Low Complexity Network Synchronization*

Lior Shabtay and Adrian Segall

Dept. of Computer Science, Technion, Israel Institute of Technology
Haifa, Israel 32000
email : liors@cs.technion.ac.il, segall@cs.technion.ac.il

Abstract. Synchronizer γ is the best synchronizer known that works with any type of synchronous model and any network topology. This paper presents three new synchronizers: η_1, η_2 and θ. These synchronizers use sparse covers in order to operate and have the following advantages over synchronizer γ: (1) they are conceptually simpler, as only one convergecast and one broadcast processes are performed along each cluster spanning-tree between each two consecutive pulses, and no preferred links are needed for inter-cluster communication. (2) synchronizer η_2 uses half the communication complexity of synchronizer γ, while retaining the time complexity. (3) synchronizer θ uses half the time complexity of synchronizer γ, while retaining the communication complexity. (4) since there is no need to elect preferred links between neighboring clusters, the initialization process of these synchronizers is more efficient: it requires only $O(|V|\log|V| + |E|)$ messages.

Key words: distributed algorithms, networks, synchronization, sparse covers, communication and time complexities.

1 Introduction

This paper deals with distributed protocols in two network models: the *synchronous* model and the *asynchronous* model. In the *asynchronous model*, nodes perform operations only upon receiving a message from some neighbor or from the outside world. At that time, the node processes the message, performs local computations, and may send messages to some or all of its neighbors. All local actions are performed atomically. Messages sent by a node to any of its neighbors are received in a FIFO order within a finite undetermined time.

The *synchronous model* assumes that all link delays are bounded by some quantity referred to as a time unit. Pulses are generated synchronously at all nodes in the network at time unit intervals. Messages are sent only at pulse ticks, and thus arrive at the destination node before the next pulse. Operations are performed by a node only at the time of a pulse or when receiving a message. When a node receives a message, it processes the message and performs local computations. At the time of a pulse, the node may perform local computations

* Also to be presented on the workshop on distributed algorithms, 1994

and in addition, it may send messages to some or all of its neighbors. All local actions are performed atomically.

Synchronizers are tools for transforming protocols written for the synchronous model into protocols that run on an asynchronous network. The synchronous protocol will be referred to as the *original protocol*. The asynchronous protocol created by the synchronizer generates a sequence of pulses at each node. The pulses occur asynchronously at different nodes. At each pulse, the nodes perform the original-protocol pulse code and send messages which are identical to the original-protocol messages. In certain circumstances, slight changes in the pulse code and/or in messages are allowed (see [16]).

The methodology of synchronizers was introduced in [1], where three synchronizers were presented: the α synchronizer, with an overhead of $O(|E|)$ in communication complexity and $O(1)$ in time complexity per pulse, the β synchronizer with an overhead of $O(|V|)$ in communication and $O(D)$ in time complexity per pulse (when D is the diameter of the network), and the γ synchronizer, which enables trade-off between the above complexities.

In [1], it is shown that synchronizer γ achieves almost optimal tradeoff between the time and communication complexities. This is proved by showing lower bounds of $\Omega(\log_k |V|)$ time and $\Omega(k|V|)$ communication on this tradeoff, given $2 \leq k \leq |V|$. Later works have presented synchronizers with lower complexities designated for special cases: [15] shows an optimal synchronizer for hypercube networks; [5] presents a synchronizer with polylogarithmic overhead that works only for synchronous protocols in which a node may send messages at a pulse only if it has received messages sent at the former one; [10] discuss synchronizers for bounded-delay networks. Thus, synchronizer γ remains the best synchronizer that works for any synchronous protocol and any asynchronous network.

In [15], the authors suggest a technique for creating synchronizers by using graph spanners. given a t-spanner with m edges for a network $N = (V, E)$, the time complexity of the created synchronizer is t and its communication complexity is $t \times m$. In [9] and [4] $O(\log_k |V|)$-spanners with $O(k|V|)$ edges are presented, arguing that they can be used in order to create an efficient synchronizer. However, the communication complexity of the created synchronizers is $O(k|V| \log_k |V|)$, that is, $O(\log_k |V|)$ times larger than that of synchronizer γ.

In a protocol created by combining a synchronous protocol with a synchronizer, a node i is said to be *safe* with respect to pulse(n), if all original-protocol messages sent by node i when performing pulse(n) have already been received by the respective neighbors. In order for the node to know when they are safe, each node that receives an original-protocol message is required to send back an explicit acknowledgment message. Most synchronizers are based on the observation that a node i can perform pulse(n) when all its neighbors are safe with respect to pulse($n-1$).

Given a parameter k, the partition algorithm that initializes γ creates a partition of the network into clusters with cluster spanning-trees of hight $H_p = \log_k |V|$ at the most. The partition algorithm also elects a preferred link between each two neighboring clusters. The number of preferred links adds up to $(k-1)|V|$

at the most. The time complexity of the partition algorithm is $O(|V| \log_k |V|)$, its communication complexity is $O(k|V|^2)$.

In order to synchronize each pulse, γ uses the following protocol: after the pulse is performed by the nodes, SAFE messages are converged along the cluster spanning-trees from the leaves to the leader node of each cluster. Then, CLUSTER_SAFE messages are broadcast on the edges and preferred links of each cluster tree. Then, CLUSTER_READY messages are converged along each cluster tree, and AWAKE messages are broadcast along each tree in order to trigger the next pulse at the nodes.

The time complexity of γ is $4 \log_k |V| + 1$ per pulse. This is due to the fact that between each two consecutive pulses, two convergecast processes and two broadcast processes are performed along each of the cluster spanning-trees, and that messages are also sent along the preferred links. The communication complexity of γ is $4(|V|-1)+2(k-1)|V|$ messages per pulse, since between each two consecutive pulses, four messages are sent along each cluster spanning-tree edge and two messages are sent along each preferred link.

