SIAM J. COMPUT.
Vol. 35, No. 1, pp. 120–131

MINIMUM-WEIGHT SPANNING TREE CONSTRUCTION IN $O(\log \log n)$ COMMUNICATION ROUNDS*

ZVI LOTKER†, BOAZ PATT-SHAMIR†, ELAN PAVLOV‡, AND DAVID PELEG§

Abstract. We consider a simple model for overlay networks, where all n processes are connected to all other processes, and each message contains at most $O(\log n)$ bits. For this model, we present a distributed algorithm which constructs a minimum-weight spanning tree in $O(\log \log n)$ communication rounds, where in each round any process can send a message to every other process. If message size is $\Theta(n^\epsilon)$ for some $\epsilon > 0$, then the number of communication rounds is $O(\log \frac{1}{\epsilon})$.

Key words. minimum-weight spanning tree, distributed algorithms

AMS subject classifications. 05C05, 05C85, 68Q22, 68Q25, 68R10

DOI. 10.1137/S0097539704441848

1. Introduction.

A minimum-weight spanning tree (MST) is one of the most useful distributed constructs, as it minimizes the cost associated with global operations such as broadcasts and convergecasts. This paper presents an MST construction algorithm that works in $O(\log \log n)$ communication rounds, where in each round each process can send $O(\log n)$ bits to every other process (intuitively allowing each message to contain the identity and weight of only a constant number of edges). Our result shows that an MST can be constructed with little communication: throughout the execution of the algorithm, each pair of processes exchanges at most $O(\log n \log \log n)$ bits; the overall number of bits sent is $\Theta(n^2 \log n)$, which is optimal. The algorithm extends to larger message sizes, in the sense that the number of communication rounds is $O(\log \frac{1}{\epsilon})$ if each message can contain n^ϵ bits for some $\epsilon > 0$. Note that if messages are not restricted in size, then the MST can be trivially constructed in a single round of communication: each process sends all its information to all its neighbors, allowing each node to locally compute the MST.

The number of communication rounds dominates the time complexity in situations where latency is high and bandwidth is scarce. This may be the situation in some *overlay networks*. Briefly, the idea in overlay networks is to think of the underlying communication network (e.g., the Internet) as a "black box" that provides reliable point-to-point communication. On top of that network run distributed applications. This approach (whose precursor is the Internet's "end-to-end argument" [13]) is different from classical distributed models, where processes reside in networks nodes (i.e., switches or routers), and thus their implementation would require using low-level communication. Rather, the pragmatic view now is that distributed applica-

*Received by the editors February 11, 2004; accepted for publication (in revised form) April 27, 2005; published electronically September 8, 2005. A preliminary version of this work appeared in *Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, CA, 2003.

http://www.siam.org/journals/sicomp/35-1/44184.html

†Department of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, Israel (zvilo@eng.tau.ac.il, boaz@eng.tau.ac.il).

‡Department of Computer Science, The Hebrew University, Jerusalem 91904, Israel (elan@cs.huji.ac.il).

§Department of Computer Science and Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel (david.peleg@weizmann.ac.il). The work of this author was supported in part by a grant from the Israel Science Foundation.

tions create their own overlay network by choosing which pairs of local processes will communicate directly according to various criteria. The concept of overlay networks is central to areas such as multicast or content distribution networks (see, e.g., [8] and the references therein), peer-to-peer systems (for example, Chord [14]), and others.

1.1. Related work. Spanning tree construction is well studied as a sequential optimization problem (see, e.g., [15, 9]). Distributed MST constructions are presented in [6, 3] (and see the references in [11]). These classical distributed algorithms are oriented towards minimizing the total number of messages in general networks, and their time complexity is inherently $\Omega(\log n)$, even when run on fully connected graphs. The model we use in this paper is a special case of the model studied in [7, 12, 10]: in these papers, each message has $O(\log n)$ bits, but the fully connected graph is not directly considered. The best previously known upper bound for fully connected graphs in this model is $O(\log n)$ communication rounds. This bound holds also for graphs of diameter 2 [10]. (It is known that the number of rounds jumps at least to $\Omega(n^{1/4})$ when the diameter of the network is 3 or more [10, 12].)

The parallel time complexity of MST construction depends on the particular architecture considered, but we are not aware of any sublogarithmic time algorithm that uses small messages. For the PRAM model, there are quite a few $O(\log n)$ algorithms, including a deterministic one for the CRCW model [4] and a randomized one for the EREW model [5]. Adler et al. [1] study the total number of bits that must be communicated in the course of an MST construction problem under various parallel architectures. For our model, their results imply that the worst-case number of bits that need to be communicated throughout the execution of the algorithm is $\Omega(n^2 \log n)$.

1.2. System model. In the underlying formal model, the system is represented by a complete n-node weighted undirected graph $G = (V, E, \omega)$, where $\omega(e)$ denotes the weight of edge $e \in E$. Each node has a distinct ID of $O(\log n)$ bits. Each node knows all the edges incident to it (and hence, since the graph is a clique, each node knows about all other nodes in the system). An execution of the system proceeds in asynchronous steps: in a "receive" step, a node receives some of the messages sent to it in previous steps. In a "send" step, a node makes a local computation and sends messages to the other nodes in the system. Each message may be different, and we require that each message contains at most $O(\log n)$ bits. (The results are extended to larger message sizes in section 4.) We assume that messages may be delayed arbitrarily but are never lost or corrupted. The time complexity of an algorithm in the asynchronous model is measured by normalizing the scale so that the longest message delivery time is one unit.

