

Abstract. This paper resolves a long-standing open problem on whether the concurrent write capability of parallel random access machine (PRAM) is essential for solving fundamental graph problems like connected components and minimum spanning trees in $O(\log n)$ time. Specifically, we present a new algorithm to solve these problems in $O(\log n)$ time using a linear number of processors on the exclusive-read exclusive-write PRAM. The logarithmic time bound is actually optimal since it is well known that even computing the "OR" of n bits requires $\Omega(\log n)$ time on the exclusive-write PRAM. The efficiency achieved by the new algorithm is based on a new schedule which can exploit a high degree of parallelism.

1 Introduction

Given a weighted undirected graph G with n vertices and m edges, the minimum spanning tree (MST) problem is to find a spanning tree (or spanning forest) of G with the smallest possible sum of edge weights. This problem has a rich history. Sequential MST algorithms running in $O(m \log n)$ time were known a few decades ago (see Tarjan [1983] for a survey). Subsequently, a number of more efficient MST algorithms have been published. In particular, Fredman and Tarjan [1987] gave an algorithm running in $O(m\beta(m,n))$ time, where $\beta(m,n) = \min\{i \mid \log^{(i)} n \leq m/n\}$. This time complexity was improved to $O(m \log \beta(m,n))$ by Gabow *et al.* [1986]. Chazelle [1997] presented an even faster MST algorithm with time complexity $O(m\alpha(m,n) \log \alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackerman function. Recently, Chazelle [1999] improved his algorithm to run in $O(m\alpha(m,n))$ time, and later Pettie [1999] independently devised a similar algorithm with the same time complexity. More recently, Pettie and Ramachandran [2000] obtained an algorithm running in optimal time. A simple randomized algorithm running in linear expected time has also been found [Karger *et al.* 1995].

In the parallel context, the MST problem is closely related to the connected component (CC) problem, which is to find the connected components of an undirected graph. The CC problem actually admits a faster algorithm in the sequential context, yet the two problems can be solved by similar techniques on various models of parallel random access machines (see the surveys by JáJá [1992] and Karp and Ramachandran

†A preliminary version of this paper appears in the proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, Maryland). ACM, New York, SIAM, Philadelphia, pp. 225-234.

‡Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong. Email: {kwchong,twlam}@csis.hku.hk. Part of this work was done while the first author was with Max-Planck-Institut für Informatik, Saarbrücken, Germany. This work was supported in part by Hong Kong RGC Grant HKU-289/95E.

§Computer Science Telecommunications Program, University of Missouri – Kansas City, 5100 Rockhill Road, Kansas, MO 64110, USA. Email: han@cstp.umkc.edu.

[1990]). With respect to the model with concurrent write capability (i.e., processors can write into the same shared memory location simultaneously), both problems can be solved in $O(\log n)$ time using $n+m$ processors [Awerbuch and Shiloach 1987; Cole and Vishkin 1986]. Using randomization, Gazit's algorithm [1986] can solve the CC problem in $O(\log n)$ expected time using $(n+m)/\log n$ processors. The work of this algorithm (defined as the time-processor product) is $O(n+m)$ and thus optimal. Later, Cole *et al.* [1996] obtained the same result for the MST problem.

For the exclusive write models (including both concurrent-read exclusive-write and exclusive-read exclusive-write PRAMs), $O(\log^2 n)$ time algorithms for the CC and MST problems were developed two decades ago [Chin *et al.* 1982; Hirschberg 1979]. For a while, it was believed that exclusive write models could not overcome the $O(\log^2 n)$ time bound. The first breakthrough was due to Johnson and Metaxas [1991, 1992]; they devised $O(\log^{1.5} n)$ time algorithms for the CC problem and the MST problem. These results were improved by Chong and Lam [1993] and Chong [1996] to $O(\log n \log\log n)$ time. If randomization is allowed, the time or the work can be further improved. In particular, Karger *et al.* [1992] showed that the CC problem can be solved in $O(\log n)$ expected time, and later Halperin and Zwick [1996] improved the work to linear. For the MST problem, Karger [1995] obtained a randomized algorithm using $O(\log n)$ expected time (and super-linear work), and Poon and Ramachandran [1997] gave a randomized algorithm using linear expected work and $O(\log n \cdot \log\log n \cdot 2^{\log^* n})$ expected time.

Another approach stems from the fact that deterministic space bounds for the *st*-connectivity problem immediately imply identical time bounds for EREW algorithms for the CC problem. Nisan *et al.* [1992] have shown that *st*-connectivity problem can be solved deterministically using $O(\log^{1.5} n)$ space, and Armoni *et al.* [1997] further improved the bound to $O(\log^{4/3} n)$. These results imply EREW algorithm for solving the CC problem in $O(\log^{1.5} n)$ time and $O(\log^{4/3} n)$ time, respectively.

Prior to our work, it had been open whether the CC and MST problems could be solved deterministically in $O(\log n)$ time on the exclusive write models. Notice that $\Theta(\log n)$ is optimal since these graphs problems are at least as hard as computing the OR of n bits. Cook *et al.* [1986] have proven that the latter requires $\Omega(\log n)$ time on the CREW or EREW PRAM no matter how many processors are used.

Existing MST algorithms (and CC algorithms) are difficult to improve because of the locking among the processors. As the processors work on different parts of the graph having different densities, the progress of the processors is not uniform, yet the processors have to coordinate closely in order to take advantage of the results computed by each other. As a result, many processors often wait rather than doing useful computation. This paper presents a new parallel paradigm for solving the MST problem, which requires minimal coordination among the processors so as to fully utilize the parallelism. Based on new insight into the structure of minimum spanning trees, we show that this paradigm can be implemented on the EREW, solving the MST problem in $O(\log n)$ time using $n+m$ processors. The algorithm is deterministic in nature and does not require special operations on edge weights (other than comparison).

Finding connected components or minimum spanning trees is often a key step in the parallel algorithms for other graph problems (see e.g., Miller and Ramachandran

[1986]; Maon *et al.* [1986]; Tarjan and Vishkin [1985]; Vishkin [1985]). With our new MST algorithm, some of these parallel algorithms can be immediately improved to run in optimal (i.e., $O(\log n)$) time without using concurrent write (e.g., biconnectivity [Tarjan and Vishkin 1985] and ear decomposition [Miller and Ramachandran 1986]).

From a theoretical point of view, our result illustrates that the concurrent write capability is not essential for solving a number fundamental graph problems efficiently. Notice that EREW algorithms are actually more practical in the sense that they can be adapted to other more realistic parallel models like the Queuing Shared Memory (QSM) [Gibbon *et al.* 1997] and the Bulk Synchronous Parallel (BSP) model [Valiant 1990]. The latter is a distributed memory model of parallel computation. Gibbon *et al.* [1997] showed that an EREW PRAM algorithm can be simulated on the QSM model with a slow down by a factor of g, where g is the bandwidth parameter of the QSM model. Such a simulation is, however, not known for the CRCW PRAM. Thus, our result implies that the MST problem can be solved efficiently on the QSM model in $O(g \log n)$ time using a linear number of processors. Furthermore, Gibbon *et al.* derived a randomized work-preserving simulation of a QSM algorithm with a logarithmic slow down on the BSP model.

The rest of the paper is organized as follows. Section 2 reviews several basic concepts and introduces a notion called concurrent threads for finding minimum spanning trees in parallel. Section 3 describes the schedule used by the threads, illustrating a limited form of pipelining (which has a flavor similar to the pipelined merge-sort algorithm by Cole [1988]). Section 4 lays down the detailed requirement for each thread. Section 5 shows the details of the algorithm. To simplify the discussion, we first focus on the CREW PRAM, showing how to solve the MST problem in $O(\log n)$ time using $(n+m)\log n$ processors. In Section 6 we adapt algorithm to run on the EREW PRAM and reduce the processor bound to linear.

Remark: Very recently, Pettie and Ramachandran [1999] made use of the result in this paper to further improve existing randomized MST algorithms. Precisely, their algorithm is the first one to run, with high probability, in $O(\log n)$ time and linear work on the EREW PRAM.

2 Basics of parallel MST algorithms: past and present

In this section we review a classical approach to finding an MST. Based on this approach, we can easily contrast our new MST algorithm with existing ones.

We assume that the input graph G is given in the form of adjacency lists. Consider any edge $e = (u, v)$ in G. Note that e appears in the adjacency lists of u and v. We call each copy of e the *mate* of the other. When we need to distinguish them, we use the notations $\langle u, v\rangle$ and $\langle v, u\rangle$ to signify that the edge originates from u and v respectively. The weight of e, which can be any real number, is denoted by $w(e)$ or $w(u, v)$. Without loss of generality, we assume that the edge weights are all distinct. Thus, G has a unique minimum spanning tree, which is denoted by T^*_G throughout this paper. We also assume that G is connected (otherwise, our algorithm finds the minimum spanning forest of G).

Let B be a subset of edges in G which contains no cycle. B induces a set of trees $F = \{T_1, T_2, \cdots, T_l\}$ in a natural sense—Two vertices in G are in the same tree if they are connected by edges of B. If B contains no edge incident on a vertex v, then v itself forms a tree.

Definition: Consider any edge $e = (u, v)$ in G and any tree $T \in F$. If both u and v belong to T, e is called an *internal* edge of T; if only one of u and v belongs to T, e is called an *external* edge. Note that an edge of T is also an internal edge of T, but the converse may not be true.

Definition: B is said to be a λ-forest if each tree $T \in F$ has at least λ vertices.

For example, if B is the empty set then B is a 1-forest of G; a spanning tree such as T^*_G is an n-forest. Consider a set B of edges chosen from T^*_G. Assume that B is a λ-forest. We can augment B to give a 2λ-forest using a greedy approach: Let F' be an arbitrary subset of F such that F' includes all trees $T \in F$ with fewer than 2λ vertices (F' may contain trees with 2λ or more vertices). For every tree in F', we pick its minimum external edge. Denote B' as this set of edges.

LEMMA 2.1. *[JáJá 1992, Lemma 5.4] B' consists of edges in T^*_G only.*

LEMMA 2.2. *$B \cup B'$ is a 2λ-forest.*

Proof. Every tree in $F - F'$ already contains at least 2λ vertices. Consider a tree T in F'. Let $\langle u, v\rangle$ be the minimum external edge of T, where v belongs to another tree $T' \in F$. With respect to $B \cup B'$, all the vertices in T and T' are connected together. Among the trees induced by $B \cup B'$, there is one including T and T', and it contains at least 2λ vertices. Therefore, $B \cup B'$ is a 2λ-forest of G. ■

Based on Lemmas 2.1 and 2.2, we can find T^*_G in $\lfloor \log n \rfloor$ stages as follows:

Notation: Let $B[p, q]$ denote $\cup_{k=p}^{q} B_k$ if $p \le q$, and the empty set otherwise.

procedure Iterative-MST(G)

1. **for** $i = 1$ to $\lfloor \log n \rfloor$ **do** /* Stage i */
   (a) Let F be the set of trees induced by $B[1, i-1]$ on G. Let F' be an arbitrary subset of F such that F' includes all trees $T \in F$ with fewer than 2^i vertices.
   (b) $B_i \leftarrow \{e \mid e \text{ is the minimum external edge of } T \in F'\}$
2. **return** $B[1, \lfloor \log n \rfloor]$.

