ELSEVIER Theoretical Computer Science 243 (2000) 217–241

Theoretical Computer Science

www.elsevier.com/locate/tcs

Simple and efficient network decomposition and synchronization

Shlomo Moran [*], Sagi Snir

Computer Science Department, Technion, Israel Institute of Technology, Haifa, 32000, Israel

Received June 1997
Communicated by M. Nivat

Abstract

We present a simple and efficient method for constructing sparse decompositions of networks. This method is used to construct the sparse decompositions needed for variants of the synchronizers in [2, 15] in $O(|V|)$ time and $O(|E| + |V| \log |V|)$ communication complexities, while maintaining constant messages size and constant memory per edge. Using these decompositions, we present simple and efficient variants of the synchronizers in the above papers. For example, our constructions enable to perform *Breadth First Search* in an asynchronous network, in which no preprocessing had been done, in communication and time complexities of $O(K|V|D + |E| + |V| \log |V|)$ and $O(D \log_K |V| + |V|)$, respectively, where $K \geqslant 2$ is a parameter, and D is the diameter of the network. We also present an efficient cover-coarsening algorithm, which uses a novel technique for efficient merging of clusters, and improves previous coarsening algorithms in several aspects. © 2000 Elsevier Science B.V. All rights reserved

Keywords: Graph decompositions; Asynchronous communication networks; Synchronizers

1. Introduction

Communication networks are divided into two types: *synchronous networks* and *asynchronous networks*. In the synchronous model all the nodes have access to a global clock that generates pulses. At the time of a pulse, a node may perform local computation and send messages to its neighbors. Messages which are sent in a given pulse arrive at their destination before the next pulse.

In the asynchronous model there is no such a clock and message delay is arbitrary but finite. Algorithms that run on synchronous networks are called synchronous algorithms and are simpler, more efficient and more comprehensible. In order to run synchronous

[*] Supported in part by the Bernard Elkin Chair in Computer Science, and by US–Israel BSF grant 95-00328.

E-mail address: moran@cs.technion.ac.il (S. Moran).

algorithms on asynchronous networks, there is a need to synchronize the network. This is done by algorithms named *synchronizers* that simulate the synchronous algorithm on the network.

Some synchronizers require a preprocessing phase in which the network is decomposed into connected components called *clusters*. This paper presents simple and efficient network decomposition algorithms; these algorithms are then used to construct few variants of existing synchronizers, which are simpler and more efficient than the original synchronizers.

Once the network is decomposed into clusters, it is sometimes required to *coarsen* the decomposition, by combining several neighboring clusters to a single cluster, so as to decrease the overlapping between clusters, and at the same time to keep the diameter of each cluster small. We present a new cover-coarsening algorithm, which improves a previous one in several aspects.

1.1. Definitions

A network is represented by a graph $G=(V,E)$ where V represents the nodes in the network, and E the links between them. All nodes have distinct identities. A *cluster* C in a graph G is a set of nodes which induces a connected subgraph of G. Each cluster has a leader node and a tree spanning it, rooted at the leader. A *cover* is a set S of clusters such that $\bigcup_{C\in\mathcal{S}} C=V$. A *partition* is a cover in which the clusters are mutually disjoint. Covers (partitions) are denoted by calligraphic letters (such as $\mathcal{S},\mathcal{T}$, etc.) and clusters by capital letters (such as A,B, etc.).

For vertices u and v in G, the distance between u and v in G, denoted $dist_G(u,v)$ (or just $dist(u,v)$), is the length of a shortest path between u and v in G. The m neighborhood of a node v is defined by $N_m(v)=\{u \mid dist_G(u,v)\leqslant m\}$, and the m-neighborhood of a cluster C is the union $\bigcup_{v\in C} N_m(v)$. The diameter of a cluster C, $Diam(C)$, is the diameter of the tree spanning C. For a node v and a cluster C, $dist(v,C)=min\{dist_G(u,v) \mid u\in C\}$. For two clusters C and C', $dist(C,C')=min\{dist_G(u,v) \mid u\in C, v\in C'\}$. We say that two clusters C and C' are *neighboring* if $dist(C,C')\leqslant 1$. Two clusters C and C' *intersect* if they have at least one common node. A cover $\mathcal{T}$ is said to *coarsen* a cover $\mathcal{S}$, if every cluster $S\in\mathcal{S}$ is included in some cluster $T\in\mathcal{T}$. The *volume* of a cover $\mathcal{S}$, denoted as $Vol(\mathcal{S})$, is the sum $\sum_{C\in\mathcal{S}} |C|$, and the diameter of $\mathcal{S}$, $Diam(\mathcal{S})$, is $\max_{C\in\mathcal{S}} Diam(C)$. The *degree* of a node v in a cover $\mathcal{S}$, $Deg_{\mathcal{S}}(v)$, is the number of clusters in $\mathcal{S}$ which contain v; The degree of a cover $\mathcal{S}$, $\Delta(\mathcal{S})$, is $\max_{v\in V} deg_{\mathcal{S}}(v)$.

A node in the network receives messages, processes them, performs local computations, and then changes its status and/or sends messages to its neighbors. These actions are assumed to be performed in negligible time, i.e. computation time is not taken into account. The messages are received in FIFO order, after a finite but unknown delay. The system is assumed to be error-free, i.e. no messages error or messages loss.

The following complexity measures are used to evaluate performances of an algorithm operating in a network. *Communication* (*bit*) *complexity*: The worst-case number of messages (bits respectively) sent in the network during an execution of the algorithm. *Messages size*: The worst-case length of a message sent in the network, measured in bits. *Time complexity*: In a synchronous network – the worst-case number of pulses from the start of the algorithm to its termination. In an asynchronous network – the worst-case number of time units from the start of the algorithm to its termination, assuming that a message delay is at most one time unit. *Memory complexity*: The worst-case memory size required at a node in the network, measured in bits.

1.2. Previous results

The synchronizers studied in this paper are based on the approach introduced in [2]. This approach can be described as follows. First, the synchronous algorithm is modified so that each message is acknowledged by its reciever. Second, a *sparse decomposition* of the network is constructed. Finally, an algorithm which uses this decomposition to enable each node to learn when it had received all the messages sent to it, is introduced. In evaluating the complexity measures of the synchronizer, one considers the complexity of the preprocessing phase, and the overhead per pulse of the synchronizer. [1]

The sparse decomposition used in [2] is a partition $\mathscr{P}$ of the network to (disjoint) clusters, such that both $Diam(\mathscr{P})$ and $\sum_{C\in\mathscr{P}}|N_1(C)|$ are kept small. An alternative definition of sparse decomposition, introduced and used in [7, 8, 14, 15], considers covers $\mathscr{S}$ consisting of overlapping clusters, such that both $Diam(\mathscr{S})$ and $Vol(\mathscr{S})$ are kept small. [2] [7, 8, 14] study the construction of sparse decompositions which are coarsening of a given *input cover*. When the input cover is the set of the edges of the graph, one get a decomposition similar to the partition used in [2]. Other applications of sparse decompositions can be found in [1, 9].

Awerbuch [2] presents a distributed algorithm for constructing the sparse decomposition used by his synchronizer. The bottleneck of this algorithm are procedures for performing the *next cluster's leader election*, and electing the *preferred edges* for inter-cluster communication. [15] presents three synchronizers, named η_1, η_2 and θ, which are similar to γ; η_2 reduces the communication overhead by half, θ reduces the time overhead by half, and all of them eliminate the need to elect preferred edges. The preprocessing phases required for these three synchronizers are performed in [15] by a general distributed cover-coarsening algorithm, which improves upon the complexity of the preprocessing in [2] by avoiding the need to elect preferred edges.

Awerbuch and Peleg [6] present a synchronizer which differs from the previous ones in that it does not simulate the synchronous algorithm in a "pulse by pulse" manner, and it requires that the originators of the simulated synchronous algorithm are known

[1] Note that there is a potential overhead also in the acknowledgement messages which were added to the synchronous algorithm; however, this overhead is typically negligible compared to the overhead introduced by the synchronizer, and hence is ignored.

[2] Another definition, which is not used by the constructions in this paper, requires that both $Diam(\mathscr{S})$ and $\Delta(\mathscr{S})$ are kept small.

in advance. This synchronizer transforms a synchronous algorithm whose time and communication complexities are t and m to an asynchronous algorithm whose time and communication complexities are $O(t \log^3 |V|)$ and $O(m \log^3 |V|)$, respectively. In certain cases, it outperforms the above synchronizers in communication complexity, but not in time complexity. The amount of memory needed by this synchronizer may be linear at $|V|$ in some nodes, and its messages size is $O(\log |V|)$. This synchronizer and its construction are considerably more complicated than the other synchronizers in [2, 15] mentioned above.