Simplification: The synchronous model. In the synchronous model, computation advances in global rounds, where in each round processes send messages, receive them, and do some local computation. This model is much more convenient as a programming mode. Fortunately, since we assume that the system is reliable, we may apply a *synchronizer* that allows us to present the algorithm in the synchronous model. Specifically, we use the α synchronizer of Awerbuch [2]. Let us outline the idea briefly. Assume that we have an algorithm SA for the synchronous model. The execution in the asynchronous model is done as follows. A process starts the next round only after receiving a special "proceed" message from a distinguished node v^* (say, the node with the lowest ID in the system). It then sends messages according to SA. For each SA message received, the receiver node sends an "ack" message back to the sender; when a sender has received acknowledgements to all the messages it

sent, the sender forwards a "safe" message to v^*; when v^* receives "safe" messages from all nodes in the system, it sends a "proceed" message to all other nodes, which may then send their SA messages of the next round. Note that since we assume that the graph is fully connected, this transformation incurs only a constant blowup in the message complexity and in time complexity. We shall henceforth use the synchronous model, but we emphasize that the algorithm works in the asynchronous model using the simple synchronizer described above.

1.3. The MST construction problem. We assume that in the initial state, the input to each node $v \in V$ consists of the weights of all its incident edges $\omega(v,u)$ for all $u \in V \setminus \{v\}$. Edge weights are assumed to be integers that can be represented using $O(\log n)$ bits. Without loss of generality, we assume that all the edge weights are distinct (otherwise we can break ties by node IDs), and hence the MST is unique. When our algorithm halts, all nodes know the full list of all $n-1$ edges in the MST of G.

2. Algorithm description. In this section we describe the algorithm. In section 2.1 we give an overview of the main ideas. In section 2.2 we specify the main algorithm, and in sections 2.3 and 2.4 we specify local subroutines used by the main algorithm.

2.1. Overview. The algorithm operates in phases: Each phase takes $O(1)$ rounds, and there are at most $O(\log \log n)$ phases. At the end of each phase $k \geq 0$, the nodes of G are partitioned into disjoint clusters $\mathcal{F}^k = \{F_1^k, \ldots, F_{m_k}^k\}$, $\bigcup_i F_i^k = V$. For each cluster $F \in \mathcal{F}^k$, the algorithm selects also a spanning subtree $T(F)$. The partition $\mathcal{F}^k$ and the corresponding subtree collection $\mathcal{T}^k = \{T(F) \mid F \in \mathcal{F}^k\}$, including the weights of the edges in those subtrees, are known to every vertex in the graph. (For notational consistency, we think of the initial situation at the beginning of phase 1 as the end of an "imaginary" phase 0, with each node forming a singleton cluster, i.e., $\mathcal{F}^0 = \{F_1^0, \ldots, F_n^0\}$, where $F_i^0 = \{v_i\}$ for every $1 \leq i \leq n$.)

Define a *fragment* to be a connected subtree of the MST. For a set of nodes $F \subseteq V$, denote by $\mathcal{T}(F)$ the subgraph of the MST induced by F. With these notations, we can state the following invariant, satisfied by the algorithm at the end of each phase $k \geq 0$:

> *$T(F) = \mathcal{T}(F)$ for every cluster F; namely, the spanning subtree selected for F is a fragment.*

In our model, it is easy for the nodes of each cluster to learn, in constant time, the lightest edge to every other cluster. Hence, intuitively, it is possible to "contract" each cluster C into a vertex v_C, thus creating a smaller logical graph $\hat{G}$, and continue working on this logical graph. (In practice, each real vertex belonging to some cluster C knows the weight of the edge connecting its vertex v_C to every other vertex in $\hat{G}$. The operations of each vertex v_C of the logical graph $\hat{G}$ are carried out by the real vertices belonging to the cluster C, or by a single representative called the *leader* of C, denoted $\ell(C)$.) This enables us to simulate the usual "fragment growing" MST construction process for $\hat{G}$, based on examining the edges one by one in increasing order of weight and including in the MST each inspected edge that is the *minimum-weight outgoing edge (MWOE)* of its fragment. This can be done in $O(\log n)$ time.

To reduce the time complexity to $O(\log \log n)$, it is necessary to speed up the process by making the cluster sizes grow quadratically in each phase. The main idea used for achieving this growth rate is the following. Essentially, we would like to provide every vertex v_C in the logical graph $\hat{G}$ with information about additional

edges in $\hat{G}$, beyond its own. In particular, if we were somehow able to let every vertex v_C learn the *entire* topology of $\hat{G}$, then we could finish the MST construction for $\hat{G}$ in a single step by asking each vertex in the graph to compute the MST locally. Unfortunately, such information exchange seems to require too much time. On the positive side, denoting the minimum cluster size by N, it is possible for the (N or more) members of each cluster to inform a distinguished vertex v^* of the graph, in constant time, of the N lightest edges connecting their cluster to other clusters, by appropriately sharing the workload of this task among them. (For concreteness, we assume that v^* is the node with the smallest ID in the system.)

Subsequently, we now face a special subtask of the MST construction problem to solve in v^*. This node now has a partial picture of the logical graph $\hat{G}$, consisting of all the vertices v_C but only some of the edges connecting them, particularly the N lightest edges emanating from each vertex of $\hat{G}$ (to N other vertices). It is now necessary to perform (locally) as many *legal* "fragment merging" steps as possible on the basis of this information. That is, we would like to sort the edges known to us by increasing order of weight, examine them one by one, and add edges that are the MWOE of one of the two fragments they connect, so long as we can be sure of that fact. So the question becomes: When is it "dangerous" to continue the merging steps in the absence of information about the weights of the edges unknown to us?