At Stage i, different strategies for choosing the set F' in Step 1(a) may lead to different B_i's. Nevertheless, $B[1, i]$ is always a subset of T^*_G and induces a 2^i-forest. In particular, $B[1, \lfloor \log n \rfloor]$ induces a $2^{\lfloor \log n \rfloor}$-forest, in which each tree, by definition, contains at least $2^{\lfloor \log n \rfloor} > n/2$ vertices. In other words, $B[1, \lfloor \log n \rfloor]$ induces exactly one tree, which is equal to T^*_G. Using standard parallel algorithmic techniques, each stage can be implemented in $O(\log n)$ time on the EREW PRAM using a linear number of processors (see e.g., JáJá [1992]). Therefore, T^*_G can be found in $O(\log^2 n)$ time. In fact, most parallel algorithms for finding MST (including those CRCW PRAM algorithms) are based on a similar approach (see e.g., [Awerbuch and Shiloach [1987]; Chin *et al.* [1982]; Cole and Vishkin [1986]; Chong and Lam [1993]; Chong [1996]; Johnson and Metaxas [1991, 1992]; Karger *et al.* [1992]). These parallel algorithms are "sequential" in the sense that the computation of B_i starts only after B_{i-1} is available (see Figure 1(a)).

An innovative idea exploited by our MST algorithm is to use concurrent *threads* to compute the B_i's. Threads are groups of processors working on different tasks, the computation of the threads being independent of each other. In our algorithm, there are $\lfloor \log n \rfloor$ concurrent threads, each finding a particular B_i. These threads are characterized by the fact that the thread for computing B_i starts long before the thread for computing B_{i-1} is completed, and actually outputs B_i in $O(1)$ time after B_{i-1} is found (see Figure 1(b)). As a result, T^*_G can be found in $O(\log n)$ time.

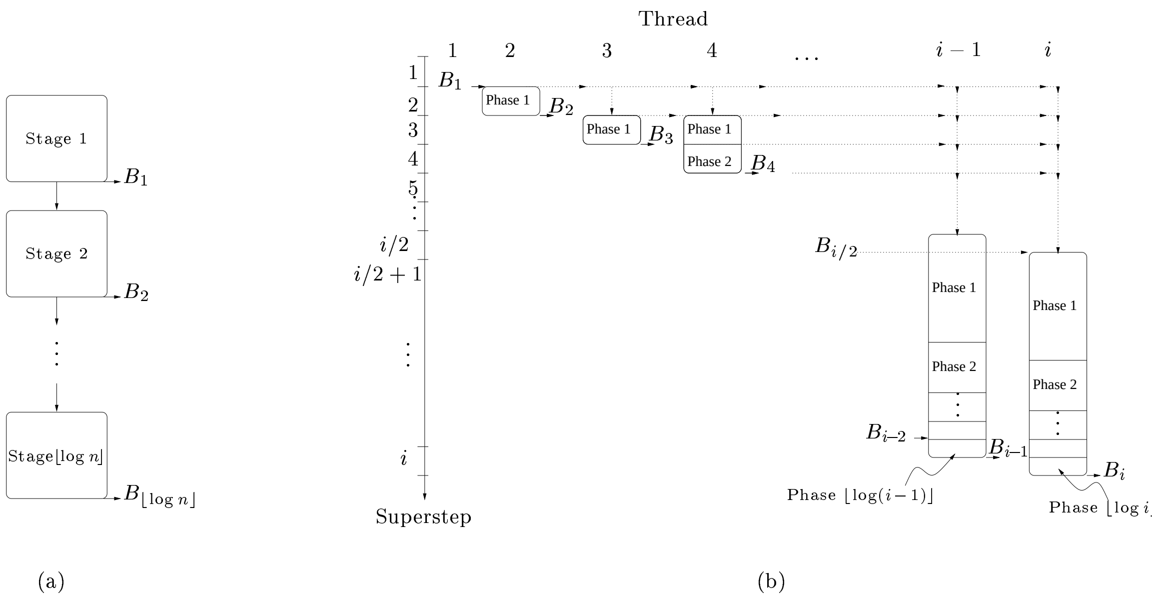

Figure 1: (a) The iterative approach. (b) The concurrent-thread approach.

Our algorithm takes advantage of an interesting property of the sets $B_1, B_2, \cdots, B_{\lfloor \log n \rfloor}$. This property actually holds with respect to most of the deterministic algorithms for finding an MST, though it has not mentioned explicitly in the literature.

Lemma 2.3. *Let T be one of the trees induced by $B[1,k]$, for any $0 \leq k \leq \lfloor \log n \rfloor$. Let e_T be the minimum external edge of T. For any subtree (i.e., connected subgraph) S of T, the minimum external edge of S is either e_T or an edge of T.*

Proof. See Appendix. ■

3 Overview and Schedule

Our algorithm consists of $\lfloor \log n \rfloor$ threads running concurrently. For $1 \leq i \leq \lfloor \log n \rfloor$, Thread i aims to find a set B_i which is one of the possible sets computed at Stage i of the procedure Iterative-MST. To be precise, let F be the set of trees induced by $B[1, i-1]$

and let F' be an arbitrary subset of F including all trees with fewer than 2^i vertices; B_i contains the minimum external edges of the trees in F'. Thread i receives the output of Threads 1 to $i-1$ (i.e., $B_1, \cdots, B_{i-1}$) incrementally, but never looks at their computation. After B_{i-1} is found, Thread i computes B_i in a further of $O(1)$ time.

3.1 Examples

Before showing the detailed schedule of Thread i, we give two examples illustrating how Thread i can speed up the computation of B_i. In Examples 1 and 2, Thread i computes B_i in time ci and $\frac{1}{2}ci$ respectively after Thread $(i-1)$ reports B_{i-1}, where c is some fixed constant. To simplify our discussion, these examples assume that the adjacency lists of a set of vertices can be "merged" into a single list in $O(1)$ time. At the end of this section, we will explain why this is infeasible in our implementation and highlight our novel observations and techniques to evade the problem.

Thread i starts with a set $\mathcal{Q}_0$ of adjacency lists, where each list contains the $2^i - 1$ smallest edges incident on a vertex in G. The edges kept in $\mathcal{Q}_0$ are already sufficient for computing B_i. The reason is as follows: Consider any tree T induced by $B[1, i-1]$. Assume the minimum external edge e_T of T is incident on a vertex v of T. If T contains fewer than 2^i vertices, at most $2^i - 2$ edges incident on v are internal edges of T. Thus, the $2^i - 1$ smallest edges incident on v must include e_T.

Example 1: This is a straightforward implementation of Lemma 2.2. Thread i starts only when $B_1, \cdots, B_{i-1}$ are all available. Let F be the set of trees induced by $B[1, i-1]$. Suppose we can merge the adjacency lists of the vertices in each tree, forming a single combined adjacency list. Notice hat if a tree in F has fewer than 2^i vertices, its combined adjacency list will contain at most $(2^i - 1)^2$ edges. For each combined list with at most $(2^i - 1)^2$ edges, we can determine the minimum external edge in time ci, where c is some suitable constant. The collection of such minimum external edges is reported as B_i. We observe that a combined adjacency list with more than $(2^i - 1)^2$ edges represents a tree containing at least 2^i vertices. By the definition of B_i, it is not necessary to report the minimum external edge of such a tree.

Example 2: This example is slightly more complex, illustrating how Thread i works in an "incremental" manner. Thread i starts off as soon as $B_{i/2}$ has been computed. At this point, only $B_1, \cdots, B_{i/2}$ are available and Thread i is not ready to compute B_i. Nevertheless, it performs some preprocessing (called Phase I below) so that when $B_{i/2+1}, \cdots, B_{i-1}$ become available, the computation of B_i can be speeded up to run in time $\frac{1}{2}ci$ only (Phase II).

Phase I: Let F' be the set of trees induced by $B[1, i/2]$. Again, suppose we can merge the adjacency lists in $\mathcal{Q}_0$ for every tree in F', forming another set $\mathcal{Q}'$ of adjacency lists. By the definition of $B[1, i/2]$, each tree in F' contains at least $2^{i/2}$ vertices. For each tree in F' with fewer than 2^i vertices, its combined adjacency list contains at most $(2^i - 1)^2$ edges. We extract from the list the $2^{i/2} - 1$ smallest edges such that each of them connects to a distinct tree in F'. These edges are sufficient for finding B_i (the argument is an extension of the argument in Example 1). The computation takes time ci only.

Phase II: When $B_{i/2+1}, \cdots, B_{i-1}$ are available, we compute B_i based on $\mathcal{Q}'$ as follows: Edges in $B[i/2+1, i-1]$ further connect the trees in F', forming a set F of bigger trees.

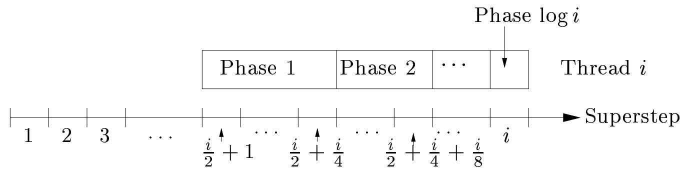

Figure 2: The schedule of Thread i, where i is a power of 2.

Suppose we can merge the lists in $\mathcal{Q}'$ for every tree in F. Notice that if a tree in F contains fewer than 2^i vertices, it is composed of at most $2^{i/2}-1$ trees in F' and its combined adjacency list contains no more than $(2^{i/2}-1)^2$ edges. In this case, we can find the minimum external edge in at most a further of $\frac{1}{2}ci$ time. B_i is the set of minimum external edges just found. In conclusion, after B_{i-1} is computed, B_i is found in time $\frac{1}{2}ci$.

Remark: The set B_i found by Examples 1 and 2 may be different. Yet in either case, $B_i \cup B[1, i-1]$ is a subset of T_G^* and a 2^i-forest.

3.2 The schedule

Our MST algorithm is based on a generalization of the above ideas. The computation of Thread i is divided into $\lfloor \log i \rfloor$ phases. When Thread $i-1$ has computed B_{i-1}, Thread i is about to enter its last phase, which takes $O(1)$ to report B_i. See Figure 1(b).

Globally speaking, our MST algorithm runs in $\lfloor \log n \rfloor$ supersteps, where each superstep lasts $O(1)$ time. In particular, Thread i delivers B_i at the end of the ith superstep. Let us first consider i a power of two. Phase 1 of Thread i starts at the $(i/2+1)$th superstep, i.e., when $B_1, \cdots, B_{i/2}$ are available. The computation takes no more than $i/4$ supersteps, ending at the $(i/2+i/4)$th superstep. Phase 2 starts at the $(i/2+i/4+1)$th superstep (i.e., when $B_{i/2+1}, \cdots, B_{i/2+i/4}$ are available) and uses $i/8$ supersteps. Each subsequent phase uses half as many supersteps as the preceding phase. The last phase (Phase $\log i$) starts and ends within the ith superstep. See Figure 2.

For general i, Thread i runs in $\lfloor \log i \rfloor$ phases. To mark the starting time of each phase, we define the sequence

$$a_j = i - \left\lfloor i/2^j \right\rfloor, \text{ for } j \geq 0.$$

(That is, $a_0 = 0$, $a_1 = \lceil i/2 \rceil$, ..., $a_{\lfloor \log i \rfloor} = i-1$.) Phase j of Thread i, where $1 \leq j \leq \lfloor \log i \rfloor$, starts at the (a_j+1)th superstep and uses $a_{j+1} - a_j = \lfloor i/2^j \rfloor - \lfloor i/2^{j+1} \rfloor = \left\lceil \frac{1}{2} \lfloor i/2^j \rfloor \right\rceil$ supersteps. Phase j has to handle the edge sets $B_{a_{j-1}+1}, \cdots, B_{a_j}$, which are made available by other threads during the execution of Phase $(j-1)$.