In this paper we present three new synchronizers, named η_1, η_2, and θ. These synchronizers make use of a sparse cover [14] [6], which is a collection of clusters such that each node is a member of at least one cluster (see Sec. 2). The initialization of these synchronizers is based on a cover creation algorithm presented in [14]. When given a parameter $2 \leq k \leq |V|$, this initialization algorithm creates clusters of radius $O(\log_k |V|)$, such that the average number of clusters each node participates in is k.

Synchronizers η_1, η_2 and θ are conceptually simpler than synchronizer γ: between each two consecutive pulses, they use only one convergecast of SAFE messages and one broadcast of AWAKE messages along each cluster spanning-tree. No preferred links are needed for inter-cluster communication.

The fact that no preferred links should be elected enables the initialization process of η_1, η_2 and θ be more efficient than the initialization process of γ. The communication complexity of a straightforward distributed implementation of the GV algorithm that is used for initializing synchronizers η_1, η_2, and θ is $O(|V|^2)$, while the communication complexity of the partition algorithm that initializes γ is $O(k|V|^2)$. Furthermore, in this paper we also present an improved version of the distributed GV algorithm, with communication complexity of $O(|V| \log_k |V| + |E|)$. The time complexity of the initialization algorithms discussed above is identical, namely $O(|V| \log_k |V|)$.

The time complexity of synchronizer η_1 is $4 \log_k |V| + 2$ per pulse, and its communication complexity is $2k|V|$ per pulse. Both complexities are identical to those of synchronizer γ. The time complexity of synchronizer η_2 is $4 \log_k |V| + 1$ per pulse, the same as the time complexity of synchronizer γ. The communication complexity of η_2 is $(k+2)|V|$ per pulse, which is half the communication complexity of γ. The time complexity of synchronizer θ is $2 \log_k |V| + 2$ per pulse, which is half the time complexity of γ. The communication complexities of γ and θ are identical.

2 Covers

In a network $N = (V, E)$, a *cluster* is defined as a set of nodes $S \subseteq V$, such that the graph induced by S (the nodes of S and the edges that connect them) is connected. A *cover* [6] of a network is a collection of clusters $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ such that $\bigcup_{i \in 1 \ldots m} S_i = |V|$. An example for such a cover is the set of $|V|$ clusters $\{S_1, S_2, \ldots, S_{|V|}\}$ where the cluster S_v contains the node v and all its neighbors.

Let S be a cluster and $v, w \in S$ two nodes in the cluster, $dist_S(v, w)$ is defined as the length of the shortest path in S between v and w. The diameter of a cluster S, $Diam(S)$, is defined as $\max_{v,w \in S}(dist_S(v, w))$. For a cluster S and a node $v \in S$, $Rad(v, S)$ is defined as $\max_{w \in S}(dist_S(v, w))$. The radius $Rad(S)$ of a cluster is defined as $\min_{v \in S}(Rad(v, S))$. The radius of a cover $Rad(\mathcal{S})$ where $\mathcal{S} = \{S_1, S_2, \ldots S_m\}$ is defined as $\max_{i \in 1 \ldots m}(Rad(S_i))$, the diameter of a cover $Diam(\mathcal{S})$ is defined as $\max_{i \in 1 \ldots m}(Diam(S_i))$. The *volume* of a cover $\mathcal{S} = \{S_1, S_2, \ldots S_m\}$ is defined as $vol(\mathcal{S}) = \Sigma_{i=1 \ldots m}|S_i|$. A cover $\mathcal{T}$ is said to be a *coarsening* of a cover $\mathcal{S}$, if for each cluster $S_i \in \mathcal{S}$, there exists a cluster $T_j \in \mathcal{T}$ such that $S_i \subseteq T_j$.

In [14], the author presents an algorithm named GV (Global Volume) that, given a cover S of a network graph and an integer parameter $z \geq 1$, constructs a coarsening cover $\mathcal{T}$ that satisfies the following properties:

1. $vol(\mathcal{T}) \leq |V|^{1+1/z}$.
2. $Rad(\mathcal{T}) \leq z \times Diam(\mathcal{S}) + Rad(\mathcal{S}) \leq (2z + 1) \times Rad(\mathcal{S})$.

In the sequel, we call the input cover clusters $\mathcal{S}$-clusters, and the output cover clusters $\mathcal{T}$-clusters. The GV algorithm operates iteratively. At each iteration, one $\mathcal{S}$-cluster S_0 is selected, and a new cluster $T \in \mathcal{T}$ is formed by merging S_0 with other $\mathcal{S}$-clusters. The merging procedure itself is performed in iterations, where in each iteration a new layer of $\mathcal{S}$-clusters is selected and merged into T. A formal description of the algorithm is given in Table 1.

```
$\mathcal{T} \leftarrow \emptyset$
while $\mathcal{S} \neq \emptyset$ do
    1. Select an arbitrary cluster $S_0 \in \mathcal{S}$
    2. $\mathcal{S} \leftarrow \mathcal{S} - \{S_0\}$; $K \leftarrow S_0$
    3. $\mathcal{Q} \leftarrow \{S | S \in \mathcal{S},\ S \cap K \neq \emptyset\}$
    4. $T \leftarrow K \cup \bigcup \mathcal{Q}$; $\mathcal{S} \leftarrow \mathcal{S} - \mathcal{Q}$
    5. if $T > |V|^{1/z}|K|$ then $K \leftarrow T$; goto 3
    6. $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
```

Table 1. Algorithm GV

The proof of correctness for GV is given in [14]. Here we provide a sketch of this proof (for the proof of the Lemmas that appear in this section, see [14]):

Lemma 1. *The resulting $\mathcal{T}$ is a cover of the network.*

For each cluster $T \in \mathcal{T}$, let $K(T)$ denote the value of the set K when this cluster is completely formed and joins the cover (line 6 in the algorithm).