The answer to this question is that it is perfectly safe to continue merging a fragment F (in the logical graph $\hat{G}$), so long as for each vertex v_C in F we have still not inspected *at least one* of its N lightest edges (which is known to us by assumption). However, once we have already inspected all the edges of some vertex v_C in the fragment F, it becomes dangerous to continue attempting to merge the fragment over edges known to us, as it is possible that the true MWOE of F is the $(N+1)$st lightest edge emanating from v_C, which is not known to us (yet is lighter than any edge emanating from C that we do know of at this moment).

The crucial observation is that this "safety rule" still allows us to grow each of the fragments to contain at least $N+1$ vertices of $\hat{G}$. This means that the clusters of the next phase will be of minimum size $\Omega(N^2)$.

An interesting observation is that even when we can no longer identify the MWOE of some fragment F, we may still be able to safely merge F with some other fragment F'. This may still be legitimate if we can ascertain that the edge connecting F and F' is the MWOE of F'.

Finally, after constructing locally the new fragments, v^* sends out the identity of the edges added to the chosen set. This can be done in constant time by letting v^* send each edge to a different intermediate node, which will broadcast that edge to all other nodes.

2.2. The main algorithm. In the algorithm, whenever a node is instructed to send a message containing the edge $e=(u,v)$, this should be interpreted as a message including the IDs of its two endpoints, $ID(u)$ and $ID(v)$, as well as the edge weight $\omega(e)$.

We now describe the steps taken in phase k for all $1 \le k \le \log\log n$. Let v^* denote the node whose ID is minimal among all nodes in the graph.

Throughout, the algorithm is illustrated on the 16-vertex complete graph K_{16} with weights as depicted in Figure 1. Note that in this case there are only two phases. The flow of the algorithm is illustrated in Figure 2. In the first column, the fragment leaders are marked by horizontal stripes. Note that in the first phase all the nodes are leaders, whereas in the second phase only half of the nodes are leaders. The

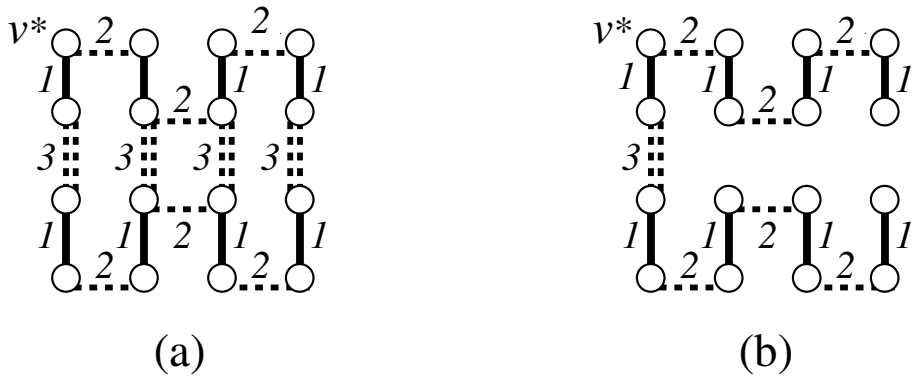

FIG. 1. *The example graph K_{16}:* (a) *The edge weights (weight* 1 *is a solid line,* 2 *is a dashed line, and* 3 *is a double-dashed line). Edges not shown in the figure have weight* 4*; hence they do not participate in the MST.* (b) *The resulting MST.*

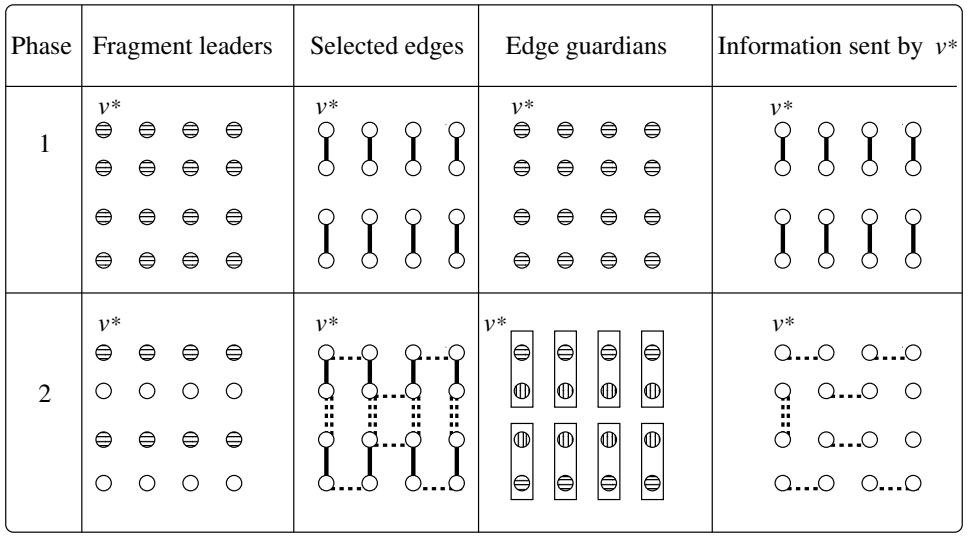

FIG. 2. *An illustration of the execution of the algorithm on the example graph K_{16}.*

second column shows the selected edges. In the first phase each fragment chooses one edge, while in the second phase each fragment chooses two edges, with the cheapest edge of each fragment denoted by a single-dashed line and the second cheapest edge denoted by a double-dashed line. The third column shows the guardian of each of the selected edges: the horizontally striped nodes are the guardians of cheapest edges and the vertically striped nodes are the guardians of the second cheapest edges. The last column shows the new edges that node v^* adds to the MST.