Awerbuch and Peleg [7] present a centralized cover-coarsening algorithm which outputs a cover with relatively small diameter, which also has either small volume or small degree. Awerbuch and Peleg [5] present a distributed algorithm for implementing the algorithms of [7], in an asynchronous setting, by means of the synchronizer described in [6].

A related decomposition, to be denoted *block decomposition*, was introduced and applied in [4, 13]. In this decomposition the nodes of the graph are partitioned into *blocks*, which do not necessarily induce connected subgraphs. It is required that both the number of blocks and the diameters of the connected components induced by nodes in the same block are small. Algorithms which construct good sparse decompositions are used to construct good block decomposition (see e.g. [13]). It is not clear whether the opposite is also true; i.e., whether constructions of good block decompositions can provide the sparse decompositions needed for our synchronizers. Linial and Saks [13] also present an elegant asynchronous randomized algorithm for constructing block decomposition in $O(\log^2 |V|)$ time and $O(|E| \log^2 |V|)$ communication complexities.

1.3. Our contribution

The contribution of this paper can be divided to three: first, we present a general simple graph-decomposition methodology, which provides better complexity bounds for the preprocessing phases of the algorithms discussed here. Second, we use this technique for constructing two variants of the algorithms in [15]; our variants are based on new constructions of sparse decompositions needed for these synchronizers, and are more efficient both in the preprocessing phase and in the memory and bit complexities of the resulted synchronizers. Finally, we provide a cover-coarsening algorithm, called ζ, which is improvement of the one in [15] (which is an improvement of a similar algorithm in [7]). The input/output specifications of algorithm ζ are similar to those of the coarsening algorithm of [15], and it improves upon this latter algorithm in two ways: First, it employs a novel technique for merging old clusters into new ones, and thus avoids certain “bad” scenarios in the algorithm of [15]; second, it obtains better complexity measures, as indicated in Fig. 2 (the comparison of the complexity measures holds also when the above mentioned bad scenarios are ignored).

The performances of our synchronizers in overhead per pulse are compared with these of previous synchronizers in Fig. 1, and the complexity measures of the preprocessing phases of these synchronizers are compared in Fig. 2 (MS97 in these figures

Synchronizer	Communication	Time	Message Size	Reference
γ	$(2K+2)\|V\|$	$4\log_K \|V\|$	O(1)	Awe85
θ	$2K\|V\|$	$2\log_K \|V\|$	$O(\log \|V\|)$	SS94
γ_1	$2K\|V\|$	$2\log_K \|V\|$	O(1)	MS97
η_2	$(K+2)\|V\|$	$4\log_K \|V\|$	$O(\log \|V\|)$	SS94
γ_2	$(K+2)\|V\|$	$4\log_K \|V\|$	O(1)	MS97

Fig. 1. Synchronizers overhead per pulse.

Synchr-onizer	Communication complexity	Time complexity	Bit complexity	Space per edge	Reference
γ	$O(\|V\|^2)$	$O(\|V\|\log_K \|V\|)$	$O(\|V\|^2 \log \|V\|)$	O(1)	Awe85
θ, η_2	$O(\|V\|\log_K \|V\|+\|E\|)$	$O(\|V\|\log_K \|V\|)$	$O((\|E\|+\|V\|\log_K \|V\|)\cdot\log \|V\|)$	$O(\log \|V\|)$	SS94
γ_1, γ_2	$O(\|E\|)$	$O(\|V\|)$	$O(\|V\|\log \|V\| + \|E\|)$	O(1)	MS97
ζ	$O(\|E\|)$	$O(\|V\|)$	$O((\|V\|\log \|V\|+\|E\|)\log \|V\|)$	$O(\log \|V\|)$	MS97

Fig. 2. Preprocessing complexities.

refers to this paper). We note that the complexity measures in Fig. 2 are valid regardless of whether the network has a leader or not.

1.4. An overview

In Section 2 we review the synchronizers of [2] and [15]; in Section 3 we present a general method for network decomposition, and use it to construct an efficient preprocessing phase of a variant of synchronizer γ of [2]. In Section 4 we use this method for presenting two synchronizers, γ_1 and γ_2, which are improved variants of synchronizers θ and η_2 of [15]. Finally, in Section 5 we present our cover coarsening algorithm.

2. Reviewing previous synchronizers

In this section we review synchronizer γ of [2] and two of its variants – called θ and η_2 – introduced in [15].

The messages sent by γ and its variants can be divided to two sets: *original messages*, which correspond to messages sent by the simulated synchronous algorithm (including acknowledgements), and *control messages* (or *synchronization messages*), which are additional messages sent by the synchronizer to guarantee synchronization. To each original message there corresponds a *pulse number*, which is the pulse in which it is supposedly sent in the synchronous execution.

Definition 1. A node is *safe* for a pulse p if it has received acknowledgments for all original messages sent by it in this pulse.

Note that in order for a synchronizer to be correct, it is sufficient to ensure that a node v sends messages of the next pulse only after all its neighbors became safe in the current pulse. If the FIFO discipline is not assumed, then it is also necessary that the node itself is safe in the current pulse.

Synchronizer γ and its variants can be viewed as a combination of two synchronizers – α and β [2]. In synchronizer α, each node, upon becoming safe at pulse p, sends a SAFE message to all its neighbors. A node, upon receiving SAFE messages from all its neighbors, produces the next pulse of the original algorithm. Communication and time overhead of the synchronizer per pulse are $O(|E|)$ and $O(1)$, respectively.

In synchronizer β, a tree spanning the network is constructed in a preprocessing phase. Each node, upon becoming safe at pulse p and receiving SAFE messages from all its children, sends a SAFE message to its parent in the tree. When the leader has received SAFE messages from all its children, it sends GOAHEAD messages to all its children. A node, upon receiving the GOAHEAD message, passes it on to its children in the tree and produces the (messages of the) next pulse of the original algorithm. Communication and time complexities of the synchronizer per pulse are both $O(|V|)$.

2.1. Synchronizer γ

In synchronizer γ, the network is first partitioned into disjoint clusters, and specific edges, called *preferred edges*, are selected to connect neighboring clusters. Each cluster is constructed by first electing a cluster leader, and then constructing a BFS tree around that leader. The tree is constructed layer by layer, where each layer consists of neighbors of nodes in the previous layer which do not belong to any cluster yet. A new layer joins the cluster as long as its magnitude is at least $K-1$ times the magnitude of the already existing cluster, where $K>1$ is a parameter. This rule, which will be denoted `Fast Growth Rule` is kept by all the variants of γ discussed in this paper. The use of the `Fast Growth Rule` in the preprocessing phase of γ guarantees that the depth (height) of each cluster is bounded by $\log_K |V|$, and that the number of preferred edges used for inter cluster communication is bounded by $(K-1)|V|$.

The preprocessing stage of γ is composed of four tasks with the following complexities:

1. Electing a leader in the network. This requires communication and time complexities of $O(|V|\log|V|+|E|)$ and $O(|V|)$, respectively [3, 11]. We note that the preprocessing phase in [2] used for this task the earlier algorithm of [12], whose communication and time complexities are $O(|V|\log|V|+|E|)$ and $O(|V|\log|V|)$, respectively.[3]
2. Clusters creation, with overall communication and time complexities of $O(|V|\log_K|V|+|E|)$ and $O(|V|)$, respectively.
3. Finding the next cluster's leader, with overall communication and time complexities of $O(|V|^2)$ and $O(|V|\log_K|V|)$, respectively.

[3] This does not affect the complexity measures of the algorithm in [2], which are dominated by the task of electing preferred edges

4. Electing preferred edges, with communication and time complexities of $O(|V|^2)$ and $O(|V| \log_K |V|)$, respectively.

A simulation of pulse p by synchronizer γ is done as follows:

1. A node that becomes safe and receives SAFE messages from all its children, sends a SAFE message to its parent. This process, called CONVERGECAST, is initiated by the leaves and terminates at the root. It requires $\log_K |V|$ time and $|V|$ communication complexity.
2. After receiving SAFE messages from all its children, and becoming safe itself, the root broadcasts a CLUSTER SAFE message to all its cluster. This message is forwarded also over the preferred edges. This requires $\log_K |V|$ time and $(2K-1)|V|$ communication complexity.
3. When a leaf receives a CLUSTER SAFE message from its parent and over all adjacent preferred edges, it initiates a convergecast of READY messages. A node sends a READY message to its parent after it receives READY messages from all its children, and CLUSTER SAFE messages over all its preferred edges. This phase has $\log_K |V|$ time and $|V|$ communication complexity.
4. Finally, after receiving READY messages from its children and CLUSTER SAFE messages over adjacent preferred edges, the root broadcasts GOAHEAD message to its cluster, which initiates the next pulse simulation. this requires $\log_K |V|$ time and $|V|$ communication complexity.