Lemma 2. *For every $T \in \mathcal{T}$, $|T| \leq |V|^{1/z}|K(T)|$.*
Lemma 3. *For every $T, T' \in \mathcal{T}$, $K(T) \cap K(T') = \emptyset$.*
Corollary 4. *$vol(\mathcal{T}) \leq |V|^{1+1/z}$.*

Now, consider some iteration of the main loop, starting with the selection of a cluster $S_0 \in \mathcal{S}$ and ending with a new cluster T added to $\mathcal{T}$. Suppose that the internal loop was executed for J iterations. Denote the initial set K by K_0 $(= S_0)$. Denote the sets T and K constructed at steps 4 and 5 of the i-th internal iteration, $i \geq 1$, by T_i and K_i respectively.

Lemma 5. *For every $0 \leq i \leq J-1$, $|K_i| \geq |V|^{i/z}$, and strict inequality holds for $i \geq 1$.*
Corollary 6. *$J \leq z$.*
Lemma 7. *$Rad(\mathcal{T}) \leq Rad(\mathcal{S}) + z \times Diam(\mathcal{S}) \leq (2z+1) \times Rad(\mathcal{S})$.*

3 Synchronizer η_1

The first synchronizer we present, η_1, possesses exactly the same time and communication complexities as synchronizer γ. The initialization phase of synchronizer η_1 uses the GV algorithm for creating a cover of the network, to be used later by the synchronizer. The input cover given to GV is the cover $\mathcal{S} = \{S_1, S_2, \ldots, S_{|V|}\}$ such that S_v is a cluster that contains node v and all its neighbors. The parameter given to the GV algorithm is $z = \log_k |V|$, where k is a parameter given at initialization time $(2 \leq k \leq |V|)$. Therefore, the cover $\mathcal{T}$, created by GV satisfies the following:

1. $Rad(\mathcal{T}) \leq 2\log_k |V| + 1$, $vol(\mathcal{T}) \leq k|V|$.
2. $\mathcal{T}$ is a coarsening of $\mathcal{S}$.

All three predicates follow trivially from the properties of algorithm GV.

Observe that for every node v, there exists a cluster $T \in \mathcal{T}$ such that v and all its neighbors are in T. This derives from the fact that $\mathcal{T}$ is a coarsening of $\mathcal{S}$, which means that there exists a cluster $T \in \mathcal{T}$ such that $S_v \subseteq T$.

After GV is performed, each node $v \in V$ chooses the cluster that contains S_v (that was created by GV by combining S_v and other $\mathcal{S}$-clusters) to be its *home cluster*. In the sequel, we call the group of nodes that have selected a cluster T to be their home cluster as the *tenants* of T. In addition, a spanning tree of height $\leq 2\log_k |V| + 1$ is constructed for each cluster $T \in \mathcal{T}$. Such a tree exists for every cluster in $\mathcal{T}$ since $Rad(\mathcal{T}) \leq 2\log_k |V| + 1$. In the sequel, the root of the created spanning tree is referred to as the *leader* of the cluster.

Synchronizer η_1 uses two types of messages: SAFE and AWAKE. Each message includes a parameter containing a name of a cluster leader. Synchronizer

η_1 works as follows: after each pulse, SAFE messages are converged along each cluster spanning-tree from the leaves to the root. Each leaf node sends a SAFE message to its parent node as soon as it is *safe*. Each intermediate node sends a SAFE message to its parent as soon as it is *safe* and has received a SAFE message from each of its children in the cluster spanning-tree. When the cluster leader receives a SAFE message from each of its children, it initiates a broadcast of AWAKE messages: each node in the cluster sends an AWAKE message to each of its children upon receiving an AWAKE message from its parent.

Each node in the network may be a member in more than one cluster spanning-tree. Therefore, SAFE and AWAKE messages sent along each spanning-tree contain the name of the cluster leader. The above protocol is performed independently at each cluster spanning-tree. A node that receives an AWAKE message that includes the name of its home cluster leader, performs the next pulse. The correctness of synchronizer η_1 follows from the following arguments:

- All nodes in the network get to perform each pulse exactly once. This is due to the fact that each node has selected exactly one home cluster.
- Each node performs each pulse only after receiving all messages sent to it at the former pulse: each node performs each pulse upon receiving an AWAKE message from its parent in its *home cluster*, that is, only after all the nodes in its home cluster are *safe*. The nodes choose their home cluster such that all neighbors of each node are members in its home cluster. Thus, a node performs a pulse only after all its neighbors are *safe* with respect to the former pulse.

The time complexity of η_1 is as follows: the convergecast of SAFE messages along each cluster spanning-tree takes $2\log_k |V| + 1$ periods of time. The broadcast of AWAKE messages along the cluster spanning-trees takes $2\log_k |V| + 1$ periods of time. This adds up to $4\log_k |V| + 2$ per pulse. The communication complexity of η_1 is at most $2k|V|$: the number of edges in all cluster spanning-trees is bounded by the volume of the cover, which is $k|V|$ at the most. One SAFE and one AWAKE message is sent along each of these edges.

4 Synchronizer η_2

The time and communication complexities of synchronizer η_1 are almost exactly identical to those of synchronizer γ. In this section we present a synchronizer, named η_2, that requires communication complexity of $(k+2)|V|$, which is half the communication complexity of γ and η_1. The time complexity of η_2 is identical to the time complexity of γ.

Synchronizer η_2 is based on the observation that there is no point in broadcasting the AWAKE messages to all nodes of each cluster, since nodes that are not *tenants* of a cluster (the cluster is not their home cluster) do not perform the pulse upon receiving the AWAKE message from their parent in this cluster.

The initialization phase of synchronizer η_2 creates two trees for each cluster: (1) the cluster spanning-tree, over which the SAFE messages are converged. (2) the cluster tenants-tree, over which the AWAKE messages are broadcast.

In the sequel we show that the number of edges in all cluster tenants-trees is bounded by $2|V|$, and that the height of each cluster tenants-tree is bounded by $2\log_k|V|$. Therefore, the communication complexity of η_2 is $k|V|+2|V|$ messages per pulse, and the time complexity of η_2 is $4\log_k|V|+1$.