Phase k: Code for node v in cluster F of size $N = |F|$

Input: A set of chosen edges. The set of connected components defined by this set is the set of clusters $\mathcal{F}^{k-1}$. For each cluster $F' \in \mathcal{F}^{k-1}$, $\ell(F')$ is the node with the minimal ID in F'.

1. (a) Compute the minimum-weight edge $e(v, F')$ that connects v to (any node of) F' for all clusters $F' \neq F$.
   (b) Send $e(v, F')$ to $\ell(F')$ for all clusters $F' \neq F$.
2. **If** $v = \ell(F)$ **then**
   (a) Using the messages received from step 1, compute the lightest edge between F' and F for every other cluster F'.
   (b) Perform (locally) Procedure Cheap_Out (described below), which does the following:
   - It selects a set $\mathcal{A}(F)$ containing the N cheapest MWOEs that go out of F to $N = |F|$ distinct clusters.
   - It appoints for each such edge e a *guardian* node $g(e)$ in F, ensuring that each node in F is appointed as guardian to at most one edge.
3. Let $e' \in \mathcal{A}(F)$ be the edge for which v was appointed as guardian, i.e., such that $g(e') = v$. Send e' to v^*, the node with the minimal ID in the graph.
   (At the end of this step, v^* knows all the edges in the set $\mathcal{A} = \bigcup_{F' \in \mathcal{F}^{k-1}} \mathcal{A}(F')$.)
4. **If** $v = v^*$ **then**
   (a) Perform (locally) Procedure Const_Frags. This procedure (described below) computes E^k, the new set of edges to add.
   (b) For each edge $e \in E^k$, send a message to $g(e)$.
5. **If** v receives a message from v^* that $e \in E^k$, then v sends e to all nodes in the graph.
6. Each node adds all edges in E^k and computes $\mathcal{F}^k$.

2.3. Procedure Cheap_Out. The local procedure Cheap_Out is invoked by cluster leaders in each phase, and it operates as follows at the leader of cluster F with $|F| = N$ at phase k.

Input: Cheapest edge $e(F, F')$ for every $F' \in \mathcal{F}^{k-1}$.

1. Sort the input edges in increasing order of weight.
2. Let $\mu = \min\{N, \ |\mathcal{F}^{k-1}| - 1\}$.
3. Define $\mathcal{A}(F)$ to be the first μ edges in the sorted list.
4. Sort the nodes of F by increasing order of ID.
5. Appoint the ith node of F as the guardian of the ith edge added to $\mathcal{A}(F)$.
6. For each node $u \in F$: send the edge to which u is appointed.

2.4. Procedure Const_Frags. The local procedure Const_Frags is invoked only by the distinguished node v^*, and it operates as follows. It receives as input the initial partition $\mathcal{F}^{k-1}$, the spanning subtree collection $\mathcal{T}^{k-1}$, and the set of edges for inspection, $\mathcal{A}$. Its output is a set of edges E^k, which defines a new partition $\mathcal{F}^k$ and its spanning subtress $\mathcal{T}^k$: the edge set of $\mathcal{T}^k$ is the union of the set of edges in $\mathcal{T}^{k-1}$ with the set E^k, and $\mathcal{F}^k$ is the set of connected components of $\mathcal{T}^k$.

The procedure operates in two stages. In the first stage, it contracts the input clusters into vertices, thus creating a *logical graph* $\hat{G}$, partitions this logical graph into "superclusters," and constructs a spanning subtree for each such supercluster. In the second stage, the procedure transforms the superclusters and spanning subtrees constructed for $\hat{G}$ into clusters and spanning subtrees for the original graph G.

We now continue with a more detailed description of the two stages. The first stage operates as follows. The procedure starts by creating the logical graph $\hat{G} = (\hat{V}, \hat{E})$, where each input cluster is viewed as a vertex, namely, $\hat{V} = \mathcal{F}^{k-1}$. The edge set $\hat{E}$ consists of the logical edges corresponding to the edges of the set $\mathcal{A}$. Set the logical edge corresponding to $e = (u, w)$ to be $X(e) = (F, F')$, where $u \in F$ and $w \in F'$. Then $\hat{E} = \{X(e) \mid e \in \mathcal{A}\}$. Each logical edge $X(e)$ is assigned the same weight as e.

Then the procedure constructs a collection $\hat{\mathcal{F}}$ of superclusters and a corresponding collection $\hat{\mathcal{T}}$ of spanning subtrees on this logical graph. The construction operates as follows. The procedure first initializes the output partition as $\mathcal{F} = \{\{F\} \mid F \in \mathcal{F}^{k-1}\}$; i.e., each vertex of $\hat{V} = \mathcal{F}^{k-1}$ is a separate supercluster. The output collection of spanning subtrees is initialized to $\hat{\mathcal{T}} = \emptyset$. The procedure then inspects the edges of $\hat{E}$ sequentially in increasing order of weight. An inspected logical edge $X(e)$ is added to $\hat{\mathcal{T}}$ if it does not close a cycle with edges already in $\hat{\mathcal{T}}$. Whenever an edge $X(e) = (F_1, F_2)$ is added to $\hat{\mathcal{T}}$, the superclusters $\hat{F}_1$ and $\hat{F}_2$ containing F_1 and F_2, respectively, are merged into one supercluster $\hat{F}$, setting $\hat{F} = \hat{F}_1 \cup \hat{F}_2$ and eliminating $\hat{F}_1$ and $\hat{F}_2$, and the corresponding spanning subtrees are fused together into a spanning subtree for the new supercluster $\hat{F}$, setting $\hat{T}(\hat{F}) = \hat{T}(\hat{F}_1) \cup \hat{T}(\hat{F}_2) \cup \{X(e)\}$.