Thus, the overhead per pulse in this simulation is $(2K+2)|V|$ in communication and $4\log_K |V|$ in time. Note that using γ is worthwhile only when $2K|V| < |E|$, as otherwise the simpler synchronizer α is more efficient.

2.2. Synchronizers θ and η_2

In [15], Shabtay and Segall presented three modifications of synchronizer γ, called θ, η_1 and η_2, which avoid the need to use (and hence to elect) preferred edges. Instead, the inter-cluster communication is obtained by having neighboring clusters share common nodes (i.e., the clusters used are not disjoint). Thus, communication on the edges of the clusters' spanning trees is sufficient.

The preprocessing phases of these synchronizers are based on a general cover-coarsening technique, based on the constructions in [7, 14]; Shabtay and Segall [15] also present an improved algorithm to construct sparse covers, which we will discuss in details in Section 5.

Synchronizers θ and η_2[4] are conceptually simpler than γ, as they use only one convergecast and one broadcast per pulse. Also, synchronizer θ has communication and time overheads per pulse of $2K|V|$ and $2\log_K |V|$, respectively, while in η_2 these overheads are $(K+2)|V|$ and $4 \log_K |V| + 1$. However, due to the more complex structure of the covers employed, the memory and bit complexities of both θ and η_2 are no longer constant, as the corresponding complexities of γ are.

[4] Synchronizer η_1 is inferior to η_2, and is not described here.

3. The decomposition method

Constructions of sparse decompositions, needed for the synchronizers discussed in this paper, are traditionally done in a “semi-sequential” manner, in which the clusters are constructed one by one, while each individual cluster is constructed in parallel. The constructions of the individual clusters depend on a parameter $K > 1$, such that the diameters of the clusters are $O(\log_K |V|)$ and the volume of the decomposition is bounded by $K|V|$.

In this section we present a general, simple method for carrying out constructions of this type. As we shall see, the difference between various decomposition algorithms is in the way they maintain and select potential leaders, and the rules by which the individual clusters are constructed. Our method treats the construction of an individual cluster as a “black box”, and concentrates in the way the cluster leaders are selected. For this sake, we introduce the notion of a *potential leader*, which is a node which can start the construction of a new cluster. All the decomposition algorithms discussed in this paper are performed (sometimes implicitly) according to the following scheme:
While there are potential leaders in the network do
1. Find a node v which is a potential leader.
2. Construct a cluster whose leader is v.

In [2, 15], a potential leader is elected in step 1 above from the last constructed cluster or from the set of nodes that were rejected from that cluster. If no node can be found in that last cluster, the next cluster’s leader is elected from the most recently created cluster with a non empty set of potential leaders. This task in [2] requires communication and time complexities of $O(|V|^2)$ and $O(|V| \log_K |V|)$, respectively, while in [15] communication and time complexities are both $O(|V| \log_K |V|)$, but memory requirement could be $O(|V|)$ per node. Another method mentioned by [5] traverses the last created cluster in a depth-first search (DFS) manner (cf. [10]), seeking for a potential leader, and when this cluster is exhausted, it backtracks to the cluster from which this one was created. The communication and time complexities of this method are proportional to the volume of the output cover (which is $O(K|V|)$), since two messages are sent on each edge of each spanning tree of each output cluster.

In this work we use the following simple method for finding the next potential leader: first, a spanning tree **T** of the whole network is constructed. If a leader is given, this requires $O(|E|)$ communication and $O(|V|)$ time, else the communication and time required, are $O(|V| \log |V| + |E|)$ and $O(|V|)$, respectively [3, 11]. All the nodes in **T** are marked as potential leaders. Now, the first cluster is constructed, starting from the root of **T**. A node that joins a cluster and satisfies a certain condition, which depends on the specific decomposition, stops being a potential leader (for constructing the partition needed for γ, **every** node that joins a cluster stops being a potential leader). When the cluster construction is terminated, the leader of the cluster (which is the root) starts a DFS traversal of **T**, searching for the next potential leader. Once such a node v is found, the traversal is suspended, and a new cluster is constructed with v as its leader. When this cluster is completed, the traversal is renewed until the next potential leader is found, and so on. This procedure continues until the traversal of the spanning tree **T** is terminated at its root.

The overhead in communication of finding the next cluster leader is $O(|V|)$, and since the clusters are constructed one by one, this is also the overhead in time. The overhead in memory needed to maintain the spanning tree is $O(1)$ bits per edge.

We complete this section by noting that in a network with a leader, the above technique can be used to reduce the communication, bit and time complexities of the preprocessing algorithm of synchronizer γ in [2] from $O(|V|^2)$, $O(|V|^2 \log |V|)$ and $O(|V| \log_K |V|)$ to $O(|E|)$, $O(|E| + |V| \log |V|)$ and $O(|V|)$, respectively. If a leader is not given, communication, bit and time complexities are reduced to $O(|V| \log |V| + |E|)$, $O((|V| \log |V| + |E|) \log |V|)$ and $O(|V|)$, respectively. This is done by first using this technique to perform the next cluster leader election required by this algorithm (task 3 in Section 2.1), thus reducing the time and communication complexities of this task to $O(|V|)$. In addition, we eliminate the stage of electing preferred edges (task 4 in Section 2.1) altogether; instead, whenever a node v is rejected from a cluster C which is currently being built, v defines the edge which connects it to the node in C which had sent him a rejection message as a preferred edge. The number of edges that are selected this way is bounded by $(K-1)$ times the number of nodes in the cluster, which is the same bound achieved by the construction in [2]. (Note, however, that it is possible that two neighboring clusters will be connected by more than one preferred edge.) Since this change in the construction does not affect the structure of the clusters constructed in the preprocessing phase, and it still guarantees that at least one preferred edge will connect each pair of neighboring clusters, the correctness and worst-case complexity of the synchronizer are unaffected by this change.

The reduction in bit complexity of our preprocessing algorithm to $O(|E| + |V| \log |V|)$ follows from the fact that all the messages sent by it are of constant size, except $O(|V|)$ messages which are used to count the number of nodes in new layers that join clusters;[5] each such message is of size at most $\log |V|$.

4. Improved variants of synchronizers θ, η_2

In this section we present two synchronizers γ_1 and γ_2 which are improved variants of synchronizers θ and η_2 of [15]. Synchronizers η and θ_2 use a sparse cover in which the clusters may overlap, and thus save the need to use (and elect) preferred edges. The preprocessing algorithms for our synchronizers are tailored to meet the topological requirements of θ and η_2; however, they outperform the preprocessing algorithms of θ and η_2 in all four complexity measures (see Fig. 2). Moreover, the resulting decompositions are easier to maintain in terms of space and bit complexities. As a result, the resulting synchronizers outperform the corresponding synchronizers in [15] in

[5] The fact that there are only $O(|V|)$ such messages follows from an argument similar to the one given in Section 4.3.2.

memory requirements and/or in the size of the messages used, while maintaining the same communication and time overheads (see Fig. 1). The preprocessing phases of both γ_1 and γ_2 are based on the method introduced in Section 3, where the construction of the individual clusters is described in the following section.

4.1. Clusters construction

In the preprocessing phases of both γ_1 and γ_2 each cluster C is constructed in iterations, where the nodes added to C during iteration i are denoted as *layer* i of C; the first layer of C, which includes the root, is layer 0. The leader of the cluster initiates iteration i by broadcasting a message along the edge of the spanning tree of C. Upon receiving this message, nodes in layer $i-1$ send messages to their neighbors, asking them to join layer i of C. A node v which receives this message and wishes to join the cluster, sends a positive acknowledgment to the first node, u, from which it received this request, and makes u its parent in (the tree spanning) C. At the end of iteration i, each node informs its parent in this tree how many descendants it has in layer i. This enables the root to determine the number of nodes in layer i. If this number fulfills the `Fast Growth Rule` – i.e., is at least $K-1$ times the current size of cluster C – then the leader initiates the construction of layer $i+1$; else, it announces the termination of the construction of the cluster. This announcement is acknowledged by the leaves to the root, who then continues the DFS traversal in a search for the leader of the next cluster. The last layer that joins cluster C, whose size violates the `Fast Growth Rule`, is denoted as the *last layer* of C. Thus, every cluster contains a last layer, which might be empty.

In the construction above, a node v is a *tenant in a cluster* C, if it belongs to C but it is not in the last layer of C; v is a *tenant* if it is a tenant in some cluster C (initially, all nodes are not tenants).