3.3 Merging

In the above examples, we assume that for every tree in F, we can merge the adjacency lists of its vertices (or subtrees in Phase II of Example 2) into a single list efficiently and the time does not depend on the total length. This can be done via the technique introduced by Tarjan and Vishkin [1985]. Let us look at an example. Suppose a tree T contains an edge e between two vertices u and v. Assume that the adjacency lists of u and v contain e and its mate respectively. The two lists can be combined by having e and its mate exchange their successors (see Figure 3). If every edge of T and its mate exchange their successors in their adjacency lists, we will get a combined adjacency list for T in $O(1)$ time. However, the merging fails if any edge of T or its mate is not included in the corresponding adjacency lists.

In our algorithm, we do not keep track of all the edges for each vertex (or subtree) because of efficiency. For example, each adjacency list in $\mathcal{Q}_0$ involves only $2^i - 1$ edges incident on a vertex. With respect to a tree T, some of its edges and their mates may not be present in the corresponding adjacency lists. Therefore, when applying the $O(1)$-time merging technique, we may not be able to megre the adjacency lists into one single list for representing T. Failing to form a single combined adjacency list also complicates the extraction of essential edges (in particular, the minimum external edges) for computing the set B_i. In particular, we cannot easily determine all the vertices belonging to T and identify the redundant edges, i.e., internal and extra multiple external edges, in the adjacency list of T.

Actually our MST algorithm does not insist on merging the adjacency lists into a single list. A key idea here is that our algorithm can maintain all essential edges to be included in just one particular combined adjacency list. Based on some structural properties of minimum spanning trees, we can filter out redundant adjacency lists to obtain a unique adjacency list for T (see Lemmas 5.1 and 5.5 in Section 5).

In the adjacency list representing T, internal edges can all be removed using a technique based on "threshold" [Chong 1996]. The most intriguing part concerns the extra multiple external edges. We find that it is not necessary to remove all of them. Specifically, we show that those extra multiple external edges that cannot be removed "easily" must have a bigger weight and their presence does not affect the correctness of the computation.

In the next section, we will elaborate on the above ideas and formulate the requirements for each phase so as to achieve the schedule.

4 Requirements for a phase

In this section we specify formally what Thread i is expected to achieve in each phase. Initially (in Phase 0), Thread i constructs a set $\mathcal{Q}_0$ of adjacency lists. For each vertex v in G, $\mathcal{Q}_0$ contains a circular linked list $\mathcal{L}$ including the $2^i - 1$ smallest edges incident on v. In addition, $\mathcal{L}$ is assigned a *threshold*, denoted by $h(\mathcal{L})$. If $\mathcal{L}$ contains all edges of v, $h(\mathcal{L}) = +\infty$; otherwise, $h(\mathcal{L}) = w(e_o)$, where e_o is the smallest edge truncated from $\mathcal{L}$. In each of the $\lfloor \log i \rfloor$ phases, the adjacency lists are further merged based on the newly arrived edge sets, and truncated according to the length requirement. For each combined adjacency list, a new threshold is computed. Intuitively, the threshold records

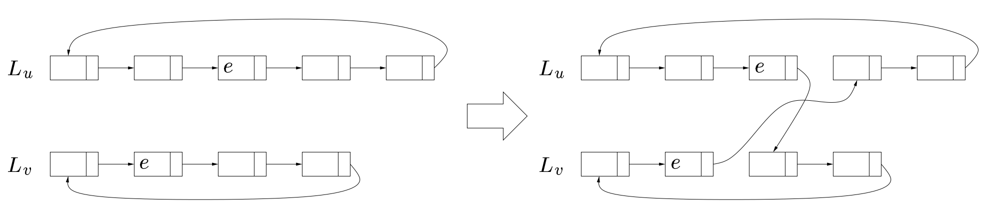

Figure 3: Merging a pair of adjacency lists L_u and L_v with respect to a common edge e.

the smallest edge that has been truncated so far.

Consider Phase j, where $1 \le j \le \lfloor \log i \rfloor$. It inherits a set $\mathcal{Q}_{j-1}$ of adjacency lists from Phase $j-1$ and receives the edge sets $B[a_{j-1}+1, a_j]$ (recall that $a_j = i - \lfloor i/2^j \rfloor$). Let F_j denote the set of trees induced by $B[1, a_j]$. Phase j aims at producing a set $\mathcal{Q}_j$ of adjacency lists capturing the external edges of the trees in F_j that are essential for the computation of B_i. Basically, we try to merge the adjacency lists in $\mathcal{Q}_{j-1}$ with respect to $B[a_{j-1}+1, a_j]$. As mentioned before, this merging process may produce more than one combined adjacency list for each tree $T \in F_j$. Nevertheless, we strive to ensure that only one combined list is retained to represent T. the rest are filtered out. In view of the time constraint imposed by the schedule of Thread i, we also need a tight bound on the length of each remaining adjacency list.

Let $\mathcal{L}$ be a list in $\mathcal{Q}_j$.

R1 (representation): $\mathcal{L}$ uniquely corresponds to a tree $T \in F_j$, storing only the external edges of T. In this case, T is said to be represented by $\mathcal{L}$ in $\mathcal{Q}_j$. Some trees in F_j may not be represented by any lists in $\mathcal{Q}_j$; however, all trees with fewer than 2^i vertices are represented.

R2 (length): $\mathcal{L}$ contains at most $2^{\lfloor i/2^j \rfloor} - 1$ edges.

We will define what edges of a tree $T \in F_j$ are essential and must be included in $\mathcal{L}$. Consider an external edge e of T that connects to another tree $T' \in F_j$. We say that e is *primary* if, among all edges connecting T and T', e has the smallest weight. Otherwise, e is said to be *secondary*. Note that if the minimum spanning tree of G contains an edge which is an external edge of both T and T', it must be a primary one. Ideally, only primary external edges should be retained in each list of $\mathcal{Q}_j$. Yet this is infeasible since Thread i starts off with truncated adjacency lists and we cannot identify and remove all the secondary external edges in each phase. (Removing all internal edges, though non-trivial, is feasible.)

An important observation is that it is not necessary to remove all secondary external edges. Based on a structural classification of *light* and *heavy* edges (defined below), we find that all light secondary external edges can be removed easily. Afterwards, each list contains all the light primary external edges and possibly some heavy secondary external edges. The set of light primary external edges may not cover all primary external edges and its size can be much smaller than $2^{\lfloor i/2^j \rfloor} - 1$. Yet we will show that the set of light primary external edges suffices for computing B_i, and the presence of heavy secondary external edges does not affect the correctness.

Below we give the definition of light and heavy edges, which are based on the notion of *base*.

Definition: Let T be a tree in F_j.

- Let γ be any real number. A tree $T' \in F_j$ is said to be γ-accessible to T if $T = T'$, or there is another tree $T'' \in F_j$ such that T'' is γ-accessible to T and connected to T' by an edge with weight smaller than γ.
- Let e be an external (or internal) edge of T. Define *base*(F_j, T, e) to be the set

$\{T' \mid T' \in F_j$ and T' is $w(e)$-accessible to $T\}$. The size of $base(F_j, T, e)$, denoted by $\|base(F_j, T, e)\|$, is the total number of vertices in the trees involved.

- Let e be an external (or internal) edge of T. We say that e is *light* if $\|base(F_j, T, e)\| < 2^i$; otherwise, e is *heavy.*

It follows from the above definition that a light edge of a tree T has a smaller weight than a heavy edge of T. Also, a heavy edge of T will remain a heavy edge in subsequent phases. More specifically, in any Phase k where $k > j$, if T is a subtree of some tree $X \in F_k$, then for any external (or internal) edge e of T, $\|base(F_j, T, e)\| \leq \|base(F_k, X, e)\|$. Therefore, if e is heavy with respect to T then it is also heavy with respect to X.

The following lemma gives an upper bound on the number of light primary external edges of each tree in F_j, which complies with the length requirement of the lists in $\mathcal{Q}_j$.

LEMMA 4.1. *Any tree $T \in F_j$ has at most $2^{\lfloor i/2^j \rfloor} - 1$ light primary external edges.*

Proof. Let x be the number of light primary external edges of T. Among the light primary external edges of T, let e be the one with the biggest weight. The set $base(F_j, T, e)$ includes T and at least $x-1$ trees adjacent to T. As $B[1, a_j]$ is a 2^{a_j}-forest, every tree in F_j contains at least 2^{a_j} vertices. We have $\|base(F_j, T, e)\| \geq 2^{a_j} + (x-1) \cdot 2^{a_j} \geq x \cdot 2^{a_j}$. By definition of a light edge, $\|base(F_j, T, e)\| < 2^i$. Thus, $x \cdot 2^{a_j} < 2^i$ and $x < 2^{i-a_j} = 2^{\lfloor i/2^j \rfloor}$. ■

The following requirement specifies the essential edges to be kept in each list of $\mathcal{Q}_j$ and characterizes those secondary external edges, if any, in each list.

R3 (base) Let T be the tree in F_j represented by a list $\mathcal{L} \in \mathcal{Q}_j$. All light primary external edges of T are included in $\mathcal{L}$, and secondary external edges of T, if included in $\mathcal{L}$, must be heavy.

Retaining only the light primary external edges in each list of $\mathcal{Q}_j$ is already sufficient for the computation of B_i. In particular, let us consider the scenario at the end of Phase $\lfloor \log i \rfloor$. For any tree $T_s \in F_{\lfloor \log i \rfloor}$ with fewer than 2^i vertices, the minimum external edge e_{T_s} of T_s must be reported in B_i. Note that $base(F_{\lfloor \log i \rfloor}, T_s, e_{T_s})$ contains T_s only. Thus, $\|base(F_{\lfloor \log i \rfloor}, T_s, e_{T_s})\| < 2^i$, and e_{T_s} is a light primary external edge of T_s. In all previous phases k, F_k contains a subtree of T_s, denoted by W, of which e_{T_s} is an external edge. Note that e_{T_s} is also a light primary external edge of W (as a heavy edge remains a heavy edge subsequently).

On the other hand, at the end of Phase $\lfloor \log i \rfloor$, if a tree $T_x \in F_{\lfloor \log i \rfloor}$ contains 2^i or more vertices, all its external edges are heavy and R3 cannot enforce the minimum external edge e_{T_x} of T_x being kept in the list for T_x. Fortunately, it is not necessary for Thread i to report the minimum external edge for such a tree. The following requirements for the threshold help us detect whether the minimum external edge of T_x has been removed. If so, we will not report anything for T_x. Essentially, we require that if e_{T_x} or any primary external edge e of T_x has been removed from the list $\mathcal{L}_x$ that represents T_x, the threshold kept in $\mathcal{L}_x$ is no bigger than $w(e_{T_x})$ (respectively, $w(e)$). Then the smallest edge in $\mathcal{L}_x$ is e_{T_x} if and only if its weight is fewer than the threshold.

Let T be a tree in F_j represented by a list $\mathcal{L} \in \mathcal{Q}_j$. The threshold of $\mathcal{L}$ satisfies the following properties.

R4 (lower bound for the threshold) If $h(\mathcal{L}) \neq \infty$, then $h(\mathcal{L})$ is equal to the weight of a heavy internal or external edge of T.

R5 (upper bound for the threshold) Let e be an external edge of T not included in $\mathcal{L}$. If e is primary, then $h(\mathcal{L}) \leq w(e)$.

(Our algorithm actually satisfies a stronger requirement that $h(\mathcal{L}) \leq w(e)$ if
- e is primary, or
- e is secondary and the mate of e is still included in another list $\mathcal{L}'$ in $\mathcal{Q}_j$.)

In summary, R1 to R5 guarantee that at the end of Phase $\lfloor \log i \rfloor$, for any tree $T \in F_{\lfloor \log i \rfloor}$, if T has fewer than 2^i vertices, its minimum external edge e_T is the only edge kept in a unique adjacency list representing T; otherwise, T may or may not be represented by any list. If T is represented by a list but e_T has already been removed, the threshold kept is at most $w(e_T)$. Every external edge currently kept in the list must have a weight greater than or equal to the threshold. Thus, we can simply ignore the list for T.