In each step during this process, whenever a logical edge $X(e) = (F_1, F_2)$ between two superclusters $\hat{F}_1$ and $\hat{F}_2$ such that $F_1 \in \hat{F}_1$ and $F_2 \in \hat{F}_2$ is inspected, the procedure also considers declaring one or two superclusters *finished* as follows:

- If the step resulted in a merge operation creating a new supercluster $\hat{F} = \hat{F}_1 \cup \hat{F}_2$, then the newly constructed supercluster $\hat{F}$ is declared finished if one of the following conditions hold:
  - e is the heaviest edge in $\mathcal{A}(F_1)$ or in $\mathcal{A}(F_2)$, or
  - either $\hat{F}_1$ or $\hat{F}_2$ is finished.
- If the step did not result in a merge between $\hat{F}_1$ and $\hat{F}_2$, then
  - the supercluster $\hat{F}_1$ is declared finished if e is the heaviest edge in $\mathcal{A}(F_1)$;
  - the supercluster $\hat{F}_2$ is declared finished if e is the heaviest edge in $\mathcal{A}(F_2)$.

Also, after every edge inspection step, some of the remaining edges become "dangerous" and are removed from the set $\mathcal{A}$. A remaining logical edge $X(e) = (F_1, F_2)$, $F_1 \in \hat{F}_1$, $F_2 \in \hat{F}_2$, is still "safe" (i.e., not dangerous) if $e \in \mathcal{A}(F_1)$ and the supercluster $\hat{F}_1$ is still unfinished, or if $e \in \mathcal{A}(F_2)$ and the supercluster $\hat{F}_2$ is still unfinished. Thus after every edge inspection step, the procedure examines every edge and removes each dangerous edge e from the set $\mathcal{A}$. The procedure also removes the corresponding logical edge $X(e)$ from $\hat{E}$. The process terminates once all superclusters are declared finished (which, as can easily be verified, happens concurrently with the set $\mathcal{A}$ becoming empty).

In the second stage, the procedure transforms the superclusters and spanning subtrees constructed for $\hat{G}$ into ones for the original graph G. Specifically, for every supercluster $\hat{F} \in \hat{\mathcal{F}}$ of the logical graph $\hat{G}$, with spanning subtree $\hat{T}(\hat{F})$, the procedure merges the original clusters included in the supercluster $\hat{F}$ into a cluster F' of G and creates the corresponding spanning subtree $T(F')$ for this cluster by merging $\hat{T}(\hat{F})$ together with all the spanning subtrees from the collection $\mathcal{T}^{k-1}$ spanning the original

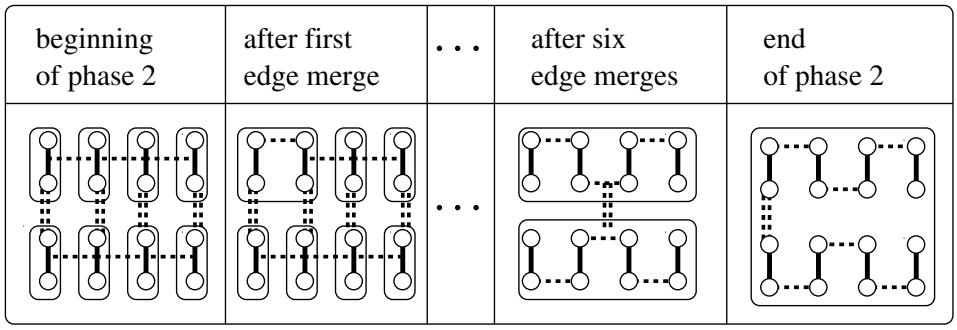

FIG. 3. *The operation of Procedure* Const_Frags *during phase* 2 *on* K_{16} *and the logical graph structure after various stages of its execution.*

clusters included in the supercluster $\hat{F}$, i.e., setting

$$T(F') = \{e \mid X(e) \in \hat{T}(\hat{F})\} \cup \bigcup_{F \in \hat{F}} T(F).$$

It then adds the cluster F' to the output cluster collection $\mathcal{F}^k$ and the spanning subtree $T(F')$ for it into $\mathcal{T}^k$.

The operation of Procedure Const_Frags during the second phase of the algorithm's execution on our example graph K_{16} is illustrated in Figure 3.

3. Analysis. In this section we prove that the algorithm described in section 2 is correct and analyze its complexity. It is more convenient to start with the complexity analysis.

3.1. Complexity. The following lemma is the key to the complexity analysis. It bounds from below the growth rate of fragments.

Consider phase k of the algorithm. Let $\hat{G}$ be the logical graph constructed by Procedure Const_Frags. Let $\hat{\mathcal{F}}$ be the collection of clusters constructed by Procedure Const_Frags for $\hat{G}$. Define μ to be the minimum between the smallest cluster size and number of clusters minus one (cf. line 2 in Procedure Cheap_Out).

LEMMA 3.1. *Every supercluster in* $\hat{\mathcal{F}}$ *consists of at least* $\mu + 1$ *logical vertices of* $\hat{G}$.

Proof. To establish the lemma, we prove the following stronger claim: whenever the procedure declares a supercluster $\hat{F}$ finished, it contains at least $\mu + 1$ logical vertices of $\hat{G}$. This claim is proved by structural induction on the superclusters.