4.2. Variant 1: *Synchronizer* γ_1

4.2.1. Synchronizer γ_1: *preprocessing*

The preprocessing phase of γ_1 guarantees that for each edge $e=\{u,v\}$ in G, there is a cluster C which includes both u and v. We describe how this is achieved, using the construction in Section 4.1 above: First, a potential leader is defined to be a node v which is not a tenant yet. The construction of a cluster C is initiated in iteration 0, where the cluster leader becomes the root of C, which is also layer 0 of C. For $i>0$, iteration i proceeds as follows: Each node v in layer $i-1$ of C, which receives an announcement to start iteration i, marks itself as a tenant of C, and then adds to layer i of C all its neighbors which are not in C and are not tenants in any other cluster. The root computes the number of nodes in layer i of C, as described above. If this number satisfies the `Fast Growth Rule`, then it instructs the nodes in layer i to initiate iteration $i+1$, otherwise it announces the termination of the construction.

Lemma 4.1. *A node is a tenant in at most one cluster, and it is a leaf in any other cluster it belongs to.*

Proof. A node v can be added to a cluster C only if it is not a tenant yet. Thus, once v becomes a tenant in some cluster C, it cannot join, let alone become a tenant, in any other cluster. Clearly, v may have children in a cluster C only if it is a tenant in C. □

Lemma 4.2. *The construction algorithm terminates, and when it terminates, any two neighboring nodes u and v are members in a common cluster.*

Proof. *Termination*: Since a cluster expansion is bounded by the network size, the construction of each individual cluster must terminate. Since whenever a new cluster is constructed, the cluster leader becomes a tenant of this cluster, the number of tenants increases by at least one in each construction of a new cluster. Thus, eventually all the nodes in the graph are tenants, and the traversal terminates at the root, terminating the whole construction phase.

Correctness: By the discussion above, when the construction algorithm terminates, all the nodes in the graph are tenants. Thus, it suffices to show that if v is a tenant then for each neighbor u of v there is a cluster which contains both u and v. We prove this by induction on the number of times a layer is added to a cluster during the run of the algorithm. Denote this number by l. The base $l=0$ is trivial – no node is a tenant yet. Assume the claim is true for all $j<l$, and let v be a node that becomes a tenant in cluster C in the lth iteration. We have to show that every neighbor u of v is in a common cluster with v. If u became a tenant before the lth iteration, then u, v are in the same cluster by the induction hypothesis. Otherwise, u is added to C in the lth iteration, if it was not added to C earlier. In both cases, after v became a tenant, it shares a common cluster with u. □

We note that the preprocessing phase of γ_1 is very similar to the improved version of the preprocessing phase of synchronizer γ, as presented in Section 3. The main difference is that in γ_1 the "last layers" are not rejected, but added to the constructed cluster. Thus, the communication, bit and time complexities of the preprocessing phase of γ_1 are the same as those of the improved version of the preprocessing phase of γ, described in Section 3.

4.2.2. Synchronizer γ_1: operation

The simulation of a pulse in γ_1 consists of two phases:

1. A node v which belongs to a cluster C, sends a SAFE message to its parent in the tree spanning C after it becomes safe and it receives SAFE messages from all its children in this tree.
2. When the root is safe and it receives SAFE messages from all its children, it broadcasts a GOAHEAD message to its cluster.

3. A root in a cluster C generates the next pulse after receiving a GOAHEAD from all its parents and SAFE from all its children in C. A non-root node generates the next pulse after receiving a GOAHEAD from all its parents.

4.2.3. Synchronizer γ_1: correctness

Theorem 4.3. *A node v generates the next pulse of the original algorithm only after all its neighbors are safe in the current pulse.*

Proof. A non-root node v generates the next pulse upon receiving GOAHEAD messages from all its parents in the clusters it belongs to. If v is a root of a cluster C, it has also to receive SAFE from all its children in C. In both cases, any node which belongs to one of the clusters that v belongs to is *safe*. Since, by Lemma 4.2, v has a common cluster with each of its neighbors, all its neighbors are *safe*. □

4.2.4. Synchronizer γ_1: overhead per pulse

Each node in a cluster C, except the root, sends one SAFE message and receives one GOAHEAD message per pulse. Thus, the overhead in messages per pulse is less than $2\sum_C |C|$, where the sum is taken over all clusters C. A cluster C with t_C tenant nodes and l_C nodes in its last layer satisfies $|C| = l_C + t_C$ and $l_C < (K-1)\cdot t_C$ (see definition of the last layer in Section 4.1). Also, by Lemma 4.1, a node is a tenant in at most one cluster. Thus we get that the sum of the sizes of the clusters satisfies: $\sum_C |C| = \sum_C (t_C + l_C) < \sum_C K \cdot t_C = K\sum_C t_C \leqslant K|V|$. Hence, the total number of control messages per pulse is less than $2K|V|$.

The overhead in time complexity in synchronizer γ_1 is at most $2\log_K |V|$, since the maximum height of a cluster is bounded by $\log_K |V|$, and the SAFE messages are sent from the leaves to the root, while the GOAHEAD messages are sent in opposite direction.

The memory requirement of γ_1 is O(1) per adjacent edge at a node. By Lemma 4.1 a node v can have children only in one cluster, say C_v. Hence, in order to maintain correct communications along the trees spanning each cluster, a node v only needs to distinguish three properties of adjacent nodes: v's children in C_v, v's parent in C_v, and v's parents in other clusters. This can be done by three bits per edge. Also, since a node v may be a parent of a neighbor u in at most one cluster, there is no need to send clusters identities with SAFE or GOAHEAD messages. Thus, constant message size is sufficient.

4.3. Variant 2: Synchronizer γ_2

4.3.1. Synchronizer γ_2: preprocessing

In this variant, the preprocessing phase guarantees that for each node v there is a cluster C which contains $N_1(v)$ (i.e. v and all its neighbors). To achieve this, we use the construction described in Section 4.1, with the following modifications: During the construction, a node v is marked as *explored by cluster C* when v and all its neighbors join cluster C; node v is *unexplored* if it is not explored by any cluster yet (initially all nodes are unexplored). Potential leaders are unexplored nodes.

Using the above definitions, the construction of a cluster C is done as follows: In iteration 0 the cluster leader, r, becomes the root of C, and then it initiates iteration 1. For $i \geqslant 1$, iteration i is split to two sub-iterations, as follows: In the first sub-iteration of iteration i, each node v in layer $i-1$ of C performs the following:

1. marks itself as a tenant of C;
2. if it is unexplored yet, then v adds to layer i of C all its neighbors which are not in C yet, and marks itself explored by C;
3. If it is already explored, then v adds to layer i of C all its unexplored neighbors.

Each node v which was added to C in the first sub-iteration and is unexplored yet, performs the second sub-iteration, in which it does the following:

1. Adds to C all its neighbors which are not in C yet, and
2. marks itself explored by C;

Finally, the number of nodes which joined layer i of C is reported to the root in the standard convergecast manner. If this number is at least $K-1$ times the number of nodes in all previous layers, then the root instructs the nodes in layer i to initiate iteration $i+1$; otherwise it announces the termination of the construction of C.

Note that in this construction, the tree spanning the cluster C is not necessarily a BFS tree of the subgraph $G(C)$ of G induced by the nodes of C, i.e. the length of the tree path from the root r to a certain node v is not necessarily equal to $dist_{G(C)}(r,v)$.

Lemma 4.4. *The construction terminates, and when it terminates, for each node $v \in V$, there is a cluster C such that v is explored by C, hence v and all its neighbors belong to C.*

Proof. The termination proof is identical to the proof of Lemma 4.2. The second part follows from the fact that when the algorithm terminates all nodes are explored. □

Before analyzing γ_2 construction, we prove some properties that will be needed later.

Lemma 4.5. *Let v be a node which is explored by or a tenant in cluster C, but it is not the root of C. Then v's parent in C is a tenant of C.*

Proof. If v became a tenant in or was explored by C during the first sub-iteration of iteration i, then v's parent is in layer j of C, for $j \leqslant i-1$. If v was explored in the second sub-iteration of iteration i, then its parent is in layer $i-1$ of C. In both cases, v's parent is not in the last layer of C, and hence it is a tenant of C. □

Claim 4.6. *Assume that the construction of cluster C was terminated, and let v be a tenant of C. Then v and all its neighbors are explored.*

Proof. Assume that v is in layer i of C. Since v is a tenant of C, v initiated iteration $i+1$ during the construction of C. The first sub-iteration guarantees that v becomes explored, and that all its unexplored neighbors join C. The second sub-iteration guarantees that all these neighbors become explored. □

Lemma 4.7. *A node can be a tenant in at most one cluster.*

Proof. Let C be the first cluster in which a node v becomes a tenant. It suffices to show that v does not join any cluster C' which is constructed after C. Let C' be such a cluster, and let r be the root of C'. Then r is unexplored when construction of C' starts, hence $r \neq v$. We now prove that v does not join C' in the ith iteration of its construction, for $i \geqslant 1$. Since, by Claim 4.6, v and all its neighbors were explored before C' was constructed, v does not join C' in the first sub-iteration of iteration i. Finally, since all of v's neighbors are already explored, v does not join C' also in the second sub-iteration of that iteration. □

Lemma 4.8. *For any cluster C, the height of the tree spanning C is at most* $2\log_K |V|$.