It is easy to check that $\mathcal{Q}_0$ satisfies the five requirements for Phase 0. In the next section we will give an algorithm that can satisfy these requirements after every phase. Consequently, Thread i can report B_i based on the edges in the lists in $\mathcal{Q}_{\lfloor \log i \rfloor}$.

5 The Algorithm

In this section we present the algorithmic details of Thread i, showing how to merge and extract the adjacency lists in each phase. The discussion is inductive in nature—for any $j \geq 1$, we assume that Phase $j-1$ has produced a set of adjacency lists satisfying the requirements R1-R5, and then show how Phase j computes a new set of adjacency lists satisfying the requirements in $O(i/2^j)$ time using a linear number of processors.

Phase j inherits the set of adjacency lists $\mathcal{Q}_{j-1}$ from Phase $j-1$ and receives the edges $B[a_{j-1}+1, a_j]$. To ease our discussion, we refer to $B[a_{j-1}+1, a_j]$ as *INPUT*. Notice that a list in $\mathcal{Q}_{j-1}$ represents one of the trees in F_{j-1} (recall that F_{j-1} and F_j denote the set of trees induced by $B[1, a_{j-1}]$ and $B[1, a_j]$ respectively). Phase j merges the adjacency lists in $\mathcal{Q}_{j-1}$ according to how the trees in F_{j-1} are connected by the edges in *INPUT*.

Consider an edge $e = (u, v)$ in *INPUT*. Denote W_1 and W_2 as the trees in F_{j-1} containing u and v respectively. Ideally, if e and its mate appear in the adjacency lists of W_1 and W_2 respectively, the adjacency lists of W_1 and W_2 can be merged easily in $O(1)$ time. However, W_1 or W_2 might already be too large and not have a representation in $\mathcal{Q}_{j-1}$. Even if they are represented, the length requirement of the adjacency lists may not allow e to be included. As a result, e may appear in two separate lists in $\mathcal{Q}_{j-1}$, or in just one, or even in none; we call e a *full*, *half*, and *lost* edge respectively. Accordingly, we partition *INPUT* into three sets, namely *Full-INPUT*, *Half-INPUT*, and *Lost-INPUT*.

Phase j starts off by merging the lists in $\mathcal{Q}_{j-1}$ with respect to edges in *Full-INPUT*. Let T be a tree in F_j. Let $W_1, W_2, \cdots, W_k$ be the trees in F_{j-1} that, together with the

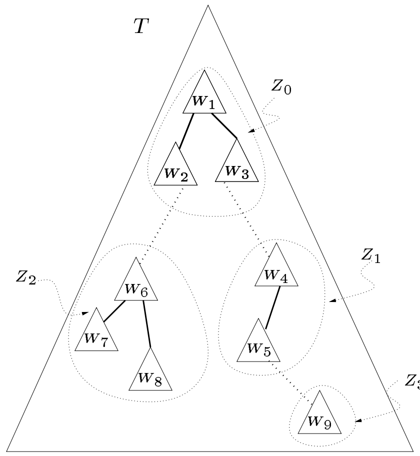

Figure 4: Each W_x represents a tree in F_{j-1}. The dotted and solid lines represent half and full edges in *INPUT* respectively. T is a tree formed by connecting the trees in F_{j-1} with the edges in *INPUT*. Each Z_y (called a cluster) is a subtree of T, formed by connecting some W_x's with full edges only. The adjacency lists of the W_x's within each Z_y can be merged into a single list easily.

edges in *INPUT*, constitute T. Note that some W_i may not be represented by a list in $\mathcal{Q}_{j-1}$. Since the merging is done with respect to *Full-INPUT*, the adjacency lists of $W_1, W_2, \cdots, W_k$, if present, may be merged into several lists instead of a single one. Let $L_1, L_2, \cdots, L_\ell$ denote these merged lists. Each L_i represents a bigger subtree Z_i of T, which is called a *cluster* below (see Figure 4). A cluster may contain one or more W_i. We distinguish one cluster, called the *core cluster*, such that the minimum external edge e_T of T is an external edge of that cluster. Note that the minimum external edge of the core cluster may or may not be e_T. For a non-core cluster Z, the minimum external edge e_Z of Z must be a tree edge of T (by Lemma 2.3) and thus e_Z is in *INPUT*. Moreover, e_Z is not a full edge. Otherwise, the merging should have operated on e_Z, which then becomes an internal edge of a bigger cluster.

The merged lists obviously need not satisfy the requirements for $\mathcal{Q}_j$. In the following sections, we present the additional processing used to fulfill the requirements. A summary of all the processing is given in Section 5.4. The discussion of the processing of the merged lists is divided according to the sizes of the trees, sketched as follows:

- For each tree $T \in F_j$ that contains fewer than 2^i vertices, there is a simple way to ensure that exactly one merged list is retained in $\mathcal{Q}_j$. Edges in that list are filtered to contain all the light primary external edges of T, and other secondary external edges of T, if included, must be heavy.
- For a tree $T' \in F_j$ that contains at least 2^i vertices, the above processing may retain more than one merged lists. Here we put in an extra step to ensure that, except possibly one, all merged lists for T' are removed.
- The threshold of each remaining list is updated after retaining the $2^{\lfloor i/2^j \rfloor} - 1$ smallest edges. We show that the requirements for the threshold are satisfied no matter whether the tree in concern contains fewer than 2^i vertices or not.

5.1 Trees in F_j with fewer than 2^i vertices

In this section we focus on each tree $T \in F_j$ that contains fewer than 2^i vertices. Denote by $L_1, \cdots, L_\ell$ the merged lists representing the clusters of T. Observe that each of these lists contains at most $(2^{\lfloor i/2^{j-1} \rfloor} - 1)^2$ edges. Below, we derive an efficient way to find a unique adjacency list for representing T, which contains all light primary external edges of T.

First of all, we realize that every light primary external edge of T is also a light primary external edge of a tree W in F_{j-1} and must be present in the adjacency list that represents W in $\mathcal{Q}_{j-1}$ (by R3). Thus, all light primary external edges of T (including the minimum external edge of T) are present in some merged lists.

Unique Representation: Let L_{cc} be the list in $\{L_1, L_2, \cdots, L_\ell\}$ such that L_{cc} contains the minimum external edge e_T of T. That is, L_{cc} represents the core cluster Z_0 of T. Our concern is how to remove all other lists in $\{L_1, L_2, \cdots, L_\ell\}$ so that T will be represented uniquely by L_{cc}.

To efficiently distinguish L_{cc} from other lists, we make use of the properties stated in the following lemma. Let L_{nc} be any list in $\{L_1, L_2, \cdots, L_\ell\} - \{L_{cc}\}$. Let Z denote the cluster represented by L_{nc}.

LEMMA 5.1. *(i) L_{cc} does not contain any edge in Half-INPUT. (ii) L_{nc} contains at least one edge in Half-INPUT. In particular, the minimum external edge of Z is in Half-INPUT.*

Proof of Lemma 5.1(i). Assume to the contrary that L_{cc} includes an edge $e = (a, b)$ in *Half-INPUT*; more precisely, $\langle a, b\rangle$ is in L_{cc} and $\langle b, a\rangle$ is not included in any list in $\mathcal{Q}_{j-1}$. Let W and W' be the trees in F_{j-1} connected by e, where $a \in W$ and $b \in W'$. The edge e is a primary external edge of W, as well as of W'. Both W and W' are subtrees of T, and W is also a subtree of Z_0. Below we show that $\|base(F_{j-1}, W', e)\| \geq 2^i$ and T contains at least 2^i vertices. The latter contradicts the assumption about T. Thus, Lemma 5.1(i) follows.

W' is a subtree of T and contains less than 2^i vertices. By R1, $\mathcal{Q}_{j-1}$ contains a list $L_{W'}$ representing W'. By R3, $L_{W'}$ contains all light primary external edges of W'. The edge $\langle b, a\rangle$ is not included in $L_{W'}$ and must be heavy. Therefore, $\|base(F_{j-1}, W', e)\| \geq 2^i$.

Next, we want to show that all trees in $base(F_{j-1}, W', e)$ are subtrees of T. Define T_a and T_b to be the subtrees of T constructed by removing e from T (see Figure 5). Assume that T_a contains the vertex a and T_b the vertex b. W, as well as Z_0, is a subtree of T_a, and W' is a subtree of T_b. By Lemma 2.3, the minimum external edge of T_b is either e_T or e. The former case is impossible because e_T is included in L_{cc} and must be an external edge of Z_0. Thus, e is the minimum external edge of T_b. By definition of base, $base(F_{j-1}, W', e)$ cannot include any trees in F_{j-1} that are outside T_b. In other words, T_b includes all subtrees in $base(F_{j-1}, W', e)$. T_b must have at least 2^i vertices, and so must T. A contradiction occurs. ■

Proof of Lemma 5.1(ii). Let e_Z be the minimum external edge of Z. As e_Z is a tree edge of T, it is in *INPUT* but is not a full edge. In this case, we can further show that e_Z is actually a half edge and included in L_{nc}, thus completing the proof. Let W be the tree

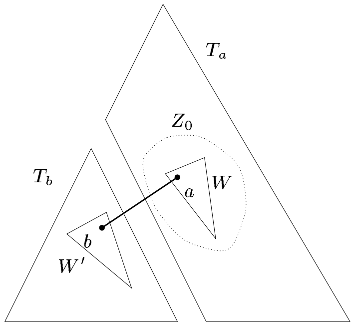

Figure 5: T is partitioned into two subtrees T_a and T_b, which are connected by e.

in F_{j-1} such that W is a component of Z and e_Z is an external edge of W. Note that e_Z is primary external edge of W. Let L_W denote the adjacency list in $\mathcal{Q}_{j-1}$ representing W. Since e_Z is the minimum external edge of Z, $base(F_{j-1}, W, e_Z)$ cannot include trees in F_{j-1} that are outside Z, and thus it has size less than 2^i. By R3, all light primary external edges of W including e_Z are present in L_W. Therefore, e_Z is in *Half-INPUT*, and L_{nc} must have inherited e_Z from L_W. ■

Using Lemma 5.1, we can easily retain L_{cc} and remove all other merged lists L_{nc}. One might worry that some L_{nc} might indeed contain some light primary external edge of T and removing L_{nc} is incorrect. This is actually impossible in view of the following fact.

LEMMA 5.2. *For any external edge e of T that is included in L_{nc}, $\|base(F_j, T, e)\| \geq 2^i$.*

Proof. Let e_Z be the minimum external edge of Z. For any external edge e of T that is included in L_{nc}, e is also an external edge of Z and $w(e) \geq w(e_Z)$.

Let W and W' be the trees in F_{j-1} connected by e_Z such that W' is not a component of Z. As shown in the previous lemma, e_Z is in *Half-INPUT*. Moreover, e_Z is a tree edge of T and is present only in the adjacency list of W. That is, W' is a component of T and the adjacency list of W' in $\mathcal{Q}_{j-1}$ does not contain e_Z. Note that e_Z is a primary external edge of W'. By R1 and R3, we can conclude that e_Z is a heavy external edge of W' and hence $\|base(F_{j-1}, W', e_Z)\| \geq 2^i$. Therefore, $\|base(F_j, T, e)\| \geq \|base(F_j, T, e_Z)\| \geq \|base(F_{j-1}, W', e_Z)\| \geq 2^i$. ■

By Lemma 5.2, L_{nc} does not contain any light primary external edge of T. In other words, all light primary external edges of T must be in L_{cc}, which is the only list retained.