There are three base cases. The first is when $\hat{F}$ is declared finished following a merge step $\hat{F} = \hat{F}_1 \cup \hat{F}_2$ where the two merged superclusters were unfinished. This merge step was based on the inspection of some logical edge $X(e) = (F_1, F_2)$ such that $F_1 \in \hat{F}_1$ and $F_2 \in \hat{F}_2$. By the specification of Procedure Const_Frags, without loss of generality we may assume that e is the heaviest edge in $\mathcal{A}(F_1)$. As the edges are inspected in increasing weight order, all other edges in $\mathcal{A}(F_1)$ have already been inspected. There are μ such edges, $e_{i_1}, \ldots, e_{i_\mu}$, leading to *distinct* original clusters $F_{j_1}, \ldots, F_{j_\mu}$. Whenever an edge e_{i_l} was inspected, either the superclusters containing F_1 and F_{j_l} were merged, or e_{i_l} was found to close a cycle, indicating that F_1 and F_{j_l} already belonged to the same supercluster. Hence the finished supercluster $\hat{F}$ contains (at least) the $\mu + 1$ original clusters $F_1, F_{j_1}, \ldots, F_{j_\mu}$.

The second base case is when $\hat{F}$ is declared finished following the inspection of some logical edge $X(e) = (F, F_2)$ such that $F \in \hat{F}$ and $F_2 \in \hat{F}_2$, which did not result in a merge. This happens since e is the heaviest edge in $\mathcal{A}(F)$. Again, all $\mu - 1$ other edges in $\mathcal{A}(F)$ have already been inspected, and by a similar reasoning as above, the finished supercluster $\hat{F}$ contains (at least) $\mu + 1$ original clusters. The third base case is the dual case where $\hat{F}$ is declared finished following the inspection of some logical edge $X(e) = (F_1, F)$ such that $F_1 \in \hat{F}_1$ and $F \in \hat{F}$, which did not result in a merge. Again this happens since e is the heaviest edge in $\mathcal{A}(F)$, and the claim follows in the same way.

The inductive claim concerns the case where $\hat{F}$ is declared finished following a merge step $\hat{F} = \hat{F}_1 \cup \hat{F}_2$ where one or both of the two merged superclusters were finished. In this case, the claim follows directly from the inductive hypothesis. □

LEMMA 3.2. *For any cluster $F \in \mathcal{F}^k$, $|F| \geq 2^{2^{k-1}}$.*

Proof. Denote by μ_k the minimum size of a cluster $F \in \mathcal{F}^k$. First, note that for all $k \geq 0$,

$$\mu_{k+1} \geq \mu_k(\mu_k + 1). \tag{3.1}$$

Equation (3.1) is true by Lemma 3.1, which implies that clusters generated in phase $k+1$ consist of the union of at least $\mu_k + 1$ clusters of phase k, each containing at least μ_k nodes. Now, since $\mu_0 = 1$, we have $\mu_1 \geq 2$. Since (3.1) implies that $\mu_{k+1} > \mu_k^2$, we conclude that $\mu_k > \mu_1^{2^{k-1}} = 2^{2^{k-1}}$. □

COROLLARY 3.3. *The algorithm terminates after at most $\log\log n + 1$ phases.*

Proof. The proof follows from Lemma 3.2, since the algorithm terminates at phase k in which $|F| \geq n$ for any $F \in \mathcal{F}^k$. □

The following statement is immediate from the code of the algorithm.

LEMMA 3.4. *Each phase requires $O(1)$ rounds.*

We now conclude with the following result.

THEOREM 3.5. *The time complexity of the algorithm is $O(\log\log n)$ rounds, and the overall number of bits communicated is $O(n^2 \log n)$.*

Proof. The time complexity bound follows directly from Corollary 3.3 and Lemma 3.4. For the total number of bits communicated, we account for each step separately as follows. In step 1 of the main algorithm, each node in a cluster F sends messages to all other clusters; i.e., each node sends $|\mathcal{F}^{k-1}| - 1$ messages. Since each cluster is of size at least $2^{2^{k-1}}$ by Lemma 3.2, it follows that $|\mathcal{F}^{k-1}| \leq n/2^{2^{k-1}}$. Therefore, the number of messages sent by a node at step 1 of phase k is less than $n/2^{2^{k-1}}$. Since each message contains at most $c \log n$ bits for some constant c, the number of bits sent over all phases in step 1 is less than

$$\sum_{k=0}^{\log\log n+1} n \cdot c \log n \cdot \frac{n}{2^{2^{k-1}}} \;=\; n^2 c \log n \sum_{k=0}^{\log\log n+1} 2^{-2^{k-1}} \;=\; O(n^2 \log n)\ .$$

No messages are sent in step 2. The number of messages sent in step 3 of the algorithm in each phase is $O(n)$ over all nodes (since each node receives at most one message), for a total of $O(n \log n \log\log n)$ bits throughout the execution. To account for the number of messages sent in steps 4 and 5, we bound the total number of messages sent in that step over all nodes and over all phases: note that each edge added to the MST contributes $O(n \log n)$ bits sent at steps 4 and 5, and since exactly $n - 1$ edges are added to the MST overall, the total number of bits sent in these steps throughout the execution of the algorithm is $O(n^2 \log n)$. The result follows. □

We note that by the results of Adler et al. [1] applied to our model, the minimal number of bits required to solve the MST problem is $\Omega(n^2 \log n)$ in the worst case.