Proof. At every iteration except the last, the number of nodes in the cluster is multiplied by at least K. Therefore, a cluster creation takes $\log_K |V|$ iterations at the most. At every iteration the tree height is increased by at most 2. □

4.3.2. Synchronizer γ_2: preprocessing complexity

For each edge (u, v), u sends v one message when u is explored, and possibly another message when u becomes a tenant. Each such message is acknowledged by v. This gives us $O(|E|)$ messages which are sent for adding vertices to clusters. In addition, in each iteration, each node in C sends also $O(1)$ messages along tree edges, in order to count the number of nodes that joined C, to initiate the next sub-iteration, or to announce termination. Since in each iteration except the last one, the number of nodes in C is multiplied by a constant K larger than 1, the total number of such messages sent during the construction of C is $O(|C|)$, which gives total of $O(\sum_C |C|)$ when summing up over all the clusters. Since a node u can be a child of a node v in at most two clusters – at most one cluster in which v is a tenant and one cluster in which v was explored – we have that each edge can belong to at most four spanning trees of clusters.[6] Hence, the volume $\sum_C |C|$ is bounded by $O(|E|)$. Thus, the communication complexity of the preprocessing phase is $O(|E|)$.

To bound the bit complexity of the preprocessing phase, we note that the only messages that have more than a constant size are the messages that count the number of nodes in a given layer. These messages are sent once by each node when it is explored, and once in each iteration by tenants of the constructed cluster. Since the number of tenant nodes in a cluster is multiplied in each iteration by a constant $K > 1$, the sum of counting messages sent by tenants nodes during the construction of a cluster C with t_C tenant nodes is $O(t_C)$. By Lemma 4.7 the sum $\sum_C t_C = O(|V|)$. Thus, we

[6] Using Claim 4.6, one can improve this somewhat, and show that for each edge (u, v)), the *parent–child* relationship can exist at most twice in one direction and once in the other.

have $O(|V|)$ such messages, each of size $O(\log |V|)$. Hence, the bit complexity of the preprocessing phase, assuming the network has a leader, is $O(|E| + |V| \log |V|)$.

Constructing a cluster C with t_C tenant nodes is done by at most $\log_K t_C + 1$ iterations. Each iteration takes at most $2(2 \log_k t_C + 2)$ time units. Thus, constructing C takes $O(\log_K^2 t_C)$ time units. By the fact that $\log_K^2 t_C = O(t_C)$ and by Lemma 4.7, we get that the number of time units for constructing all the clusters in the network, is $O(|V|)$. Using the method of Section 3, the time complexity of the next cluster leader election is also $O(|V|)$.

Thus, the communication, bit and time complexities of γ_2 construction in a network with a leader are $O(|E|), O(|E| + |V| \log |V|)$ and $O(|V|)$, respectively.

4.3.3. *Synchronizer γ_2: operation*

The simulation of a pulse by γ_2 is done in two phases, as in γ_1, with the following changes: the GOAHEAD messages are forwarded from a node u in a cluster C to its child v in C, only if v is a tenant of C, or it is explored by C. Note that by Lemma 4.5, these messages are sent only on the tree in which u is a tenant. A non-root node generates the next pulse upon receiving the GOAHEAD message on the (single) cluster by which it was explored; a root generates the next pulse upon receiving SAFE messages from all its children.

4.3.4. *Synchronizer γ_2: correctness*

For γ_2 to be correct, it suffices that each node v receives a GOAHEAD message iff all its neighbors are safe. By Lemma 4.4 there is a cluster C in which v is explored, and thus all v's neighbors belong to C. It is easy to see that the root of C will receive SAFE message from all its children iff all the nodes in C are safe, and in particular all v's neighbors are safe. Once the root receives SAFE messages from all its children, it broadcasts a GOAHEAD message to its cluster, and this message is forwarded by every node in C to all its children in C which are either tenants in C or explored by C. Since, by Lemma 4.5, a node v which is not the root of C is a tenant or explored in C only if its parent is a tenant in C, each node explored in C will receive a GOAHEAD message from its parent in C.

4.3.5. *Synchronizer γ_2: efficient implementation*

In this section we show that synchronizer γ_2 can be maintained using a constant memory per edge and a constant message size.

For brevity, in the discussion below, we identify a cluster C with the tree spanning it. Since a node v may participate in more than a constant number of clusters, v needs to distinguish between the different (trees spanning the) clusters it belongs to. Thus, a straightforward implementation of the operation of γ_2, as described in Section 4.3.3, may require $\Omega(\log |V|)$ bits per tree edge, in order to keep the identity of the cluster (or clusters) which uses this edge, and each SAFE or GOAHEAD message should also carry the identity of the cluster on which it is sent.

In our construction, though, a node v can be a non-leaf node in at most two clusters: The cluster in which v is explored, which we call $C_E(v)$, and the cluster in which v is a tenant, called $C_T(v)$ (observe that both can be the same cluster). Thus, v only needs to maintain the following information for each adjacent edge (v,u). First, whether this edge is an outgoing edge $(v \rightarrow u)$ in at least one tree, and if it is – to which of the trees $C_E(v), C_T(v), C_E(u)$ and $C_T(u)$ it belongs. Similarly, if it is an incoming edge $(u \rightarrow v)$ in at least one tree, and if so – to which of the above four trees it belongs. Altogether, eight bits per edge at a node are sufficient.

We now show that constant message size suffices to implement γ_2. Using the above structure, a SAFE message sent by a node v to its parent u needs to carry only the information whether it is sent on $C_E(u)$ or $C_T(u)$ (in fact, this information should be sent only in the case where both trees use the edge $(u \rightarrow v)$). This enables each vertex u to forward a SAFE message to its parent in one of these two trees only when u and all its children (if any) on that tree are safe. Similarly, since the information kept in each node u enables u to identify its children in $C_T(u)$ which are either explored by or tenants in $C_T(u)$, u needs no additional information in order to forward the GOAHEAD messages it receives from its parent in $C_T(u)$ only to these nodes, as required.

4.3.6. *Synchronizer γ_2: overhead per pulse*

Since, by Lemma 4.8, the height of each cluster is at most $2\log_K |V|$, the overhead in time per pulse is at the most $4\log_K |V|$. In each cluster C, all nodes except the root send one SAFE message, which gives total of $\sum_C |C| \leqslant K|V|$ messages per pulse. In addition, each node receives one GOAHEAD message in the cluster in which it is explored, and possibly another such messages in the cluster in which it is tenant (if any). This gives additional less than $2|V|$ messages. Thus, the total communication overhead per pulse is less than $(K+2)|V|$.

As mentioned in Section 4.3.5, the memory requirement of γ_2 is O(1) per adjacent edge at a node, and the size of the synchronization messages is also bounded by a constant.

5. A cover-coarsening algorithm

The preprocessing algorithms discussed in Section 4 can be viewed as special cases of the cover-coarsening algorithm, specified as follows:

Input: A number $K>1$, and a *source cover* $\mathscr{S} = \{S_1, S_2, \ldots, S_m\}$ of a graph $G = (V,E)$, where each cluster $S_i \in \mathscr{S}$ is given by a tree which spans its nodes, rooted at the cluster's leader. The diameter of the tree is bounded by some global constant $d_{\mathscr{S}}$.

Output: A *target cover* $\mathscr{T} = \{T_1, T_2, \ldots, T_l\}$, where each cluster T_i is represented by a tree spanning it, satisfying: (a) each cluster $S \in \mathscr{S}$ is included in some cluster $T \in \mathscr{T}$, (b) $Vol(\mathscr{T}) \leqslant K|V|$, and (c) the diameter of each cluster $T_i \in \mathscr{T}$ is bounded by $d_{\mathscr{T}} = (2\log_K(|V|)+1)d_{\mathscr{S}}$.