Excluding All Light Internal and Secondary External Edges: L_{cc} contains all the light primary external edges of T and also some other edges. Because of the length requirement (i.e. R2), we retain at most $2^{\lfloor i/2^j \rfloor} - 1$ edges of L_{cc}. Note that the light primary external edges may not be the smallest edges in L_{cc}. Based on the following two lemmas, we can remove all other light edges in L_{cc}, which include the light internal and secondary external edges of T. Then the light primary external edges of T will be the smallest edges left in the list and retaining the $2^{\lfloor i/2^j \rfloor} - 1$ smallest edges will always include all the light primary external edges.

LEMMA 5.3. *Suppose L_{cc} contains a light internal edge $\langle u, v\rangle$ of T. Then its mate, $\langle v, u\rangle$, also appears in L_{cc}.*

Proof. Recall that L_{cc} is formed by merging the adjacency lists of some trees in F_{j-1}. By R1, each of these lists does not contain any internal edge of the tree it represents. If L_{cc} contains a light internal edge $\langle u, v\rangle$ of T, the edge (u, v) must be between two trees W and W' in F_{j-1} which are components of Z_0. Assume that $u \in W$ and $v \in W'$. Then $\langle u, v\rangle$ and $\langle v, u\rangle$ are light external edges of W and W' respectively. Let L_W and $L_{W'}$ be their adjacency lists in $\mathcal{Q}_{j-1}$. As $\langle u, v\rangle$ appears in L_{cc}, $\langle u, v\rangle$ also appears in L_W. By R3, all light edges found in L_W, including $\langle u, v\rangle$, must be primary external edges of W. By symmetry, $\langle v, u\rangle$ is a primary external edge of W'. By R3 again, $\langle v, u\rangle$ appears in $L_{W'}$. Since L_{cc} inherits the edges from both L_W and $L_{W'}$, we conclude that both $\langle u, v\rangle$ and $\langle v, u\rangle$ appear in L_{cc}. ■

LEMMA 5.4. *Suppose L_{cc} contains a light secondary external edge e of T. Let e_0 be the corresponding primary external edge. Then e and e_0 both appear in L_{cc}, and their mates also both appear in another merged list L'_{cc}, where L'_{cc} represents the core cluster of another tree $T' \in F_j$.*

Proof. Suppose L_{cc} contains a light secondary external edge e of T. Assume that e connects T to another tree $T' \in F_j$, and e_0 is the primary external edge between T and T'. As $w(e_0) < w(e)$, $\|base(F_j, T, e_0)\| \le \|base(F_j, T, e)\| < 2^i$. Thus, e_o is also a light

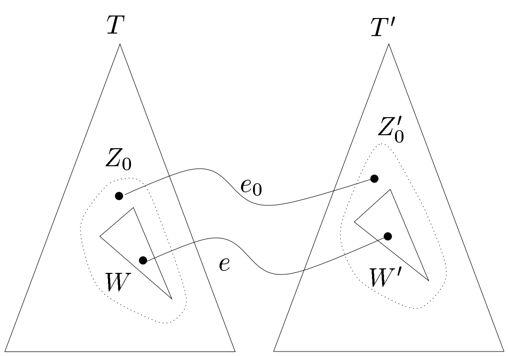

Figure 6: e is a light secondary external edge of T and e_0 is the corresponding primary external edge.

primary external edge of T and must be included in L_{cc}. On the other hand, since e is secondary, $base(F_j, T, e)$ is equal to $base(F_j, T', e)$, and thus $base(F_j, T, e)$ contains T'. Since $\|base(F_j, T, e)\| < 2^i$, T' contains less than 2^i vertices. After merging the lists in $\mathcal{Q}_{j-1}$, we obtain a merged list L'_{cc} that includes all light primary external edges of T'. Below we show that L'_{cc} contains the mates of e_0 and e.

- Observe that $\|base(F_j, T', e_o)\| \le \|base(F_j, T', e)\| = \|base(F_j, T, e)\| < 2^i$. Thus, both e_0 and e are light external edges of T'. As e_0 is also a primary external edge of T', e_0 (more precisely, its mate) must be included in L'_{cc}.
- Let W and W' be the two trees in F_{j-1} connected by e, where W is a subtree of T and W' of T'. Because e is a light external edge of T and T', it is also a light external edge of W and W'. Note that L_{cc} inherits e from the adjacency list $L_W \in \mathcal{Q}_{j-1}$ that represents W. By R3, L_W does not include any light secondary

external edge of W, so e is a primary external edge of W. By symmetry, e is also a primary external edge of W'; thus, by R3, e is in the adjacency list $L_{W'} \in \mathcal{Q}_{j-1}$ that represents W'. Note that L'_{cc} must include all the edges of $L_{W'}$ as well as other lists in $\mathcal{Q}_{j-1}$ that contain light external edges of T (see Lemma 5.2). L'_{cc} contains e, too.

■

By Lemma 5.3, we can remove all light internal edges by simply removing edges whose mates are in the same list. Lemma 5.4 implies that if L_{cc} contains a light secondary external edge, the corresponding primary external edge also appears in L_{cc} and their mates exist in another list L'_{cc}. This suggests a simple way to identify and remove all the light secondary external edges as follows. Without loss of generality, we assume that every edge in L_{cc} can determine the identity of L_{cc} (any distinct label given to L_{cc}). If an edge $e \in L_{cc}$ has a mate in another list, say, L'_{cc}, e can announce the identity of L_{cc} to its mate and vice versa. By sorting the edges in L_{cc} with respect to the identities received from their mates, multiple light external edges connected to the same tree come together. Then we can easily remove all the light secondary external edges.

Now we know that L_{cc} contains all light primary external edges of T and any other edges it contains must be heavy. Let us summarize the steps required to build a unique adjacency list for representing T.

procedure `M&C` // `M&C` means Merge and Clean up //

1. Edges in *INPUT* that are full with respect to $\mathcal{Q}_{j-1}$ are activated to merge the lists in $\mathcal{Q}_{j-1}$. Let $\mathcal{Q}$ be the set of merged adjacency lists.
2. For each merged adjacency list $\mathcal{L} \in \mathcal{Q}$,
   (a) if $\mathcal{L}$ contains an edge in *Half-INPUT*, remove $\mathcal{L}$ from $\mathcal{Q}$;
   (b) detect and remove internal and secondary external edges from $\mathcal{L}$ according to Lemmas 5.3 and 5.4.

5.2 Trees with at least 2^i vertices

Consider a tree $T' \in F_j$ containing 2^i or more vertices. Let $L_1, \cdots, L_\ell$ be the merged lists, each representing a cluster of T'. Some of these lists may contain more than $(2^{\lfloor i/2^{j-1} \rfloor} - 1)^2$ edges. Unlike the case in Section 5.1, the minimum external edge $e_{T'}$ of T' is heavy, and we cannot guarantee that there is a merged list containing $e_{T'}$ and representing the core clusters of T'. Nevertheless, Thread i can ignore such a tree T', and we may remove all the merged lists. In Lemma 5.5, we show that those lists in $\{L_1, \cdots L_\ell\}$ that represent the non-core clusters of T' can be removed easily. If there is indeed a merged list L_{cc} representing the core cluster, Thread i may not remove L_{cc}. Since T' contains at least 2^i vertices and has no light primary external edge, we have nothing to enforce on L_{cc} regarding the light primary external edges. The only concern for L_{cc} is the requirements for the threshold, which will be handled in Section 5.3.

As T' contains at least 2^i vertices, any merged list L_{nc} that represents a non-core cluster of T' may not satisfy the properties stated in Lemma 5.1(ii). We need other ways to detect such an L_{nc}. First, we can detect the length of L_{nc}. If L_{nc} contains more than $(2^{\lfloor i/2^{j-1}\rfloor}-1)^2$ edges, we can remove L_{nc} immediately. Next, if L_{nc} contains less than $(2^{\lfloor i/2^{j-1}\rfloor}-1)^2$ edges, we make use of the following lemma to identify it. Denote $h(L)$ as the threshold associated with a list $L \in \mathcal{Q}_{j-1}$. Define *tmp-h*(L_{nc}) $= \min\{h(L) \mid L \in \mathcal{Q}_{j-1}$ is merged into $L_{nc}\}$.

LEMMA 5.5. *Any list $L_{nc} \in \{L_1, \cdots, L_\ell\} - \{L_{cc}\}$ that represents a non-core cluster of T' satisfies at least one of the following conditions.*

1. *L_{nc} contains an edge in Half-INPUT.*
2. *For every edge $\langle u, v\rangle$ in L_{nc}, either $\langle v, u\rangle$ is also in L_{nc} or $w(u, v) \geq$ tmp-h(L_{nc}).*

Proof. Assume that L_{nc} does not contain any edge in *Half-INPUT*, and L_{nc} contains an edge $\langle u, v\rangle$ but does not contain $\langle v, u\rangle$. Below we show that $w(u, v) \geq$ *tmp-h*(L_{nc}). The edge (u, v) can be an internal or external edge of Z.

Case 1. (u, v) is an internal edge of Z. L_{nc} inherits $\langle u, v\rangle$ from a list $L \in \mathcal{Q}_{j-1}$. Let $W \in F_{j-1}$ be the tree represented by L. By R1, $\langle u, v\rangle$ is an external edge of W. Thus, Z includes another tree $W' \in F_{j-1}$ with $\langle v, u\rangle$ as an external edge, and L_{nc} also inherits the edges in the list $L_{W'} \in \mathcal{Q}_{j-1}$ that represents W'. Note that $\langle v, u\rangle$ does not appear in $L_{W'}$. By R5, $h(L_{W'}) \leq w(u, v)$. Since *tmp-h*$(L_{nc}) \leq h(L_{W'})$, we have *tmp-h*$(L_{nc}) \leq w(u, v)$.

Case 2. (u, v) is an external edge of Z. It is obvious that $w(u, v) \geq w(e_Z)$ where e_Z is the minimum external edge of Z. We further show that $w(e_Z) \geq$ *tmp-h*(L). Let $W \in F_{j-1}$ be the tree in Z and with e_Z as an external edge. Let $L_W \in \mathcal{Q}_{j-1}$ be the adjacency list representing W. As mentioned before, e_Z is in *INPUT* and is not a full edge. If e_Z is a lost edge, then L_{nc} does not contain e_Z. If e_Z is a half edge, L_{nc} again does not contain e_Z because L_{nc} does not contain any edge in *Half-INPUT*. In conclusion, e_Z does not appear in L_{nc} and hence cannot appear in L_W. Since e_Z is a primary external edge of W, we know that, by R5, $h(L_W) \leq w(e_Z)$. By definition, *tmp-h*$(L_{nc}) \leq h(L_W)$. Therefore, *tmp-h*$(L_{nc}) \leq w(e_Z) \leq w(u, v)$. ∎

Using Lemma 5.5, we can extend Procedure `M&C` to remove every merged list L_{nc} that represents a non-core cluster of any tree T in F_j (see Procedure `Ext_M&C`). Precisely, if T has fewer than 2^i vertices, L_{nc} is removed in Step 2(a); otherwise, L_{nc} is removed in Step 1(b) or Steps 2(a)-(c).

procedure `Ext_M&C`

1. (a) Edges in *INPUT* that are full with respect to $\mathcal{Q}_{j-1}$ are activated to merge the lists in $\mathcal{Q}_{j-1}$. Let $\mathcal{Q}$ be the set of merged adjacency lists.
   (b) For each list $\mathcal{L} \in \mathcal{Q}$, if $\mathcal{L}$ contains more than $(2^{\lfloor i/2^{j-1}\rfloor}-1)^2$ edges, remove $\mathcal{L}$ from $\mathcal{Q}$.
2. For each merged adjacency list $\mathcal{L} \in \mathcal{Q}$,
   (a) if $\mathcal{L}$ contains an edge in *Half-INPUT*, remove $\mathcal{L}$ from $\mathcal{Q}$;
   (b) detect and remove internal and secondary external edges from $\mathcal{L}$;
   (c) if, for all edges $\langle u, v\rangle$ in $\mathcal{L}$, $w(u, v) \geq$ *tmp-h*$(\mathcal{L})$, remove $\mathcal{L}$ from $\mathcal{Q}$.