Let us describe a version of GV that builds a cluster tenants-tree for each created $\mathcal{T}$-cluster. The algorithm works only for the input cover $\mathcal{S}=\{S_1,S_2,\ldots,S_{|V|}\}$ where S_v is a cluster that contains the node v and all its neighbors. The code of the new version of GV is presented in Table 2.

```
$\mathcal{T} \leftarrow \emptyset$
$\boldsymbol{while}$  $\mathcal{S} \neq \emptyset$ $\boldsymbol{do}$
            { Constructing a new $\mathcal{T}$-cluster named $T$ }
    1.   Select an arbitrary cluster $S_v \in \mathcal{S}$
    2.   $\mathcal{S} \leftarrow \mathcal{S} - \{S_v\}$; $K \leftarrow S_v$; $T \leftarrow S_v$
    2a.  $Tenants(T) \leftarrow \{v\}$; $leader(T) \leftarrow v$
    3.   $\mathcal{Q} \leftarrow \{S | S \in \mathcal{S},\ S \cap K \neq \emptyset\}$
    3a.  For each $S_w \in \mathcal{Q}$ $\boldsymbol{do}$
             $\boldsymbol{if}$  $S_w \cap Tenants(T) \neq \emptyset$ $\boldsymbol{then}$
                 $parent_T(w) \leftarrow$ a node picked from $S_w \cap Tenants(T)$
             $\boldsymbol{else}$
                 $l \leftarrow$ a node picked from $S_w \cap T$
                 $parent_T(w) \leftarrow l$
                 $parent_T(l) \leftarrow$ a node picked from $S_l \cap Tenants(T)$
    3b.  $Tenants(T) \leftarrow Tenants(T) + \{w | S_w \in \mathcal{Q}\}$
    4.   $T \leftarrow K \cup \bigcup \mathcal{Q}$; $\mathcal{S} \leftarrow \mathcal{S} - \mathcal{Q}$
    5.   $\boldsymbol{if}$  $T > |V|^{1/z}|K|$ $\boldsymbol{then}$  $K \leftarrow T$; goto 3
    6.   $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
```

Table 2. Algorithm GV with cluster tenants-trees construction

The correctness of step 3a which actually constructs the cluster tenants-trees follows from the fact that at all times, each node in T is either in $Tenants(T)$ or is a neighbor of a node that is in $Tenants(T)$. From step 3b and 4, each node is a tenant of exactly one cluster. From step 3a, each node contributes 2 edges to its cluster tenants-tree at the most. Therefore, the number of edges in all cluster-tenants is $2|V|$ at the most.

Each execution of step 3a increases the height of the cluster tenants-tree by 2 at the most, since it connects each of the new tenant nodes to an old one via 2 hops at the most. The height of the initial tenants tree created in step 2a is 0, and step 3a is performed $\log_k|V|$ times at the most per cluster, therefore the height of each cluster tenants-tree is $2\log_k|V|$ at the most. The cluster spanning-tree, of height $2\log_k|V|+1$ at the most, is created for each cluster T by performing a BFS algorithm, initiated by $leader(T)$.

5 Synchronizer θ

The synchronizer presented in this section, θ, requires communications complexity of $2k|V|$ per pulse, the same as synchronizer γ. The time complexity of θ is $2\log_k |V| + 2$ per pulse, which is *half the time complexity of* γ.

Like with synchronizers η_1 and η_2, the initialization phase of synchronizer θ uses the GV algorithm in order to create a cover of the network needed for the operation of the synchronizer. However, in this case the input cover given to the GV algorithm is different. In θ, the input cover given to the GV algorithm is built of $|E|$ clusters, where each two neighbors in the network form a cluster. In other words, the input cover $\mathcal{S}$ is $\{S_{v,w} | (v,w) \in E\}$ where $S_{v,w}$ is a cluster that contains nodes v, w and the edge that connects them.

Observe that $Diam(\mathcal{S}) = Rad(\mathcal{S}) = 1$. Therefore, the cover $\mathcal{T}$, created by applying the GV algorithm on $\mathcal{S}$, satisfies the following properties:

1. $Rad(\mathcal{T}) \leq \log_k |V| + 1$.
2. $vol(\mathcal{T}) \leq k|V|$.

Synchronizer θ works similarly to η_1: it uses two types of messages, SAFE and AWAKE. After each pulse, SAFE messages are converged along each cluster spanning-tree in the same way as done in η_1. In each cluster, when the leader receives SAFE messages from each of its children in the cluster spanning-tree, it initiates a broadcast of AWAKE messages along the cluster spanning tree.

The difference between the protocols of η_1 and θ is in the timing of the pulse at each node. In θ, each node performs the pulse only after receiving an AWAKE message from its parent on each of the cluster spanning-trees it belongs to. For example, a node that belongs to three clusters, waits to receive three AWAKE messages before performing the next pulse.

As in η_1 and η_2, the AWAKE and SAFE messages sent along the cluster spanning tree edges when performing θ contain the name of the cluster leader. In this way, a node that belongs to more than one cluster can distinguish between messages sent over different cluster spanning trees.

The correctness of synchronizer θ follows from the following two arguments:

- All nodes in the network get to perform each pulse exactly once. This is proved by induction on the pulse number n. The base is trivial: the cluster leaders initiate a broadcast of AWAKE messages along the cluster spanning-trees. Eventually, each node receives an AWAKE message from its parent on each of the cluster spanning-trees it belongs to and performs pulse(0).
  The induction step is proved as follows: assume that each node gets to perform pulse(n) exactly once. Since the convergecast of SAFE messages is performed independently over each cluster spanning-tree, each cluster leader eventually receives a SAFE message from each of its children and initiates a broadcast of AWAKE messages on its cluster spanning-tree. Thus, each node eventually receives AWAKE from its parent on each of the cluster spanning-trees it belongs to. When this happens, the node performs pulse($n+1$).