We present here a cover-coarsening algorithm, named ζ. Assuming that the network has a leader, the communication and time complexities of ζ are $O(Vol(\mathscr{S}))$ and $O(\min(Vol(\mathscr{S}), d_{\mathscr{S}} \cdot |V|)$, respectively. Our algorithm improves a similar algorithm given in [15] in several aspects: It introduces a novel technique for merging clusters, thus avoid possible bad scenarios in the algorithm presented there, as described in Section 5.2, and it is simpler and more efficient in its complexity measures. Algorithms related to ζ are also studied in [5, 14]: Peleg [14] presents a centralized cover-coarsening algorithm. In [5], some algorithms for the related task of coarsening a cover comprising of the m neighborhoods of every $v \in V$ for $1 \leqslant m \leqslant \log |V|$ are given. Specifically, it presents two *synchronous* algorithms, called SYNC-AV-COVER and SYNC-MAX-COVER, and one asynchronous algorithm, called ASYNC-MAX-COVER. Our algorithm ζ is related to an asynchronous version of SYNC-AV-COVER, which is not presented there formally (though it is used implicitly in the construction of ASYNC-MAX-COVER). This algorithm differs from ours in several points: The input cover there is the set of m-neighborhoods of all vertices, the output cover guarantees that the spanning tree of each cluster is a BFS tree on the induced subgraph, and the height of such a tree may be $4 \log_k |V|$; its communication, time, bit and memory complexities are $O(|E| + |V| \log_K^3 |V| \log |V|)$, $O(K|V| \log_K^3 |V| \log |V|)$, $O(K|V| \log |V|$ and $O(|V|)$, respectively.

Note that ζ can serve as a preprocessing algorithm for γ_1, by letting the input cover $\mathscr{S}$ be the set of all edges in E (thus $Vol(\mathscr{S}) = 2|E|$ and $d_{\mathscr{S}} = 1$), and for γ_2, by letting $\mathscr{S}$ include for each node v the cluster composed of v and its neighbors (thus $Vol(\mathscr{S}) = \sum_{v \in V} (d(v) + 1) = |V| + 2|E|$ and $d_{\mathscr{S}} = 2$). We note, however, that the bit complexity of ζ is higher than that of the preprocessing algorithms for γ_1 and γ_2 presented in Section 4, since it sends cluster identities in its messages. Also, ζ does not imply synchronizers which have constant messages size and constant memory per edge, like γ_1 and γ_2.

5.1. Description of the algorithm

Algorithm ζ has the structure described in Section 3, only that now S-clusters are treated as nodes, as we describe below.

A target cluster T is constructed in iterations, where the nodes added to T in iteration i are denoted as layer i of T. We denote by $T[i]$ the cluster T constructed during the first i iterations. When there is no ambiguity, we will identify $T[i]$ with its spanning tree. In iteration 0, some cluster $S \in \mathscr{S}$ is eliminated from $\mathscr{S}$ and becomes layer 0 of T. In iteration $i > 0$, each cluster $S \in \mathscr{S}$ which intersects $T[i-1]$ is eliminated from $\mathscr{S}$ and merged into $T[i-1]$ to form $T[i]$. The tree spanning $T[i-1]$ is extended to span $T[i]$ in a way that guarantees that its height is increased by at most $d_{\mathscr{S}}$. At the end of iteration i, the nodes at layer i are counted, and if their number is at least $K - 1$ times larger than the number of nodes in $T[i-1]$, iteration $i+1$ starts; otherwise the construction of T is terminated. By arguments similar to the ones given in Section 4, we show that a node can be a tenant in at most one cluster, and (hence) the sum of the cardinalities of the last layers of the clusters in $\mathscr{T}$ is less than $(K-1)|V|$. Thus, $Vol(\mathscr{T}) \leqslant K|V|$. Similarly, it can be shown that the number of iterations is bounded by $\log_K |V|$, and we prove later that the height of the tree spanning every output cluster T is bounded by $d_{\mathscr{S}} \log_K |V|$.

The construction of a target cluster T starts in iteration 0, when a cluster $S \in \mathscr{S}$ is eliminated from $\mathscr{S}$ and becomes layer 0 of T, and its root becomes the root of T. For $i>0$, iteration i starts when the root of $T[i-1]$ broadcasts a message on (the tree spanning) $T[i-1]$, to announce the initiation of iteration i. A node v in $T[i-1]$ which receives this message becomes a tenant of T. If v belongs to one or more source clusters, it sends S-T *expansion messages* to all its neighbors in each such cluster S, in order to merge S into T. During this merging process, the original parent–child relation in cluster S is ignored, and a new parent–child relation is created, as follows. A node v in $S \backslash T[i-1]$ which receives an S-T expansion message for the first time, marks the sender of this message as its new parent in S. In addition, each such node v also *establishes* a parent in T. Thus, the following structures are maintained during iteration i.

- For each source cluster S which intersects $T[i-1]$, a digraph $\vec{S}$ induced by the set of edges $E(\vec{S}) = \{(u,v) \mid u$ marked v as its parent in S during iteration $i\}$.
- A digraph $\vec{T}$ induced by the set of edges $E(\vec{T}) = \{(u,v) \mid v$ is the last node which u established as its parent in T during iteration $i\}$.

Each S-T expansion message carries the identities of the source cluster S and the target cluster T, and a variable d, initiated to zero, that indicates the distance of the sender from $T[i-1]$. For each node v in layer i of T there is a unique source cluster S s.t. v is *explored in T by S*. Each S-T expansion message received by a node v from its neighbor u is eventually acknowledged by an S-T *acknowledgment* message. This message contains a count field c that indicates the number of nodes explored in T by S in the sub-tree of $\vec{S}$ rooted at v, and a bit f which is set to 1 if v establishes u as its parent in T. v may later replace its parent in T by another node, as we describe below.

A node v which receives from a neighbor u an S-T expansion message with distance field d acts as follows:

Case 1: v was not yet explored in T. In this case v will:
1. mark itself "explored in T by S",
2. mark u as its parent in T,
3. mark u as its parent in S,
4. increment d by one,
5. set $d(T)$, its distance from $T[i-1]$, to d,
6. send S-T expansion messages to all its other neighbors in S.

Case 2: $v \notin T[i-1]$, and is already explored in T:
- If this is the first S-T expansion message received by v, then v will
  1. mark u as its parent in S,
  2. increment d by one,
  3. If $d < d(T)$ then it marks u as its parent in T, and sets $d(T)$ to d,

4. If $d > d(T)$ then it sets d to $d(T)$,
5. send S-T expansion messages to all its other neighbors in S.

- Else, v will send back to u an S-T acknowledgment with $c = 0$ and $f = 0$.

Case 3: $v \in T[i-1]$: v sends u an S-T acknowledgment with $c = 0$ and $f = 0$.

A node v which has sent or received S-T expansion messages during iteration i, waits until it receives S-T acknowledgments for all the S-T expansion messages it sent. At this point, it stops being a member of cluster S. If it has a parent in S, say u, then it also sends an S-T acknowledgment message to u. The count field c and the parent indicator bit f of this message are determined as follows:

1. c is first set to be the sum of the c fields in all S-T acknowledgments messages received.
2. If v was explored in T by S, c is incremented by 1.
3. If u is marked as v's parent in T, then the f bit is set to 1, else it is set to 0.

If the f-bit sent by v was set to 1, meaning that v establishes u as its parent in T. It might be the case that v had previously established a different parent in T, say w. In this case, v has to send w a *delete parent* message announcing that it is no longer its parent in T.

The number of nodes in layer i is computed by the nodes in $T[i-1]$ as follows. A node v which is a leaf in $T[i-1]$, sends a *report*(c) message to its parent in T after receiving acknowledgments for all the expansion messages it sent (if any); the value of c is the sum of the c fields in all the acknowledgment messages received. These *report* messages are forwarded towards the root by internal nodes of $T[i-1]$ which receive *report* messages from all their children in $T[i-1]$ and acknowledgment messages for all expansion messages sent by them. Each such node updates the count c to be the sum of the c values it received. The sum of the c values received by the root is the number of nodes in layer i.

5.2. An example

Some of the main ingredients of Algorithm ζ are illustrated by the scenario described below, and depicted in Fig. 3.

Suppose that nodes u_1 and u_2 were added to the target cluster T in iteration $i-1$. Assume also that u_1 and u_2 belong to source clusters S_1 and S_2, respectively (see Fig. 3). Now let the ith iteration start, and let u_1 initiate an S_1-T expansion message; this message eventually reaches v. v now sets $d(T)$ to 5, sets w to be its parent in T (and in S_1), increments d by one and sends the message to x, which is a leaf in S_1. x now sets $d(T)$ to 6, marks v as its parent in T, and then sends an S_1-T acknowledgment message to v. In this message, x sets c to 1, since x was explored in T by S_1, and v is x's parent in S_1; x also sets the f bit to 1, to establish v as its parent in T. When v receives this S_1-T acknowledgment message, it sets the c field to 2 (since the sum of all c fields received by v is 1, v was explored in T by S_1, and w is v's parent in S_1) and the f bit to 1 (since w is marked as v's parent in T). Then, v forwards this message to w, its parent in S_1.