After Procedure `Ext_M&C` is executed, all the remaining merged lists are representing the core clusters of trees in F_j. Moreover, for tree $T \in F_j$ with fewer than 2^i vertices, Procedure `Ext_M&C`, like Procedure `M&C`, always retains the merged list L_{cc} that represents the core cluster of T. L_{cc} is not removed by Step 1(b) because L_{cc} cannot contain more than $(2^{\lfloor i/2^{j-1} \rfloor} - 1)^2$ edges. In addition, L_{cc} contains all the light primary external edges of T. In Lemma 5.6 below, we show that $tmp\text{-}h(L_{cc})$ is the weight of a heavy internal or external edge of T. Thus, L_{cc} contains edges with weight less than $tmp\text{-}h(L_{cc})$ and cannot be removed by Step 2(c).

LEMMA 5.6. *$tmp\text{-}h(L_{cc})$ is equal to the weight of a heavy internal or external edge of T.*

Proof. Among all the lists in $\mathcal{Q}_{j-1}$ that are merged into L_{cc}, let L be the one with the smallest threshold. That is, $tmp\text{-}h(L_{cc}) = h(L)$. Let W denote the tree in F_{j-1} represented by L. By R4, $h(L)$ is equal to the weight of a heavy internal or external edge e of W. Thus e is also a heavy internal or external edge of T. Lemma 5.6 follows. ■

5.3 Updating Threshold and Retaining only External Edges

After Procedure `Ext_M&C` is executed, every remaining merged list is representing the core-cluster of a tree in F_j. Let L_{cc} be such a list representing a tree $T \in F_j$. If T contains less than 2^i vertices, all light primary external edges of T appear among the $2^{\lfloor i/2^j \rfloor} - 1$ smallest edges in L_{cc}, and all other edges in L_{cc} are heavy edges. If T has at least 2^i vertices, no external or internal edges of T are light and all edges in L_{cc} must be heavy. By the definition of `Ext_M&C`, the number of edges in L_{cc} is at most $(2^{\lfloor i/2^{j-1} \rfloor} - 1)^2$, but may exceed the length requirement for Phase j (i.e., $2^{\lfloor i/2^j \rfloor} - 1$). To ensure that L_{cc} satisfies R2 and R3, we retain only the $2^{\lfloor i/2^j \rfloor} - 1$ smallest edges on L_{cc}. The threshold of L_{cc}, denoted by $h(L_{cc})$, is updated to be the minimum of $tmp\text{-}h(L_{cc})$ and the weight of the smallest edge truncated.

No matter whether T contains fewer than 2^i vertices or not, every edge truncated from L_{cc} is heavy. Together with Lemma 5.6, we can conclude that $h(L_{cc})$ is equal to the weight of a heavy internal or external edge of T, satisfying R4.

Next, we give an observation on L_{cc} and in Lemma 5.8, we show that R5 is satisfied. Denote Z_0 as the core-cluster of T represented by L_{cc}.

LEMMA 5.7. *Let e be an external edge of Z_0. If e is a tree edge of T, then e is not included in L_{cc} and $h(L_{cc}) \leq w(e)$.*

Proof. Suppose e is included in L_{cc}. Note that e cannot be a full edge with respect to $\mathcal{Q}_{j-1}$ because a full edge and its mate should have been removed in Step 2(b) in Procedure `Ext_M&C`. Then e is in *Half-INPUT* and Procedure `Ext_M&C` should have removed L_{cc} at Step 2(a). This contradicts that L_{cc} is one of the remaining lists after Procedure `Ext_M&C` is executed. Therefore, e is not included in L_{cc}.

Next, we show that $h(L_{cc}) \leq w(e)$. Let W be the subtree of Z_0 such that e is an external edge of W. Since e is a tree edge, e is a primary external edge of W. As e is not included in L_{cc} and L_{cc} inherits the adjacency list $L_W \in \mathcal{Q}_{j-1}$ representing W, e is also not included in L_W. By R5, $h(L_W) \leq w(e)$. Recall that $h(L_{cc}) \leq tmp\text{-}h(L_{cc}) \leq h(L_W)$. Therefore $h(L_{cc}) \leq w(e)$. ■

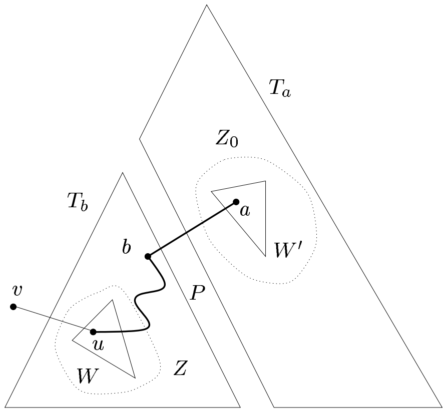

Figure 7: Z_0 and Z is connected by a path P in T, and P contains an edge (a, b), which is an external edge of both Z_0 and W'.

Lemma 5.8. *Let e be an external edge of T currently not found in L_{cc}. If (i) e is primary, or (ii) e is secondary and the mate of e is still included in some other list L' in $\mathcal{Q}_j$, then $h(L_{cc}) \leq w(e)$.*

Proof. Let $e = \langle u, v\rangle$ be an external edge of T currently not found in L_{cc}, satisfying the conditions stated in Lemma 5.8. Let W be the tree in F_{j-1} such that W is a subtree of T and e is an external edge of W. With respect to W, either e is primary, or e is secondary and the mate of e is included in another list in $\mathcal{Q}_{j-1}$. We consider whether W is included in the core cluster Z_0 of T.

Case 1. W is a subtree of Z_0. By definition of Z_0, W must be represented by a list $L_W \in \mathcal{Q}_{j-1}$. At the end of Phase $j-1$, e may or may not appear in L_W. If e does not appear in L_W, then, by R5, $h(L_W) \leq w(e)$. Since $h(L_{cc}) \leq \textit{tmp-}h(L_{cc}) \leq h(L_W)$, we have $h(L_{cc}) \leq w(e)$. Suppose that e is in L_W. Then e is passed to L_{cc} when Procedure `Ext_M&C` starts off. Yet e is currently not in L_{cc}. If e is removed from L_{cc} within Procedure `Ext_M&C`, this has to take place at Step 2(b), and e is either an internal edge of T or a secondary external edge removed together with its mate. This contradicts the assumption about e. Thus, e is removed after Procedure `Ext_M&C`, i.e., due to truncation. In this case, the way $h(L_{cc})$ is updated guarantees $h(L_{cc}) \leq w(e)$.

Case 2. W is a subtree of a non-core cluster Z. We show that Z_0 has an external edge $e' = \langle a, b\rangle$ such that $h(L_{cc}) \leq w(a, b)$ and $w(a, b) < w(e)$. Observe that T contains a path connecting Z_0 and Z, and this path must involve an external edge $\langle a, b\rangle$ of Z_0. By Lemma 5.7, $h(L_{cc}) \leq w(a, b)$.

Next we show that $w(a, b) < w(e)$. Let W' be the tree in F_{j-1} such that W' is a subtree of Z_0 and $\langle a, b\rangle$ is an external edge of W'. See Figure 7. Suppose we remove the edge (a, b) from T, T is partitioned into two subtrees T_a and T_b, containing the vertices a and b, respectively. Note that T_b contains W, and e is an external edge of T_b. On the other hand, Z_0 is included in T_a, and e_T is not an external edge of T_b. By Lemma 2.3, the minimum external edge of T_b is $\langle b, a\rangle$. Therefore, $w(a, b) < w(e)$. As a result, $h(L_{cc}) \leq w(a, b) < w(e)$ and the lemma follows. ■

Removing remaining internal edges: Note that L_{cc} may still contain some

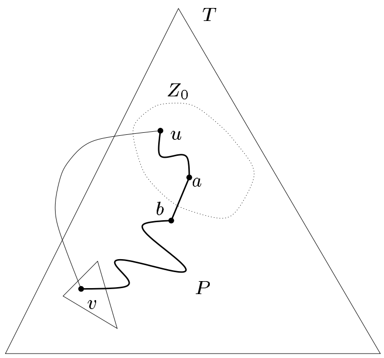

Figure 8: The pair of vertices u and v of e is connected by a path P in T. Every edge on P has weight smaller than $w(e)$.

internal edges of T. This is because Procedure `Ext_M&C` only remove those internal edges whose mates also appear in L_{cc}. The following lemma shows that every remaining internal edges in L_{cc} has a weight greater than $h(L_{cc})$. Thus by discarding those edges in L_{cc} with weight greater than $h(L_{cc})$, we ensure that only external edges of T are retained. Of course, no light primary external edge can be removed by this step.

LEMMA 5.9. *For any internal edge e of T that is currently included in L_{cc}, $h(L_{cc}) \leq w(e)$.*

Proof. We consider whether $e = \langle u, v \rangle$ is an internal or external edge of Z_0.

Case 1. e is an internal edge of Z_0. Suppose L_{cc} inherits e from the list $L_W \in \mathcal{Q}_{j-1}$, where L_W represents a tree $W \in F_{j-1}$ and W is a subtree of Z_0. By R1, $\langle u, v \rangle$ is an external edge of W. Then Z_0 includes another tree $W' \in F_{j-1}$ which contains the vertex v. Denote $L_{W'}$ as the list in $\mathcal{Q}_{j-1}$ that represents W'. The edge $\langle v, u \rangle$ is an external edge of W'. But $\langle v, u \rangle$ does not appear in $L_{W'}$ (otherwise, L_{cc} would have also inherited $\langle v, u \rangle$ from $L_{W'}$ and Procedure `Ext_M&C` should have removed both $\langle u, v \rangle$ and $\langle v, u \rangle$ from L_{cc} at Step 2(b)). By R5, $h(L_{W'}) \leq w(u, v)$. As $h(L_{cc}) \leq$ *tmp*-$h(L_{cc}) \leq h(L_{W'})$, we have $h(L_{cc}) \leq w(u, v)$.

Case 2. e is an external edge of Z_0. By Lemma 5.7, e is not a tree edge of T. Let P be a path on T connecting u and v. See Figure 8. Since T is a subtree of T^*_G, every edge on P has weight smaller than $w(u, v)$. On P, we can find an external edge $\langle a, b \rangle$ of Z_0. By Lemma 5.7 again, $h(L_{cc}) \leq w(a, b)$ and hence $h(L_{cc}) \leq w(u, v)$. ■

5.4 The complete algorithm

The discussion of Thread i in the previous three sections is summarized in the following procedure. The time and processor requirement will be analyzed in the next section.

Thread i

Input: G; B_k, where $1 \leq k \leq i-1$, is available at the end of the kth superstep
Output: B_i

◇ // **Phase 0** *(Initialization)*
Construct $\mathcal{Q}_0$ from G; $a_0 \leftarrow 0$

◇ For $j = 1$ to $\lfloor \log i \rfloor$ do // **Phase** j
// denote *INPUT* as $B[a_{j-1}+1, a_j]$, where $a_j = i - \lfloor i/2^j \rfloor$.