– Each node performs each pulse only after receiving all messages sent to it at the former pulse. This argument is proved as follows: assume, in contradiction, that when a node v performs pulse$(n+1)$, an original-protocol message sent to v from a neighbor w at pulse(n) is still on its way. Since the cover $\mathcal{T}$ is a coarsening of $\mathcal{S}$, there exists a cluster $T \in \mathcal{T}$ such that $S_{v,w} \subseteq T$. Synchronizer θ ensures that node v performs pulse$(n+1)$ only after receiving an AWAKE message from its parent in the spanning tree of T. Therefore, when v performs pulse$(n+1)$, all nodes in T are already *safe*, including w. This is in contradiction to the assumption that there is still an original-protocol message sent at pulse(n) on its way from w to v.

The time complexity of θ is as follows: the convergecast of SAFE messages along each cluster spanning-tree takes $\log_k |V|+1$ periods of time. The broadcast of AWAKE messages along the cluster spanning-trees takes $\log_k |V|+1$ periods of time. This adds up to a time complexity of $2\log_k |V|+2$ per pulse. The communication complexity of θ is at most $2k|V|$: the number of edges in all cluster spanning-trees is bounded by the volume of the cover, which is $k|V|$ at the most. One SAFE and one AWAKE message is sent along each of these edges.

6 A Distributed Version of Algorithm GV

In this section we describe a distributed version of the GV algorithm, that is used in order to initialize the η_1, η_2 and θ synchronizers. For clarity reasons, we have tried to keep this implementation as close as possible to the partition algorithm used in order to initiate synchronizer γ. The time complexity of the distributed GV algorithm presented in this section is $O(|V|\log|V|)$, the same as the time complexity of the partition algorithm used in order to initialize synchronizer γ. The communication complexity of this distributed GV algorithm is $O(|V|^2)$. This is k times better than the communication complexity of the partition algorithm of γ, which is $O(k|V|^2)$. The reason for the reduced communication complexity is that there is no need to elect preferred links.

The initialization required by the distributed GV algorithm is as follows: a leader is selected for each $\mathcal{S}$-cluster. In addition, a spanning tree is created for each $\mathcal{S}$-cluster, by using a BFS algorithm initiated by its leader. We call these leader nodes and spanning trees '$\mathcal{S}$-leader nodes' and '$\mathcal{S}$-spanning trees', as opposed to the $\mathcal{T}$-leader nodes and the $\mathcal{T}$-spanning trees created by GV.

When initializing η_1 and η_2, $\mathcal{S} = \{S_1, S_2, \ldots, S_{|V|}\}$, where S_v is a cluster that contains v and all its neighbors. In this case, the $\mathcal{S}$-leader of S_v is v, and the $\mathcal{S}$-spanning tree of S_v is rooted by v and contains the edges $\{(v,w)|w$ is a neighbor of $v\}$. When initializing θ, $\mathcal{S} = \{s_{v,w}|(v,w) \in E\}$. In this case, the $\mathcal{S}$-leader of a cluster $S_{v,w}$ is selected arbitrarily from $\{v,w\}$. The $\mathcal{S}$-spanning tree is built of the edge (v,w).

The algorithm starts with an execution of a leader election algorithm [11], [2]. The election of an $\mathcal{S}$-leader and the creation of an $\mathcal{S}$-spanning tree for each $\mathcal{S}$-cluster can be performed by this algorithm without any penalty in communication or time complexities. This stage is also used in order to create at each node v a list $QList_v$ that contains the names of the $\mathcal{S}$-leaders of all $\mathcal{S}$-clusters v belongs to. Again, this is performed without penalty in communication or time.

In the distributed GV algorithm, $\mathcal{T}$-clusters are built one by one. Each time, a new $\mathcal{T}$-cluster is built from remaining $\mathcal{S}$-clusters ($\mathcal{S}$-clusters which were not merged into previously created clusters). This is done by selecting a $\mathcal{T}$-leader node and then creating the $\mathcal{T}$-cluster from $\mathcal{S}$-clusters in its neighborhood. The operation of removing an $\mathcal{S}$-cluster from the remaining $\mathcal{S}$-clusters list (after it had been used when building a $\mathcal{T}$-cluster) is performed by removing the name of the $\mathcal{S}$-cluster leader from $QList_v$ for each node v in this $\mathcal{S}$-cluster.

The algorithm makes sure that the selected $\mathcal{T}$-leader node is always a node that is a leader of some $\mathcal{S}$-cluster. The job of creating a $\mathcal{T}$-cluster T around a given $\mathcal{T}$-leader node v is performed by a procedure named Cluster_Creation, which operates in the following way: first, an $\mathcal{S}$-cluster that is leaded by node v is selected. At the first iteration, T contains this $\mathcal{S}$-cluster, and the spanning tree of T is selected to be the spanning tree of this $\mathcal{S}$-cluster.

Each of the following iterations of the Cluster_Creation procedure is performed as follows: the leader of T initiates a broadcast of PULSE messages along the spanning tree of T. Each node v that receives the PULSE message, checks $QList_v$. If $QList_v$ is not empty, all $\mathcal{S}$-clusters whose leader names appear in $QList_v$ are merged into T. This is done by sending LAYER messages along the $\mathcal{S}$-spanning trees of the $\mathcal{S}$-clusters. The LAYER messages contain the name of the $\mathcal{S}$-cluster along which they are sent.

A node w that is not in T and receives a LAYER(S) message, joins T. Assuming this LAYER(S) message is sent from a neighbor u, w marks u as its parent in the spanning tree of T. Node w also removes S from $QList_w$, sends LAYER(S) messages further along the spanning tree of S, waits for an $\mathrm{ACK}_0(S)$ or an $\mathrm{ACK}_1(S)$ message as an acknowledgement for each of the LAYER(S) messages it has sent, and then sends back an $\mathrm{ACK}_1(S)$ message, meaning that u is the parent of w in the spanning tree of T.