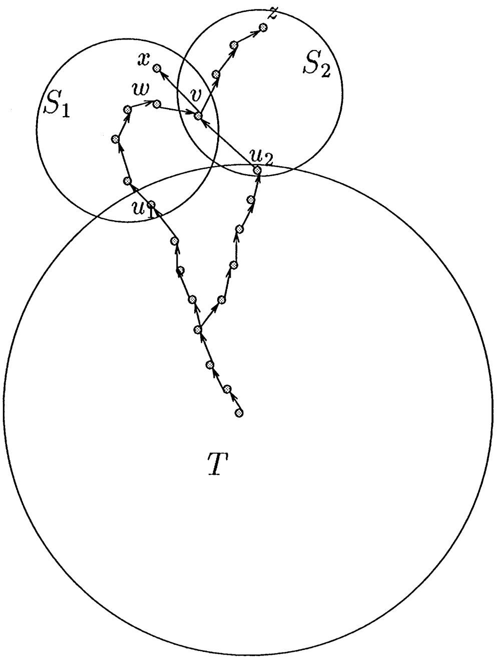

Fig. 3. An illustration of the cover-coarsening algorithm.

Later on, v receives an S_2-T expansion message from u_2. It then updates d to 1, and marks u_2 as its parent in T. Then it forwards this message on the tree spanning S_2. This message eventually arrives z, who then sends back an S_2-T acknowledgment. This message will eventually arrives v, with $c=3$ (which is the number of descendants of v on the path ending in z). At this point, v sends u_2 an S_2-T acknowledgment message with $c=3$ (i.e., v does not increase c) and $f=1$, thus establishing u_2 as its parent in T. v will also send a *delete parent* message to w, informing w that it is no longer its parent in T. The tree edges that were added to T during this process are all the edges in S_1 and S_2, except the edge $(w \rightarrow u)$.

There are two important points that this example illustrates: first, that the change of parent is essential in order to keep the incremental addition to the diameter of T bounded by $d_{\mathscr{S}}$. Second, that while counting the nodes added to T in iteration i, each node is reported on the S cluster in which it was explored, which guarantees that it is counted exactly once (in the example above: v was counted by w, its parent in the S_1-cluster). It appears that these two properties are not maintained by the algorithm in [15]; specifically, in the scenario described above, this algorithm will fail to count the three descendants of v in S_2, and will leave w as the parent of v in T, thus setting the distance of z from $T[i-1]$ to 8 – larger than the diameters of both S_1 and S_2; in fact, this example can be expanded to show that this algorithm may increase the diameter of T by $\theta(|V|)$ during a single phase.

5.3. Correctness proof

In this section we prove that at the end of iteration i, $T[i]$ is a tree that spans all the nodes in source clusters that intersects $T[i-1]$, its height is incremented by at most $d_{\mathcal{S}}$, and the root of T computes correctly the number of nodes that were added to T in this iteration.

Claim 5.1. *Let v be a node not in $T[i-1]$. Then if v belongs to some source cluster S which intersects $T[i-1]$, v will receive an S-T expansion message during iteration i.*

Proof. Let v be in a cluster S as in the statement of the claim, and consider a path $(u, u_1, \ldots, u_{k-1}, v)$ in the cluster S which connects a vertex $u \in S \cap T[i-1]$ and v. When u receives a message announcing the initiation of iteration i, it will send an S-T expansion message to u_1. Also, for $1 \leqslant j \leqslant k-1$, u_j forwards the first S-T expansion message it receives to u_{j+1}. Thus, by an elementary induction, an S-T expansion message reaches v. □

Lemma 5.2. *A node v not in $T[i-1]$ will be explored in T during iteration i iff it belongs to some source cluster S which intersects $T[i-1]$.*

Proof. Let S be a source cluster intersecting $T[i-1]$ which contains v. Then by Claim 5.1, v receives an S-T expansion message. If v was not explored before receiving this message, it will be explored upon receiving it. From the fact that S-T expansion messages are initiated only by nodes in source clusters which intersect $T[i-1]$, and are forwarded only on edges of these clusters, nodes which do not belong to such clusters will not receive expansion messages during iteration i, and hence are not explored during this iteration. □

Claim 5.3. *For each cluster S which intersects $T[i-1]$, the graph $\vec{S}$ is a directed forest, whose roots are the nodes in $S \cap T[i-1]$.*

Proof. Each node $v \in \vec{S}$ which is not in $T[i-1]$ has exactly one parent in S – the first node from which v received an S-T expansion message; thus such a node v cannot be a root. Each node $v \in S \cap T[i-1]$ has no parent in S, thus such a node must be a root. Finally, u is the parent of v in $\vec{S}$ only if it is a root or it received an S-T expansion message before v did. Thus, $\vec{S}$ contains no cycles. The claim follows. □

We define a *string of messages* $(m_0, m_1, \ldots, m_k)$ as a sequence of messages, generated at nodes $(u_0, u_1, \ldots, u_k)$, respectively, where for $1 \leqslant i \leqslant k$, message m_i generated at a node u_i as a consequence of receiving message m_{i-1} from a node u_{i-1}.

Claim 5.4. *A string of S-T expansion messages is finite.*

Proof. Since S-T expansion messages are sent only to nodes in S, and $|S|$ is finite, a string of S-T expansion messages eventually arrives at a node u_k, where either u_k already received an S-T expansion message, or $u_k \in T[i-1]$, or u_k is a leaf of S. In all these cases u_k does not forward the S-T expansion message, and acknowledges immediately. □

Lemma 5.5. *Iteration i terminates, and when it terminates, $T[i]$ is a directed tree.*

Proof. As proved in Claim 5.4, all strings of S-T expansion messages are finite, and the last node in each such string acknowledges immediately. The other nodes in the string of S-T expansion messages form a path in the directed forest $\vec{S}$, on which the S-T acknowledgments converge to the originators of these messages in the standard convergecast process.

The second part of the lemma is proved by induction on the number of iterations. The base $i=0$ is clear since $T[0]$ is a source cluster S, which by the assumption is given by its rooted spanning tree.

We now assume that the lemma holds for $T[i-1]$ and prove it for $T[i]$. For this, observe that $T[i]$ is the graph induced by the set of directed edges $E' = \{(u,v) \mid v$ is the established parent of u in T at the end of iteration $i\}$. Recall that $\vec{T}$ is the digraph induced by the set of edges $E(\vec{T}) = \{(u,v) \mid v$ is the last node which u established as its parent in T during iteration $i\}$. Then $T[i]$ is obtained by adding to $T[i-1]$ the edges in $\vec{T}$ at the end of iteration i. Hence, it is enough to show that at the end of iteration i, $\vec{T}$ is a directed forest whose roots are nodes in $T[i-1]$, and all its other nodes are in $T[i] \backslash T[i-1]$. The proof is similar to that of Claim 5.3, using the following observations:

1. Every node in $T[i] \backslash T[i-1]$ has exactly one established parent in $\vec{T}$.
2. Every node in $\vec{T} \cap T[i-1]$ has no established parent in $\vec{T}$.
3. Let d_u denote the value of the field $d(T)$ at node u at the end of iteration i. Then for every u and v s.t. $(u,v) \in \vec{T}$, $d_u > d_v$, thus there are no circuits in $\vec{T}$. □

Lemma 5.6. *At each iteration, the height of T is increased by at most $d_{\mathscr{S}}$.*

Proof. By Lemma 5.5, $T[i]$ is a tree. Thus, we have to show that for each node v in layer i of T, the length l_v of the shortest path in $T[i]$ connecting v to some node $u \in T[i-1]$ is bounded by $d_{\mathscr{S}}$. Let $d_v(T)$ denote the value of the variable $d(T)$ at node v, and let d_v be the value of $d_v(T)$ at the end of iteration i. We prove that $l_v \leqslant d_v \leqslant d_{\mathscr{S}}$.

Let S be a source cluster by which v was explored. Let $p = (u_0, u_1, \ldots, u_{k-1}, v)$ be the path in S along which a string of S-T expansion messages that explored v was sent (note that p is a path in S, hence $k \leqslant d_{\mathscr{S}}$). For $0 \leqslant i < K$, when a node u_i forwards an expansion message to u_{i+1}, it sets the d field in this message to $d_{u_i}(T)+1$. Hence, when u_{i+1} receives this message it sets the value of $d_{u_{i+1}}(T)$ to $\min(d_{u_i}(T)+1, d_{u_{i+1}}(T))$. Moreover, the value of $d_u(T)$, once set, can only decrease. Hence, when v is explored, $d_v(T)$ is at most $k \leqslant d_{\mathscr{S}}$, and hence $d_v \leqslant k \leqslant d_{\mathscr{S}}$.