1. (a) Edges in *INPUT* that are full with respect to $\mathcal{Q}_{j-1}$ merge their lists in $\mathcal{Q}_{j-1}$. Let $\mathcal{Q}$ be the set of merged adjacency lists.
   (b) For each list $\mathcal{L} \in \mathcal{Q}$, if $\mathcal{L}$ contains at most $(2^{\lfloor i/2^{j-1} \rfloor} - 1)^2$ edges, $tmp\text{-}h(\mathcal{L}) \leftarrow \min\{h(\mathcal{L}') \mid \mathcal{L}' \in \mathcal{Q}_{j-1}$ and $\mathcal{L}'$ is a part of $\mathcal{L}\}$; otherwise, remove $\mathcal{L}$ from $\mathcal{Q}$.
2. For each list $\mathcal{L} \in \mathcal{Q}$, // *Remove unwanted edges and lists*
   (a) if $\mathcal{L}$ contains an edge in *Half-INPUT*, remove $\mathcal{L}$ from $\mathcal{Q}$;
   (b) detect and remove internal and secondary external edges from $\mathcal{L}$;
   (c) if, for all edges $\langle u, v \rangle$ in $\mathcal{L}$, $w(u,v) \geq tmp\text{-}h(\mathcal{L})$, remove $\mathcal{L}$ from $\mathcal{Q}$.
3. // *Truncate each list if necessary and remove remaining internal edges*
   (a) For each list $\mathcal{L} \in \mathcal{Q}$, if $\mathcal{L}$ contains more than $2^{\lfloor i/2^j \rfloor} - 1$ edges, retain the $2^{\lfloor i/2^j \rfloor} - 1$ smallest ones and update $h(\mathcal{L})$ to the minimum of $tmp\text{-}h(\mathcal{L})$ and $w(e_o)$, where e_o is the smallest edge just removed from $\mathcal{L}$.
   (b) For each edge $\langle u, v \rangle \in \mathcal{L}$, if $w(u,v) \geq h(\mathcal{L})$, remove $\langle u, v \rangle$ from $\mathcal{L}$.
4. $\mathcal{Q}_j \leftarrow \mathcal{Q}$.

◇ $B_i \leftarrow \{(u,v) \mid \langle u,v \rangle$ or $\langle v,u \rangle$ appears in some list $\mathcal{L}$ in $\mathcal{Q}_{\lfloor \log i \rfloor}$; $w(u,v) < h(\mathcal{L})\ \}$.

6 Time and processor complexity

First, we show that the new MST algorithm runs in $O(\log n)$ time using $(n+m)\log n$ CREW PRAM processors. Then we illustrate how to modify the algorithm to run on the EREW PRAM and reduce the processor bound to linear.

Before the threads start to run concurrently, they need an initialization step. First, each adjacency list of G is sorted in ascending order with respect to the edge weights. This set of sorted adjacency lists is replicated $\lfloor \log n \rfloor$ times and each copy is moved to the "local memory" of a thread, which is part of the global shared memory dedicated to the processors performing the "local" computation of a thread. The replication takes $O(\log n)$ time using a linear number of processors. Then each thread constructs its own $\mathcal{Q}_0$ in $O(1)$ time. Afterwards, the threads run concurrently.

As mentioned in Section 3, the computation of a thread is scheduled to run in a number of phases. Each phase starts and ends at predetermined supersteps. We need to show that the computation of each phase can be completed within the allocated time interval. In particular, Phase j of Thread i is scheduled to start at the (a_j+1)th superstep and end at the a_{j+1}th superstep using $\lfloor i/2^j \rfloor - \lfloor i/2^{j+1} \rfloor = \left\lceil \frac{1}{2} \lfloor i/2^j \rfloor \right\rceil$ supersteps. The following lemma shows that Phase j of Thread i can be implemented in $c(i/2^{j-1})$ time, where c is a constant. By setting the length of a superstep to a constant c' such that $c(i/2^{j-1})/c' \leq \left\lceil \frac{1}{2} \lfloor i/2^j \rfloor \right\rceil$, Phase j can complete its computation in at most $\left\lceil \frac{1}{2} \lfloor i/2^j \rfloor \right\rceil$ supersteps. It can be verify that $c' \geq 8c$ satisfies this condition.

LEMMA 6.1. *Phase j of Thread i can be implemented in $O(i/2^{j-1})$ time using $n+m$ CREW PRAM processors.*

Proof. Consider the computation of Phase j of Thread i. Before the merging of the adjacency list starts, Thread i reads in $B[a_{j-1}+1, a_j]$, which may also be read by many other threads, into the local memory of Thread i. The merging of adjacency lists in Step 2(a) takes $O(1)$ time. In Step 2(b), testing the length of a list ($\leq (2^{\lfloor i/2^{j-1} \rfloor}-1)^2$) can be done by performing pointer jumping in $O(\log(2^{\lfloor i/2^{j-1} \rfloor}-1)^2) = O(i/2^{j-1})$ time. After that, all adjacency lists left have length at most $(2^{\lfloor i/2^{j-1} \rfloor}-1)^2$. In the subsequent steps, we make use of standard parallel algorithmic techniques including list ranking, sorting, and pointer jumping to process each remaining list. The time used by these techniques is the logarithmic order of the length of each list (see e.g., JáJá 1992). Therefore, all the steps of Phase j can be implemented in $c(i/2^{j-1})$ time, for some constant c, using a linear number of processors. ■

COROLLARY 6.1. *The minimum spanning tree of a weighted undirected graph can be found in $O(\log n)$ time using $(n+m)\log n$ CREW PRAM processors.*

Proof. By Lemma 6.1, the computation of Phase j of Thread i satisfies the predetermined schedule. Therefore, B_i can be found at the end of the ith superstep and $B[1, \lfloor \log n \rfloor]$ are all ready at the end of the $\lfloor \log n \rfloor$th superstep. That means the whole algorithm runs in $O(\log n)$ time. As Thread i uses at most $n+m$ processors, $(n+m)\log n$ processors suffice for the whole algorithm. ■

6.1 Adaptation to EREW PRAM

We illustrate how to modify the algorithm to run on the EREW PRAM model. Consider Phase j of Thread i. As discussed in the proof of Lemma 6.1, concurrent read is used only in accessing the edges of $B[a_{j-1}+1, a_j]$, which may also be read by many other threads at the same time. If $B[a_{j-1}+1, a_j]$ have already resided in the local memory of Thread i, all steps can be implemented on the EREW PRAM.

To avoid using concurrent read, we require each thread to copy its output to each subsequent thread. By modifying the schedule, each thread can perform this copying process in a sequential manner. Details are as follows: As shown in the proof of Lemma 6.1, Phase j of Thread i can be implemented in $c(i/2^{j-1})$ time, where c is a constant. The length of a superstep was set to be c' so that Phase j of Thread i can be completed within $\left\lceil \frac{1}{2} \lfloor i/2^j \rfloor \right\rceil$ supersteps. Now the length of each superstep is doubled (i.e., each superstep takes $2c'$ time instead of c'). Then the computation of Phase j can be deferred to the last half supersteps (i.e., the last $\left\lceil \frac{1}{4} \lfloor i/2^j \rfloor \right\rceil$ supersteps). In the first half supersteps of Phase j (i.e., from the $(a_{j+1}+1)$th to $(a_{j+1}+\left\lfloor \frac{1}{4} \lfloor i/2^j \rfloor \right\rfloor)$th supersteps), no computation is performed. Thread i is waiting for other threads to store the outputs $B_{a_{j-1}+1}, \cdots, B_{a_j}$ into the local memory.

To complete the schedule, we need to show how each Thread k, where $k < i$, perform the copying in time. Recall that Thread k completes its computation at the kth superstep. In the $(k+t)$th superstep, where $t \geq 0$, Thread k copies B_k to four threads, namely Thread $(k+4t+1)$ to Thread $(k+4t+4)$. Each replication takes $O(1)$ time using a linear number of processors.

LEMMA 6.2. *Consider any Thread i. At the end of the $(a_j + \left\lfloor \frac{1}{4} \lfloor i/2^j \rfloor \right\rfloor)$th superstep, there is a copy of $B[a_{j-1}+1, a_j]$ residing in the local memory of Thread i.*

Proof. For $k < i$, Thread i receives B_k at the $(k + \lfloor (i-k)/4 \rfloor)$th superstep. For Phase j of Thread i, B_{a_j} is the last set of edges to be received and it arrives at the $(a_j + \lfloor (i-a_j)/4 \rfloor) = (a_j + \left\lfloor \frac{1}{4} \lfloor i/2^j \rfloor \right\rfloor)$th superstep, just before the start of the second half of Phase j. ■

6.2 Linear processors

In this section we further adapt our MST algorithm to run on a linear number of processors. We first show how to reduce the processor requirement to $m + n \log n$. Then, for a dense graph with at least $n \log n$ edges, the processor requirement is dominated by m. Finally, we give a simple extra step to handle sparse graphs.

To reduce the processor requirement to $m + n \log n$, we would like to introduce some preprocessing to each thread so that each thread can work on only n (instead of m) edges to compute the required output using n processors. Yet the preprocessing of each thread still needs to handle m edges and requires m processors. To sidestep this difficulty, we attempt to share the preprocessing among the threads. Precisely, the computation is divided into $\lceil \log \log n \rceil + 1$ stages. In Stage k, where $1 \le k \le \lceil \operatorname{loglog} n \rceil$, we perform one single preprocessing, which then allows up to 2^{k-1} threads to compute concurrently the edge sets $B_{2^{k-1}+1}, \cdots, B_{2^k}$ in 2^{k-1} supersteps using $2^{k-1} \cdot n$ processors. The preprocessing itself runs in $O(2^k)$ supersteps using $m + n$ processors. Thus, each stage makes use of at most $m + n \log n$ processors, and the total number of supersteps over all stages is still $O(\log n)$.

LEMMA 6.3. *The minimum spanning tree of a weighted undirected graph can be found in $O(\log n)$ time using $m + n \log n$ processors on the EREW PRAM.*

Proof. The linear-processor algorithm runs in $\lceil \operatorname{loglog} n \rceil + 1$ stages. In Stage 0, B_1 is found by Thread 1. For $1 \le k \le \lceil \operatorname{loglog} n \rceil$, Stage k is given $B[1, 2^{k-1}]$ and is to compute $B[2^{k-1}+1, 2^k]$. Specifically, let $x = 2^{k-1}$, Stage k involves Thread $2x$ for the preprocessing and Threads $1, 2, \cdots, x$ for the actual computation of $B_{x+1}, B_{x+2}, \cdots, B_{2x}$. Both parts require $O(x)$ supersteps.

The preprocessing is to prepare the initial adjacency lists for each thread. Let F be the set of trees induced by $B[1, x]$, which is, by definition, a 2^x-forest of G. We invoke Thread $2x$ to execute Phase 1 only, computing a set $\mathcal{Q}_1$ of adjacency lists. By definition, each list in $\mathcal{Q}_1$ has length at most $2^{2x/2} - 1 = 2^x - 1$, representing a tree T in F and containing all primary external edges of T with base less than 2^{2x}. $\mathcal{Q}_1$ contains sufficient edges for finding not only B_{2x} but also $B_{x+1}, \cdots, B_{2x-1}$. As F contains at most $n/2^x$ trees, $\mathcal{Q}_1$ contains a total of at most n edges.

Each list in $\mathcal{Q}_1$ is sorted with respect to the edge weight using $O(x)$ supersteps and n processors. Then $\mathcal{Q}_1$ is copied into the local memory of Threads 1 to x one by one in x supersteps using n processors. For $1 \le i \le x$, Thread i replaces its initial set of adjacency lists $\mathcal{Q}_0$ with a new set $\mathcal{Q}_0^{(i)}$, which is constructed by truncating each list in $\mathcal{Q}_1$ to include the smallest $2^i - 1$ edges.