A node w that is in T and receives a LAYER(S) message, checks whether the name of the leader of S is in $QList_w$. If not, the LAYER(S) message is immediately acknowledged with an $\mathrm{ACK}_0(S)$ message. If S appears in $QList_w$, S is removed from $QList_w$, LAYER(S) messages are sent further along the spanning tree of S and w waits for $\mathrm{ACK}_0(S)$ or $\mathrm{ACK}_1(S)$ as an acknowledgement for each LAYER(S) message it has sent. Then w sends back $\mathrm{ACK}_0(S)$.

Each node that has sent LAYER messages upon receiving a PULSE message, waits to receive ACK for each LAYER message it has sent, and then sends a COUNT message to its parent in the spanning tree of T. The COUNT messages are converged along the spanning tree of T. The COUNT message sent by each node contains a parameter which is the number of the nodes in the subtree rooted by it, the same is true for ACK_1 messages as well. When the leader of T receives COUNT messages from each of its children, it decides whether to initiate another iteration or to initiate a search for another $\mathcal{T}$-cluster leader.

This concludes the description of the Cluster_Creation procedure process. We now describe the way in which the GV algorithm uses this procedure in order to create the cover $\mathcal{T}$.

Recall that the distributed GV algorithm begins with an execution of a leader election algorithm. The leader election algorithm [11], [2] ends at an elected 'core' edge. At least one of the nodes at the two ends of this edge must be a leader of an $\mathcal{S}$-cluster. We will call this node v_{init}. Node v_{init} initiates an execution of the Cluster_Creation procedure. The procedure execution ends at node v_{init}, after creating a $\mathcal{T}$-cluster led by it. At this point, node v_{init} calls the Search_For_Leader procedure, which searches for a node that will be a leader of a new $\mathcal{T}$-cluster. This node calls Cluster_Creation which creates a cluster around it, then it calls Search_For_Leader, and so on.

The Search_For_Leader procedure works as follows: it is initiated by the leader of a $\mathcal{T}$-cluster T that has just been formed. This leader node initiates a broadcast of TEST messages along the cluster spanning-tree. Then, a convergecast process of CANDIDATE messages is performed, where each node sends to its parent a message telling whether one of the nodes in its sub-tree is a member of an $\mathcal{S}$-cluster that has not yet been joined to a $\mathcal{T}$-cluster. If such a node is found, the new $\mathcal{T}$-cluster leader will be the leader of the $\mathcal{S}$-cluster found. If not, the center of activity backtracks to the cluster from which the leader of T was elected, and the above procedure is repeated there. In the sequel, we denote the cluster from which the leader of a $\mathcal{T}$-cluster T was elected as the *parent cluster* of T. Notice that the parent-child relation between the clusters creates a DFS tree.

The time complexity of the leader election algorithm in [11] is $O(|V|\log|V|)$. The time complexity of the leader election algorithm in [2] $O(|V|)$. The communication complexity is $O(|V|\log|V|+|E|)$ in both cases.

The Cluster_Creation procedure creates clusters of height $O(\log_k|V|)$ at the most. Therefore, at most $O(\log_k|V|)$ iterations are needed in order to create each cluster. In each $\mathcal{T}$-cluster T, PULSE and COUNT messages are sent only by nodes in $K(T)$ (See Sec. 2). Recall that $\Sigma_{T\in\mathcal{T}}K(T)\le|V|$. This means that at most $|V|\log_k|V|$ PULSE and COUNT messages are sent during the whole algorithm execution. The number of LAYER and ACK messages sent along each edge in each direction equals to the number of $\mathcal{S}$-cluster spanning-trees to which this edge belongs. When initializing η_1 or η_2, each edge belongs to exactly two $\mathcal{S}$-cluster spanning trees. When initializing θ, each edge belongs to exactly one $\mathcal{S}$-cluster spanning tree. Therefore, $O(|E|)$ LAYER and ACK messages are sent during the whole algorithm execution. Thus, the communication complexity of the Cluster_Creation procedure is $O(|V|\log_k|V|+|E|)$.

Now, consider a $\mathcal{T}$-cluster with c nodes, the height of this cluster spanning tree is at most $O(\log_k c)$. Hence, the total amount of time spent in forming this cluster is $O(c\log_k c)$. Summing up for all $\mathcal{T}$-clusters gives $O(|V|\log_k|V|)$ — the overall time complexity of the invocations of Cluster_Creation.

The Search_For_Leader procedure traverses the already created clusters in a DFS order, trying to find a free node which would be the leader node of a new cluster. Each time a cluster is traversed in search for a new candidate is referred to as a *move* (see [1]). A *move* involves a broadcast of TEST messages and a convergecast of CANDIDATE messages. It also involves a string of

LEADER messages from the cluster leader to the elected new node, or a string of RETREAT messages to the leader of the cluster that has elected the current leader (its parent in the clusters DFS tree). Thus, for both kind of clusters, $C_{move} = O(|V|)$ and $T_{move} = O(\log_k |V|)$.

Observe that each node v may be a leader of at most one $\mathcal{T}$-cluster. This is due to the fact that the Cluster_Creation procedure initiated by a node v performs at least one iteration per cluster, ensuring that $QList_v$ is empty. Each cluster contributes two *moves* to the execution of the algorithm, one is when the leader of the cluster is elected, and one is when the cluster is traversed and no new leader is found. There are at most $O(|V|)$ clusters in the network and therefore the total communication complexity of all invocations of Search_For_Leader is $O(|V|^2)$, and the total time complexity is $O(|V| \log_k |V|)$.

The total communication complexity of the distributed GV algorithm is, thus, $O(|V|^2)$. The total time complexity of the distributed GV algorithm is $O(|V| \log_k |V|)$. The process of creating the cluster-tenants tree needed by synchronizer η_2 adds $2|V|$ to the communication and time complexities of the distributed GV. This is straightforward from the scheme in Table 2.