Next we prove by induction on l_v that for each node v, $l_v \leqslant d_v$. If $l_v = 1$, then v has a neighbor u in $T[i-1]$ such that (u,v) is an edge in a source cluster S, and hence $d_v = 1$. Assume the claim holds for $l_v < m$ and we prove for $l_v = m$. Let w be the parent of v in T. Then $l_w = m-1$, and w is the last node which v establishes as a parent in T during iteration i. When v marked w as its parent in T, upon receiving an S-T expansion message from w, it set $d_v(T)$ to the value d in that message, which was previously set to $d_w(T)+1$. Denote this value by $d^\star$. Since v did not establish a different parent later, the final value of $d_v(T)$, d_v, is equal to $d^\star$. Since the value of $d_w(T)$ could only decrease, we have that at the end of iteration i, $d_v = d^\star \geqslant d_w + 1 \geqslant l_w + 1 = l_v$. □

Corollary 5.7. *For all $T \in \mathscr{T}$ the height of T is at most $d_{\mathscr{S}} \log_K |V|$.*

Lemma 5.8. *At the end of iteration i, the root computes the number of nodes at layer i.*

Proof. First we prove that the c field in an S-T acknowledgment message sent by a node v to its parent in $\vec{S}$ contains exactly the number of nodes explored by S in the subtree of $\vec{S}$ rooted at v. We prove this by induction on h, the height of this subtree. For $h=0$, v is a leaf; if v was explored in T by S then the c field is set to 1, otherwise it is set to 0. We now assume that the claim holds for $k<h$ and prove it for a node v which is a root of a subtree of $\vec{S}$ of height h: v sends an S-T acknowledgment after it receives acknowledgment messages for all the S-T expansion messages it sent. The c fields in the acknowledgment messages from nodes which are not its children in $\vec{S}$ is 0. The sum of the c fields in the acknowledgment messages from its children in $\vec{S}$, by the induction assumption, equals the number of nodes explored in T by S in the subtree of $\vec{S}$ rooted at v, excluding v itself. If v was explored by S, it increases this sum by one, otherwise, it leaves it as is. Hence, the claim holds also for v.

By the said above, each node v in $T[i-1]$ eventually receives for each S-T message it sent to a neighbor u, the number of nodes explored in T by S in the sub-tree of $\vec{S}$ rooted at u. Since $\vec{S}$ is a forest, and since each node in layer i is explored exactly once, the sum of these numbers taken over all expansion messages sent in iteration i, is the number of nodes in layer i. These numbers are now summed up and forwarded towards the root of $T[i-1]$ in a standard way, and hence the root of T will eventually compute their sum. □

5.4. Complexity analysis

Claim 5.9. *A node can be a tenant in at most one target cluster T.*

Proof. A node v which becomes a tenant in T during iteration i, sends in this iteration an S-T expansion messages for each source cluster S it belongs to. By Lemma 5.5, the iteration terminates, meaning that all expansion messages sent by v were acknowledged.

This means that v stopped being a member in all source clusters it belonged to, and hence will not join any other target cluster. □

The messages sent by algorithm ζ are divided into four classes:

- **expansion** and **acknowledgments** messages are sent only on tree edges of source clusters, and at most once in each direction of such an edge. Hence, there are $O(Vol(\mathcal{S}))$ such messages.
- **delete parent** messages are sent at most once by each node for each source cluster it belongs to. Thus, the number of these messages is $O(Vol(\mathcal{S}))$.
- **counting** and **start iteration** messages are sent during iteration i only on tree edges of $T[i-1]$, and only once in each direction. In particular, these messages are sent and received only by tenant nodes. nodes. By the fact that cluster T's cardinality is multiplied in each iteration by at least $K \geqslant 2$, the number of such messages sent during construction of a cluster T with t_T tenant nodes is $O(t_T)$. Summing up for the whole network, and by claim 5.9 we get $O(|V|)$ such messages.
- **searching for new cluster leader** messages, as explained in Section 3 is done by $O(|V|)$ messages.

In total, we get communication complexity of $O(Vol(\mathcal{S}))$. In case the graph does not have a leader, then constructing a spanning tree of the graph will add $O(|V|\log|V| + |E|)$ to the total communication complexity.

The time-consuming parts in ζ are the broadcast and convergecast messages sent at the beginning and the end of each iteration of the clusters construction. By arguments similar to these in the complexity analysis of γ_2 preprocessing phase in Section 4.3.2, and by Claim 5.9, we get that the time complexity of ζ is $O(|V| \cdot d_{\mathcal{S}})$.

Note that in the case where $d_{\mathcal{S}}$ is constant and $Vol(\mathcal{S}) = O(|E|)$, as is the case for the preprocessing needed for γ_1, γ_2, the resulting communication and time complexities of ζ, assuming the network has a leader, are $O(|E|)$ and $O(|V|)$, respectively.

6. Concluding remarks

This paper presented a simple and efficient technique for constructing sparse decompositions of graphs, which are useful for synchronizing distributed networks. Specific implementation of this technique were used to construct synchronizers γ_1 and γ_2, which are simple and efficient variants of existing synchronizers. Synchronizers γ_1 and γ_2 and their preprocessing algorithms enable the performing of a *Breadth First Search* in an asynchronous network, with no pre-established synchronizer, in total communication and time complexities of $O(K|V|D + |E| + |V|\log|V|)$ and $O(|V| + D\log_K|V|)$, respectively, instead of $O(|V|^2 + K|V|D)$ and $O(|V|\log_K|V|)$ by [2] (Where D is the diameter of the network) with constant memory per edge and constant messages size. In [2], the construction phase dominates the communication and time complexities of the whole algorithm, while due to the efficiency of our preprocessing phase, in our algorithm the total complexity is dominated by simulating the synchronous algorithm itself. Thus, the complexity measures above are implied by the facts that a synchronous *BFS* in a network with diameter D requires at most D pulses, and that the communication and time overhead per pulse of our synchronizers are $O(K|V|)$ and $O(|\log_K |V|)$, respectively. time complexity networks

An interesting question related to our construction is whether there are truly parallel constructions of the sparse covers needed for our synchronizers, i.e. constructions which require polylogarithmic, or even just sub-linear, time. In particular, is there a (possibly randomized) algorithm similar to the one presented in [13] for constructing such covers.

Acknowledgements

We would like to thank Hagit Attiya for bringing reference [15] to our attention, and Oded Goldreich, Shay Kutten and David Peleg for some helpful discussions. We would also like to thank two anonymous referees for their many helpful comments.

References

[1] Y. Afek, M. Ricklin, Sparser: a paradigm for running distributed alogorthms, J. Algorithms 14 (1993) 316–328.
[2] B. Awerbuch, Complexity of network synchronization, J. ACM 32(4) (1985) 804–825.
[3] B. Awerbuch, Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems, in: Proc. 19th ACM Symp. on Theory of Computing, 1987, pp. 230–240.
[4] B. Awerbuch, A.V. Goldberg, M. Luby, S.A. Plotkin, Network decomposition and locality in distributed computation, in: 30th Ann. Symp. on Foundations of Computer Science, 30 October–1 November 1989, pp. 364–369.
[5] B. Awerbuch, D. Peleg, Efficient distributed construction of sparse covers, Tech. Report CS90-17, Weizmann Institute, July 1990.
[6] B. Awerbuch, D. Peleg, Network synchronization with polylogarithmic overhead, in: Proc. 31st IEEE Symp. on Foundations of Computer Science, 1990, pp. 514–522.
[7] B. Awerbuch, D. Peleg, Sparse partitions, in: Proc. 31st IEEE Symp. on Foundations of Computer Science, 1990, pp. 514–522.
[8] B. Awerbuch, D. Peleg, Routing with polynomial communication-space trade-off, SIAM J. Discrete Math. 5 (1992) 151–162.
[9] B. Awerbuch, D. Peleg, Online tracking of mobile usuers, J. ACM 42 (5) (1995) 1021–1058.
[10] S. Even, Graph Algorithms, Computer Science Press, Woodland Hills, CA, 1979.
[11] M. Faloutsos, M. Molle, Optimal distributed algorithms for minimum spanning trees, revisited, in: Proc. 14 ACM Symp. Principles of Distributed Computing, 1995, pp. 231–237.
[12] R.G. Gallager, P.A. Humblet, P.M. Spira, A distributed algorithm for minimum weight spanning trees, ACM Trans. Program. Lang. Syst. 5(1) (1983) 66–77.
[13] N. Linial, M. Saks, Low diameter graph decompositions, Combinatorica 13 (1993) 441–454.
[14] D. Peleg, Sparse graph partitions, Tech. Report CS89-01, Weizmann Institute, February 1989.
[15] L. Shabtay, A. Segall, Low complexity network synchronization, in: Proc. 8th Internat. Workshop on Distributed Algorithms, 1994, pp. 223–237.