Threads 1 to x are now ready to run concurrently, computing $B_{x+1}, \cdots, B_{2x}$, respectively. For all $1 \leq i \leq x$, Thread i uses its own $\mathcal{Q}_0^{(i)}$ as the initial set of adjacency lists and follows its original phase-by-phase schedule to execute the algorithm stated in Section 5.4. Note that the algorithm of a thread is more versatile than was stated. When every Thread i starts with $\mathcal{Q}_0^{(i)}$ as input, Thread i will compute the edge set B_{x+i} (instead of B_i) in i supersteps. That is, $B_{x+1}, \cdots, B_{2x}$ can be found by Threads 1 to x in x supersteps. Note that $\mathcal{Q}_0^{(i)}$ has at most n edges, the processors requirement of each thread is n only.

In short, Stage k takes $O(x)$ supersteps using $m+x\cdot n \leq m+n\log n$ processors. Recall that $x = 2^{k-1}$. The $\lceil \log\log n \rceil$ stages altogether run in $O(\log n)$ time using $m + n\log n$ processors. ■

If the input graph is sparse, i.e., $m < n\log n$, we first construct a contracted graph G_c of G as follows. We execute Threads 1 to $\mathrm{loglog}\, n$ concurrently to find $B[1, \log n]$, which induces a $(\log n)$-forest B of G. Then, by contracting each tree in the forest, we obtain a contracted graph G_c with at most $n/\log n$ vertices. The contraction takes $O(\log n)$ time using $m+n$ processors. By Lemma 6.3, the minimum spanning tree of G_c, denoted $T^*_{G_c}$, can be computed in $O(\log n)$ time using $m + (n/\log n)\log n = m + n$ processors. Note that $T^*_{G_c}$ and B include exactly all the edges in T^*_G. We conclude with the following theorem.

THEOREM 6.1. *The minimum spanning tree of an undirected graph can be found in $O(\log n)$ time using a linear number of processors on the EREW PRAM.*

References

Armoni, R., Ta-Shma, A., Wigderson, A., and Zhou., S. (1997). $SL \subseteq L^{4/3}$. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 230–239.

Awerbuch, B. and Shiloach, Y. (1987). New Connectivity and MSF Algorithms for Shuffle-exchange Network and PRAM. *IEEE Trans. Comput.*, 36:1258–1263.

Chazelle, B. (1997). A Faster Deterministic Algorithm for Minimum Spanning Trees. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31.

Chazelle, B. (1999). A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. Technical Report NECI Technical Report 99-099.

Chin, F. Y., Lam, J., and Chen, I.-N. (1982). Efficient Parallel Algorithms for some Graph Problems. *Comm. ACM*, 25:659–665.

Chong, K. W. (1996). Finding Minimum Spanning Trees on the EREW PRAM. In *Proceedings of the International Computer Symposium*, pages 7–14, Taiwan.

Chong, K. W. and Lam, T. W. (1993). Finding Connected Components in $O(\log n \,\mathrm{loglog}\, n)$ time on the EREW PRAM. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–20.

Cole, R. (1988). Parallel Merge Sort. *SIAM Journal on Computing*, 17:770–785.

Cole, R., Klein, P. N., and Tarjan, R. E. (1996). Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proceedings of the 8th Annual ACM Symposium on Parallel Architectures and Algorithms*, pages 243–250.

Cole, R. and Vishkin, U. (1986). Approximate and Exact Parallel Scheduling with Applications to List, Tree, and Graph Problems. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–491.

Cook, S. A., Dwork, C., and Reischuk, R. (1986). Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15(1):87–97.

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their used in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615.

Gabow, H., Galil, Z., Spencer, T., and Tarjan, R. E. (1986). Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:2:109–122.

Gazit, H. (1986). An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 492–501.

Gibbons, P. B., Matias, Y., and Ramachandran, V. (1997). Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 72–83.

Halperin, S. and Zwick, U. (1996). Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 438–447.

Han, Y. and Wagner, R. A. (1990). A fast and efficient parallel connected component algorithm. *J. of ACM*, 37(3):626–642.

Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. (1979). Computing Connected Components on Parallel Computers. *Comm. ACM*, 22:461–464.

JáJá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley.

Johnson, D. B. and Metaxas, P. (1991). Connected Components in $O(\lg^{3/2}|V|)$ Parallel Time for the CREW PRAM. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 688–697.

Johnson, D. B. and Metaxas, P. (1992). A Parallel Algorithm for Computing Minimum Spanning Trees. In *Proceedings of the 4th Annual ACM Symposium on Parallel Architectures and Algorithms*, pages 363–372.

Karger, D. R. (1995). *Random sampling in Graph Optimization Problems*. PhD thesis, Department of Computer Science, Stanford University.

Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328.

Karger, D. R., Nisan, N., and Parnas, M. (1992). Fast Connected Components Algorithms for the EREW PRAM. In *Proceedings of the 4th Annual ACM Symposium on Parallel Architectures and Algorithms*, pages 373–381.

Karp, R. M. and Ramachandran, V. (1990). Parallel Algorithms for Shared-Memory Machines. In van Leeuwen Ed, J., editor, *Handbook of Theoretical Computer Science vol A.*, pages 869–941. MIT Press, Massachusetts.

Maon, Y., Schieber, B., and Vishkin, U. (1986). Parallel Ear Decomposition Search (EDS) and s-t numbering in graphs. *Theoretical Computer Science*, 47:277–298.

Miller, G. L. and Ramachandran, V. (1986). Efficient Parallel Ear Decomposition with Applications. In *Manuscript*.

Nisan, N., Szemeredi, E., and Wigderson, A. (1992). Undirected Connectivity in $O(\log^{1.5} n)$ Space. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 24–29.

Pettie, S. (1999). Finding Minimum Spanning Trees in $O(n\alpha(m,n))$ Time. Technical Report UTCS Technical Report TR99-23, Department of Computer Sciences, The University of Texas at Austin.

Pettie, S. and Ramachandran, V. (1999). A Randomized Time-Work Optimal Parallel Algorithm for Finding a Minimum Spanning Forest. In *Proceedings of the 3rd International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 233–244.

Pettie, S. and Ramachandran, V. (2000). An Optimal Minimum Spanning Tree Algorithm. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*.

Poon, C. K. and Ramachandran, V. (1997). A Randomized Linear Work EREW PRAM Algorithm to Find a Minimum Spanning Forest. In *Proceedings of the 8th Annual International Symposium on Algorithms and Computation*, pages 212–222.

Tarjan, R. E. (1983). *Structures and Network Algorithms*. SIAM, Philadelphia, Pa.

Tarjan, R. E. and Vishkin, U. (1985). An Efficient Parallel Biconnectivity Algorithm. *SIAM J. Comput.*, 14:862–874.

Valiant, L. G. (1990). A Bridging Model For Parallel Computation. *Communications of the ACM*, 33(8):103–111.

Vishkin, U. (1985). On efficient parallel strong orientation. *Information Processing Letters*, 20:235–240.

Appendix

We prove the following claim and lemma that capture some interesting properties of the trees induced by the edge sets $B_1, B_2, \cdots, B_{\lfloor \log n \rfloor}$.

Claim. *Let T be any one of the trees induced by $B[1,k]$, for any $0 \le k \le \lfloor \log n \rfloor$. Let (a,p) and (b,q) be two external edges of T, where $a, b \in T$. For each edge on the path connecting a and b in T, its weight is smaller than* $\max\{w(a,p), w(b,q)\}$.

Proof. We prove the lemma by induction on k. The base case, $k = 0$, is true as every tree contains only one vertex. Thus $a = b$ and the path involves no edge.

Inductive case, $k \ge 1$: Let P be the path in T connecting a and b. Let $X = \{W_1, W_2, \cdots, W_l\}$ be the subset of trees induced by $B[1, k-1]$ such that each of them involves at least one vertex of P. Without loss of generality, we can assume that $a \in W_1, b \in W_l$ and for $1 \le i \le l-1$, W_i and W_{i+1} are connected by an edge (v_i, u_{i+1}), where $v_i \in W_i$ and $u_{i+1} \in W_{i+1}$. By the construction of B_k, (v_i, u_{i+1}) is the minimum external edge of W_i or that of W_{i+1} (or both). Let P_i be the "sub-path" of P in W_i, i.e., $P_i = P \cap W_i$. Let $u_1 = a$ and $v_l = b$. Then P_i is a path in W_i connecting u_i and v_i. We can partition P into l smaller paths and $l-1$ edges as follows: $\langle P_1, (v_1, u_2), P_2, (v_2, u_3), \cdots, (v_{l-1}, u_l), P_l \rangle$.

We are going to prove that for $1 \le i \le l-1$, $w(v_i, u_{i+1}) < \max\{w(a,p), w(b,q)\}$. Then, by the induction hypothesis, we can show that every edge on P_i also has weight smaller than $\max\{w(a,p), w(b,q)\}$.

- **Edges connecting W_i and W_{i+1}:** Recall that (v_i, u_{i+1}) is the minimum external edge of either W_i or W_{i+1}. We can find a level t for which a tree $W_t \in X$ has the minimum external edge with the following property: Either it is not in P or it is also the minimum external edge of W_{t-1}. Then for $1 \le i \le t-1$, the minimum external edge of W_i is (v_i, u_{i+1}); for $t+1 \le i \le l$, the minimum external edge of W_i is (v_{i-1}, u_i). As a result, $w(v_1, u_2) > w(v_2, u_3) > \cdots > w(v_{t-1}, u_t)$ and $w(v_t, u_{t+1}) < w(v_{t+1}, u_{t+2}) < \cdots < w(v_{l-1}, u_l)$. Since $w(a,p) > w(v_1, u_2)$ and $w(v_{l-1}, u_l) < w(b,q)$, it follows that $w(v_i, u_{i+1}) < \max\{w(a,p), w(b,q)\}$ for $1 \le i \le l-1$.

- **Edges on P_i:** Consider the path P_i in W_i which connects u_i and v_i. By the induction hypothesis, every edge on P_i has weight smaller than $\max\{w(v_{i-1}, u_i), w(v_i, u_{i+1})\}$ (or $\max\{w(p,a), w(v_1, u_2)\}$ if $i = 1$, $\max\{w(v_{l-1}, u_l), w(b,q)\}$ if $i = l$). As shown above, both $w(v_{i-1}, u_i)$ and $w(v_i, u_{i+1})$ are smaller than $\max\{w(a,p), w(b,q)\}$. Therefore, every edge on P_i has weight smaller than $\max\{w(a,p), w(b,q)\}$.

As a result, every edge on P has weight smaller than $\max\{w(a,p), w(b,q)\}$. ■

Lemma 2.3. *Let T be any one of the trees induced by $B[1,k]$, for any $0 \le k \le \lfloor \log n \rfloor$. Let e_T be the minimum external edge of T. For any subtree (i.e., connected subgraph) S of T, the minimum external edge of S is either e_T or an edge of T.*

Proof. Let e be the minimum external edge of S. Assume to the contrary that e is not an edge of T and $e \neq e_T$. That means e is an external edge of T and $w(e) > w(e_T)$. Then e_T cannot be an external edge of S.

Let $e_T = \langle u, v\rangle$ and $e = \langle x, y\rangle$. Consider the path P in T connecting u and x. Since u does not belong to S, we can find an edge e' on P that is an external edge of S (and of course an edge of T). By the above claim, every edge on P has weight smaller than $\max\{w(e), w(e_T)\} = w(e)$. Thus e' has weight smaller than $w(e)$. Therefore, e cannot be the minimum external edge of S. We obtain a contradiction. ■