6.1 An Improved Distributed Version of the GV Algorithm

Since the Preferred_Link_Election procedure used in the partition algorithm of γ is not needed by the distributed GV algorithm, the communication complexity of the Search_For_Leader procedure, which is $O(|V|^2)$ turns out to be dominant. The reason for the large communication complexity of the Search_For_Leader procedure is that a cluster with $O(|V|)$ nodes may execute a *move* of the Search_For_Leader procedure $O(|V|)$ times.

This section presents a distributed GV algorithm with communication complexity $O(|V| \log |V| + |E|)$ and time complexity $O(|V| \log_k |V|)$. The improvement in communication complexity is achieved by reducing the communication complexity of each *move* of the Search_For_Leader procedure to $O(\log_k |V|)$.

The idea is to let the Cluster_Creation procedure maintain a data structure by which the leader of each $\mathcal{T}$-cluster can decide whether there are nodes in this $\mathcal{T}$-cluster that are members of remaining $\mathcal{S}$-clusters, and also tells which is the way from that leader to such a node (if exists). In the sequel, we name such nodes *'potential nodes'*. With such a data structure, it is obvious that every *move* of the Search_For_Leader procedure takes $O(\log_k |V|)$ messages.

In order to create this data structure, each node in the network selects an *inspector* cluster, which will be the cluster that is responsible to check whether this node is a potential node or not. The inspector cluster of a node v is the first $\mathcal{T}$-cluster to which node v joins during the execution of the distributed GV algorithm. The nodes that have selected a cluster T to be their inspector are named the *'subordinates'* of T.

Each node v maintains one flag $f_{v,T}$ for each cluster T it participates in. The flag $f_{v,T}$ says whether there is a potential node that is a subordinate of T in the sub-tree of the spanning tree of T, rooted by v. After the execution of the Cluster_Creation procedure that creates a $\mathcal{T}$-cluster T, an additional iteration of broadcast and convergecast of messages along the created cluster spanning tree is performed. During this iteration the values of $f_{v,T}$ are set for all nodes $v \in T$. Observe that this additional iteration does not change the order of time or communication complexity of the Cluster_Creation procedure.

The new Cluster_Creation procedure works as follows: like in the old Cluster_Creation procedure, when an $\mathcal{S}$-cluster S joins the created $\mathcal{T}$-cluster T, the nodes in S remove the name of the leader of S from their $QList$ variable. In the new Cluster_Creation procedure, when a node w sets $QList_w \leftarrow \emptyset$, it checks to see whether this effects the value of its $f_{w,T'}$ flag, where T' is its inspector cluster. If not, the Cluster_Creation procedure continues. If $f_{w,T'}$ has been changed, node w sends a FLAG_CHANGED(T') message to its parent u in the spanning tree of T' and waits for a FLAG_CHANGED_ACK(T') message from u. When node u receives the FLAG_CHANGED(T') message from w, it updates $f_{u,T'}(w)$, which is a variable that contains the estimation at u for the value of $f_{w,T'}$. Then node u checks whether the value of $f_{u,T'}$ should be changed. If not, u sends a FLAG_CHANGED_ACK(T') message to w. If $f_{u,T'}$ should be changed, node u changes it and sends a FLAG_CHANGED(T') message to its parent in the spanning tree of T' and so on.

When performing Search_For_Leader, each *move* starts at a leader node l of a $\mathcal{T}$-cluster T. Node l checks $f_{l,T}$ to see whether there are potential nodes that are subordinates of T. If this is the case, the $f_{v,T}(w)$ flags will lead from l to a potential node using $O(\log_k |V|)$ messages. If not, RETREAT messages are sent to the leader of the cluster that has selected l, this costs $O(\log_k |V|)$ messages at the most.

Each node v performs $QList_v \leftarrow \emptyset$ exactly once during the whole execution of the GV algorithm, causing a string of $O(\log_k |V|)$ FLAG_CHANGED messages at the most and $O(\log_k |V|)$ FLAG_CHANGED_ACK messages at the most. Therefore, we have added $O(|V| \log_k |V|)$ messages to the communication complexity of the Cluster_Creation procedure, and the same amount to its time complexity. Hence, the communication complexity of the improved distributed GV algorithm is $O(|V| \log |V| + |E|)$ and its time complexity is $O(|V| \log_k |V|)$.

6.2 Other Known Distributed Cover-Construction Algorithms

Two types of cover construction algorithms are discussed in the literature: (1) algorithms that construct covers while minimizing their global volume (for example, the GV algorithm). (2) algorithms that construct covers while minimizing the maximum node degree, which is the maximum over the nodes of the number of clusters in which a node participates.

Algorithms of the second type are discussed in [14], [6], [7], [12], [8], [3]. Given a parameter z and a cover $\mathcal{S}$, these algorithms construct a coarsening cover $\mathcal{T}$ that satisfies the following properties: the maximum node degree $deg(\mathcal{T})$ is $O(z|V|^{1/z})$, $vol(\mathcal{T}) = O(z|V|^{1+1/z})$, $Rad(\mathcal{T}) = O(z \times Rad(\mathcal{S}))$. Setting $z = \log_k |V|$, we gain: $vol(\mathcal{T}) = O(k|V| \log_k |V|)$, $Rad(\mathcal{T}) = O(\log_k |V| \times Rad(\mathcal{S}))$.

Observe that the volume of the created cover $\mathcal{T}$ is $O(\log_k |V|)$ times the volume of the cover constructed when using the GV algorithm, and that there is no tradeoff between the radius of the created clusters and the volume of the cover. Therefore, using one of these cover construction algorithms in order to initiate η_1, η_2 or θ, would cause the communication complexity of the synchronizer to be $O(\log_k |V|)$ times larger than that of γ, and would eliminate the tradeoff between the synchronizer time and communication complexities.

Thus, only cover construction algorithms of type (1) are useful for our purpose. A distributed synchronous algorithm of this type is presented in [7]. However, the algorithm presented in Sec. 6.1 is better suited for our purpose than the algorithm presented in [7] since it creates clusters of smaller radius and since it has lower communication complexity.