3.2. Correctness. The correctness of the algorithm is proved by the following invariant.

LEMMA 3.6. *In each phase k, for every cluster $F \in \mathcal{F}^k$ constructed by Procedure* Const_Frags*, the corresponding spanning tree is a fragment, namely, $T(F) = \mathcal{T}(F)$.*

Proof. The proof is by induction on k. The initial partition, $\mathcal{F}^0$, trivially satisfies the claim. Now suppose that the collection $\mathcal{T}^{k-1}$ consists of only MST edges, and consider the collection $\mathcal{T}^k$ constructed in phase k. The spanning subtrees in this collection are composed of spanning subtrees from $\mathcal{T}^{k-1}$ fused together by new edges added by Procedure Const_Frags. It suffices to show that every edge added to the trees of $\mathcal{T}^k$ in phase k is indeed an MST edge. For this, we rely on the standard MST construction rule which says that if e is the lightest outgoing edge incident on a fragment, then it belongs to the MST. Consequently, we have to show that whenever Procedure Const_Frags selects a logical edge $X(e) = (F_1, F_2)$, $F_1 \in \hat{F}_1$, $F_2 \in \hat{F}_2$, and uses it to merge the two superclusters $\hat{F}_1$ and $\hat{F}_2$ in $\hat{G}$, then e is the lightest edge outgoing from one of the two corresponding clusters $H_1 = \bigcup_{F \in \hat{F}_1} F$ and $H_2 = \bigcup_{F \in \hat{F}_2} F$ in G.

As the edge e has not been erased prior to this step, necessarily either $e \in \mathcal{A}(F_1)$ and $\hat{F}_1$ is unfinished, or $e \in \mathcal{A}(F_2)$ and $\hat{F}_2$ is unfinished. Without loss of generality suppose the former. We claim that in this case, e is the lightest outgoing edge incident on H_1.

Consider some other outgoing edge e' incident on H_1; i.e., e' is incident on some fragment $F' \in \hat{F}_1$. Suppose, towards contradiction, that $\omega(e') < \omega(e)$. If $e' \in \mathcal{A}(F')$, then e' should have been considered by Const_Frags before e, and subsequently either added to the spanning subtree $\hat{T}(\hat{F}_1)$ or discarded as an internal edge, in either case contradicting our assumption that e' is an outgoing edge of H_1 (hence $X(e')$ is an outgoing edge of $\hat{F}_1$). It follows that $e' \notin \mathcal{A}(F')$. Let $X(e') = (F', F'')$. There may be two reasons why e' was not added to $\mathcal{A}(F')$. The first is that some other edge e'' with $X(e'') = (F', F'')$ was already included in $\mathcal{A}(F')$ before e'. In that case, $\omega(e'') < \omega(e')$, and hence also $\omega(e'') < \omega(e)$. This implies that e'' has already been inspected by the procedure at some earlier step. But then the clusters F' and F'' must already belong to the supercluster $\hat{F}_1$; and hence in $\hat{F}_1$, the edge e' is internal, a contradiction. The other possible reason why e' was not added to $\mathcal{A}(F')$ is that there are μ lighter edges incident on F', which were added to $\mathcal{A}(F')$. Letting e'' be the heaviest edge in $\mathcal{A}(F')$ in this case, it follows that $\omega(e'') < \omega(e')$, and hence $\omega(e'') < \omega(e)$. This means that e'' has already been inspected by the procedure at some earlier step. But then the supercluster $\hat{F}_0$ that contained F' at the end of that step should have been declared finished upon inspection of its heaviest edge. This would necessitate that $\hat{F}_1$ is finished now, a contradiction.

THEOREM 3.7. *The tree produced by the algorithm is an MST of the graph.*

Proof. The proof follows from Lemma 3.6 and the fact that by Lemma 3.2, $\mathcal{F}^k$ contains exactly one cluster for $k > \log \log n$. □

4. Extension to larger messages. In this section we extend the algorithm to a model in which each message can contain any number of bits (so long as it is at least $\log n$). Specifically, we assume that each meassage may contain $\ell \log n$ bits. The extension of the algorithm to this case is straightforward. It turns out that the asymptotic worst-case number of rounds drops to a constant when the message size

is n^ϵ for $\epsilon > 0$, but $\Theta(\log \log n)$ rounds are required by our algorithm for any polylog message size.

First, we explain how to modify the algorithm to use messages that can contain ℓ edges. The idea is to change steps 2(b) (which is the invocation of Procedure Cheap_Out) and 3 in the main algorithm so that each node can be the guardian of ℓ edges. Specifically, the modified algorithm is identical to the algorithm of section 2, except for the following steps.

2b*. Perform (locally) Procedure Cheap_Out*. This procedure (described below) does the following:
- It selects a set $\mathcal{A}(F)$ containing the $\ell \cdot N$ cheapest edges that go out of F to $\ell \cdot N$ distinct clusters.
- It appoints for each such edge e a *guardian* node $g(e)$ in F, ensuring that each node in F is appointed as guardian to at most ℓ edges.

3*. Let $\{e'_1, \ldots, e'_\ell\} \subseteq \mathcal{A}(F)$ be the edges for which v was appointed as guardian, i.e., all edges e'_i such that $g(e'_i) = v$. Send $\{e'_1, \ldots, e'_\ell\}$ to v^*, the node with the minimal ID in the graph.
(At the end of this step, v^* knows all the edges in the set $\mathcal{A} = \bigcup_{F' \in \mathcal{F}^{k-1}} \mathcal{A}(F')$.)

The modified Cheap_Out* procedure is identical to Procedure Cheap_Out, except for the following two steps:

2*. Let $\mu = \min\{\ell \cdot N, \ |\mathcal{F}^{k-1}| - 1\}$.

5*. Appoint the ith node of F as the guardian of the jth edge added to $\mathcal{A}(F)$ if $j \bmod (\ell \cdot N) = i$.

The correctness of the modification is obvious, as Lemma 3.6 is stated in terms of a general μ, and it relies only on the assumption that $\mathcal{A}(F)$ contains the μ lightest edges connecting F to μ distinct clusters.

The complexity analysis of the generalized algorithm requires a little work. First, we observe that Lemma 3.1 holds without change: it is also stated in terms of a general μ. Lemma 3.4 also holds by the assumption that each message can contain ℓ edges, and since each node is the guardian of at most ℓ messages by the modified procedure Cheap_Out*. However, Lemma 3.2 holds only for $\ell = \Theta(1)$. Below we generalize Theorem 3.5 to different values of ℓ.

THEOREM 4.1. *The extended algorithm terminates in $O(\log(\frac{\log n}{\log \ell}))$ rounds, and the total number of bits communicated is $\Theta(n^2 \log n)$.*

Proof. Let μ_k be the smallest possible cluster size after the kth round. By definition, $\mu_0 = 1$. If each guardian node sends ℓ edges, then each cluster merges with at least $\ell\mu_k$ other clusters in the kth phase. It follows that $\mu_{k+1} \geq (\ell\mu_k + 1)\mu_k > \ell\mu_k^2$. Since $\mu_0 = 1$, we have $\mu_1 > \ell$, and therefore $\mu_k > \ell^{2^{k-1}}$. The bound on the number of rounds follows, since the algorithm terminates when $\mu_k \geq n$, and each phase takes only $O(1)$ rounds by Lemma 3.4. For the total number of bits communicated by the algorithm, we observe that the only difference is the number of messages sent in step 3*. Let B_3 denote the total number of bits sent in step 3* throughout the execution of the algorithm. We claim that $B_3 = O(n^2 \log n)$. To see this, note that the total number of edge identifiers sent in step 3 in a single phase, over all nodes, is $O(n\ell)$. It follows that $B_3 = O(Tn\ell \log n)$, where T is the number of phases. As shown

above, $T = O(\log(\frac{\log n}{\log \ell}))$. This means that $B_3 = O(n^2 \log n)$: If $1 < \ell < n/\log \log n$, then T is bounded by $O(\log \log n)$; and if $\ell \geq n/\log \log n$, then $T = O(1)$. □

Let us interpret the result of Theorem 4.1 in two typical cases. First, if ℓ is polynomial in n, i.e., $\ell = n^\epsilon$ for some $\epsilon > 0$, then the total running time of the algorithm is $O(\log(1/\epsilon))$. However, the number of rounds remains $\Theta(\log \log n)$ if ℓ is only polylogarithmic in n. (In fact, it remains $O(\log \log n)$ even if ℓ is as large as $(\log n)^{(\log n)^{1-\epsilon}}$ for some constant $\epsilon > 0$.)

5. Conclusion. This paper shows that an MST can be constructed in sublogarithmic time, even if each message can contain only a constant number of edges. We believe that the algorithm may be useful in some overlay networks. Theoretically, important gaps remain. While there are nontrivial lower bounds on the running time of MST construction in graphs of diameter 3 or more, currently no superconstant lower bound is known even for graphs of diameter 2. We do not know whether there exist lower bounds or better algorithms for graphs of diameter 1 and 2 (recall that the fastest known algorithm for diameter 2 runs in $O(\log n)$ rounds).

Acknowledgment. We thank the anonymous reviewers, whose comments helped improve the presentation of the paper.

REFERENCES

[1] M. Adler, W. Dittrich, B. Juurlink, M. Kutylowski, and I. Rieping, *Communication-optimal parallel minimum spanning tree algorithms*, in Proceedings of the 1998 ACM Symposium on Parallel Algorithms and Architecture, 1998, pp. 27–36.

[2] B. Awerbuch, *Complexity of network synchronization*, J. ACM, 32 (1985), pp. 804–823.

[3] B. Awerbuch, *Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 230–240.

[4] B. Awerbuch and Y. Shiloach, *New connectivity and MSF algorithms for the shuffle-exchange network and PRAM*, IEEE Trans. Comput., 36 (1987), pp. 1258–1263.

[5] R. Cole, P. N. Klein, and R. E. Trajan, *Finding minimum spanning forests in logarithmic time and linear work using random sampling*, in Proceedings of the ACM Symposium on Parallel Algorithms and Architecture, 1996, pp. 243–250.

[6] R. G. Gallager, P. A. Humblet, and P. M. Spira, *A distributed algorithm for minimum-weight spanning trees*, ACM Trans. Prog. Lang. and Syst., 5 (1983), pp. 66–77.

[7] J. A. Garay, S. Kutten, and D. Peleg, *A sublinear time distributed algorithm for minimum-weight spanning trees*, SIAM J. Comput., 27 (1998), pp. 302–316.

[8] J. Jannotti, D. Gifford, K. L. Johnson, M. F. Kaashoek, and J. J. W. O'Toole, *Overcast: Reliable multicasting with an overlay network*, in Proceedings of the 4th Annual USENIX OSDI, 2000, pp. 197–212.

[9] D. R. Karger, P. N. Klein, and R. E. Tarjan, *A randomized linear time algorithm to find minimum spanning trees*, J. ACM, 42 (1995), pp. 321–328.

[10] Z. Lotker, B. Patt-Shamir, and D. Peleg, *Distributed MST for constant diameter graphs*, in Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, 2001, pp. 63–71.

[11] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, San Mateo, CA, 1995.

[12] D. Peleg and V. Rubinovich, *Near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction*, SIAM J. Comput., 30 (2000), pp. 1427–1442.

[13] J. H. Saltzer, D. P. Reed, and D. D. Clark, *End-to-end arguments in system design*, ACM Transactions on Computer Systems, 2 (1984), pp. 277–288.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, in Proceedings of the ACM SIGCOMM '01 Conference, San Diego, CA, 2001.

[15] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conf. Ser. in Appl. Math. 44, SIAM, Philadelphia, 1983.