7 Coping with Apparent Shortcomings

The bit complexity of a distributed protocol is defined as the worst case total number of bits in all messages sent by the nodes in V during an execution of the protocol. The bit complexity per pulse of η_1, η_2 and θ is larger than the bit complexity per pulse of γ. This is due to the fact that the AWAKE and SAFE messages sent along each cluster spanning tree when executing η_1, η_2 and θ contain the identity of the cluster leader, that occupies $\log |V|$ bits. The messages sent by the synchronization protocol of γ are all $O(1)$ in length.

Usually when computing the communication complexity of a distributed protocol, messages that are $O(\log_k |V|)$ bits long are assumed to cost exactly the same as messages that are $O(1)$ bits long. This is reasonable since even for very large networks, $\log |V|$ bits are much less than the number of bits appended to each message as a data-link control header, CRC, etc.

However, it is easy to change the synchronization protocols of η_1, η_2 and θ to send only messages that are $O(1)$ bits long: observe that the distributed GV algorithm described in Sec. 6 ensures that the edges that form the spanning tree of a $\mathcal{T}$-cluster T are all edges of the spanning trees of the $\mathcal{S}$-clusters that form T. The GV algorithm also ensures that an $\mathcal{S}$-cluster that is used when building a $\mathcal{T}$-cluster T is not used when building any other $\mathcal{T}$-cluster. When initializing θ, each edge is a member of exactly one $\mathcal{S}$-cluster. Thus, each edge is a member of at most one $\mathcal{T}$-cluster spanning tree. Therefore, there is no need to send any kind of information in the AWAKE and SAFE messages: the $\mathcal{T}$-cluster to which each such message applies can be extracted at the receiving node from the edge from which the message was received.

When initializing η_1 or η_2, each edge belongs to at most two $\mathcal{S}$-cluster spanning trees, and therefore, to at most two $\mathcal{T}$-cluster spanning trees. Thus, one bit of information is enough for distinguishing between messages that apply to different $\mathcal{T}$-clusters.

The amount of memory needed by the synchronization protocols of η_1, η_2 and θ at each node depends on the number of links connected to the node. At a node with d links, this amount of memory is $O(d \log |V|)$, as $O(\log |V|)$ bits are needed to store the names of the (at most two) $\mathcal{T}$-cluster spanning trees each edge participates in. This is larger than the $O(d)$ amount of memory needed for the synchronization protocol of γ at a similar node. However, $O(\log |V|)$ bits per link is still considered as a reasonable amount of memory. This amount of memory is much less than the amount of memory needed anyway in order to ensure correct simulation of the synchronous model (see [13], [16], [17], [18], [19]).

8 Acknowledgments

The authors wish to thank Hagit Attiya for helpful discussions.

References

1. B. Awerbuch, *Complexity Of Network Synchronization*, Journal of the Association for Computing Machinery, Vol. 32, No. 4, October 1985, pp. 804-823.
2. B. Awerbuch, *Optimal Distributed Algorithms of Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems*, STOC 1987, pp. 230-240.
3. B. Awerbuch, B. Berger, L. Cowen and D. Peleg, *Fast Network Decomposition*, PODC 1992.
4. I. Althofer, G. Das, D. Dobkin and D. Joseph, *Generating sparse spanners for weighted graphs*, 2nd Scandinavian Workshop on Algorithm Theory 1990, 26-37.
5. B. Awerbuch and D. Peleg, *Network Synchronization with Polylogarithmic Overhead*, 31st Symposium on Foundations of Computer Science 1990, pp. 514-522.
6. B. Awerbuch and D. Peleg, *Sparse Partitions*, 31st FOCS, 1990.
7. B. Awerbuch and D. Peleg, *Efficient Distributed Construction of Sparse Covers*, CS90-17, The Weizmann Institute, July 1990.
8. B. Awerbuch, B. Patt-Shamir, D. Peleg and M. Saks, *Adapting to Asynchronous Dynamic Networks*, Proc. 24th ACM STOC, 1992, pages 557-570.
9. D. Peleg and A. Schaffer, *Graph Spanners*, J. of Graph Theory 13, 1989, 99-116.
10. C.T. Chou, I. Cidon, I.S. Gopal and S. Zaks, *Synchronizing Asynchronous Bounded Delay Networks*, IEEE Transactions on communications, Vol. 38, No. 2, February 1990, 144-147.
11. R. G. Gallager, P. A. Humblet and P. M. Spira, *A Distributed Algorithm for Minimum-Weight Spanning Trees*, ACM Transactions on Programming Languages and Systems 5, 1983, 66-77.
12. N. Linial and M. Saks, *Decomposing Graphs into Regions of Small Diameter*, ACM/SIAM symp. on Discrete Algorithms, 1991 pages 320-330.
13. K.B. Lakshmanan and K. Thulasiraman, *On The Use Of Synchronizers For Asynchronous Communication Networks*, 2nd WDAG, Amsterdam, July 1987.
14. D. Peleg, *Sparse Graph Partitions*, CS89-01, The Weizmann Institute, Feb. 1989.
15. D. Peleg and J.D. Ullman, *An Optimal Synchronizer for the Hypercube*, SIAM Journal on computing, Vol 18, No. 4, pp. 740-747, August 1989.
16. L. Shabtay and A. Segall, *Active and Passive Synchronizers*, TR-706, December 1991, Computer Science Department, Technion IIT, submitted to NETWORKS.
17. L. Shabtay and A. Segall, *Message Delaying Synchronizers*, 5th WDAG, Delphi, October 1991.
18. L. Shabtay and A. Segall, *A Synchronizer with Low Memory Overhead*, 14th International Conference on Distributed Computing Systems, Poznan pp. 250-257.
19. L. Shabtay and A. Segall, *On the Memory Overhead of Synchronizers*, LPCR Report #9313, May 1993, Computer Science Department, Technion IIT.