
For each node $v$, and for each neighbor $u$ of $v$, there exist a cluster $C$ s.t. $u$ and $v$ are members in $C$ ($v$ and all its neighbors are members in $C$ respectively).

The maximum cluster height is $\log_K |V|$ ($2 \log_K |V|$ respectively).

$\text{Vol}(T) \leq K|V|$.

The algorithms used in the preprocessing phases of $\gamma_1$ ($\gamma_2$) to construct these decompositions are the most efficient algorithms known for these tasks in terms of time, communication, space, and bit complexities.

Using each of our synchronizers, performing a Breadth First Search in an asynchronous network, with no pre-established synchronizer, yields total communication and time complexities of $O(K|V|D + |E| + |V| \log |V|)$ and $O(|V| + D \log_K |V|)$ respectively, instead of $O(K|V|^2)$ and $O(|V| \log |V|)$ by [Awe85] (Where $D$ is the diameter of the network) with constant memory per edge and constant messages size. In [Awe85], the construction phase dominates the communication and time complexities of the whole algorithm, while with our synchronizers, the construction is much simpler and efficient, and 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 $O(|V| \log |V|)$ respectively.

The synchronizers $\theta_1$ and $\theta$ of [SS94] have the same time and communication overhead as ours, but their messages’ size and memory requirement are not constant. The preprocessing phase presented there requires $O(|V| \log_K |V|)$ time, and logarithmic message size, and log $|V|$ memory per edge. More important, the general purpose coarsening algorithm their construction is based upon, does not deal correctly with certain situations, as was shown in section 5.2, and the algorithm proposed by us, corrects it.

References


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 \( \tilde{S} \) rooted at \( u \). Since \( \tilde{S} \) 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.

\[ \square \]

### 5.4 Construction’s Complexity

**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 \( \zeta \) are divided into 4 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 \( O(\text{Vol}(S)) \) such messages.

- **delete father** messages are sent at most once by each node for each source cluster it belongs to. Thus, the number of these messages is \( O(\text{Vol}(S)) \).

- **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. By the fact that cluster \( T \)’s cardinality is multiplied in each iteration by at least \( K \geq 2 \), 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 \( O(\text{Vol}(S)) \). 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 \( \zeta \) 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 \( \gamma_2 \) preprocessing phase in section 4.2.2, and by claim 5.9, we get that the time complexity of \( \zeta \) is \( O(|V| \cdot d_S) \).

### 6 Conclusions

Due to the elimination of the need to elect preferred edges, and to the efficient election of the clusters’ leaders, our synchronizers \( \gamma_1 \) and \( \gamma_2 \), along with their preprocessing phases, remain the simplest, all-purpose, memory-efficient synchronizers.

Synchronizers \( \gamma_1 \) (\( \gamma_2 \) respectively) use network decompositions into clusters which satisfy, for a given parameter \( K \):
is the graph induced by the set of directed edges \( E' = \{(u, v) \mid v \text{ is the established father of } u \text{ in } T \text{ at the end of iteration } i \} \). Recall that \( \tilde{T} \) is the digraph induced by the set of edges \( E(\tilde{T}) = \{(u, v) \mid v \text{ is the last node which } u \text{ established as its father in } T \text{ during iteration } i \} \). Then \( T[i] \) is obtained by adding to \( T[i - 1] \) the edges in \( \tilde{T} \) at the end of iteration \( i \). Hence it is enough to show that at the end of iteration \( i \), \( \tilde{T} \) is a directed forest whose roots are nodes in \( T[i - 1] \), and all its other nodes are in \( T[i] \setminus T[i - 1] \). The proof is similar to that of Claim 5.3, using the following observations:

1. Every node in \( T[i] \setminus T[i - 1] \) has exactly one established father in \( \tilde{T} \).
2. Every node in \( \tilde{T} \cap T[i - 1] \) has no established father in \( \tilde{T} \).
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. \((u, v) \in \tilde{T} \), \( d_u > d_v \), thus there are no circuits in \( \tilde{T} \).

\[ \square \]

**Lemma 5.6** At each iteration, the height of \( T \) is increased by at most \( d_S \).

**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 \( d_S \). 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 \( l_v \leq d_v \leq d_S \).

Let \( S \) be a source cluster by which \( v \) was explored. Let \( p = (u_0, u_1, \cdots, u_{k-1}, v) \) 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 \( k \leq d_S \)). For \( 0 \leq i < K \), 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 \( d_{u_i}(T) \), once set, can only decrease. Hence, when \( v \) is explored, \( d_v(T) \) is at most \( k \leq d_S \), and hence \( d_v \leq k \leq d_S \).

Next we prove by induction on \( l_v \) that for each node \( v \), \( l_v \leq d_v \). 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 father of \( v \) in \( T \). Then \( l_w = m - 1 \), and \( u \) is the last node which \( v \) establishes as a father in \( T \) during iteration \( i \). When \( v \) marked \( w \) as its father 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 \( d_{u_i}(T) + 1 \). Denote this value by \( d^* \). Since \( v \) did not establish a different father later, the final value of \( d_v(T) \), \( d_v \), is equal to \( d^* \). Since the value of \( d_{u_i}(T) \) could only decrease, we have that at the end of iteration \( i \), \( d_v = d^* \geq d_w + 1 \geq l_w + 1 = l_v \). \[ \square \]

**Corollary 5.7** For all \( T \in T \) the height of \( T \) is at most \( d_S \cdot \log_K |V| \).

**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 father in \( \tilde{S} \) contains exactly the number of nodes explored by \( S \) in the subtree of \( \tilde{S} \) 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 \( \tilde{S} \) 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 sons in \( \tilde{S} \) is 0. The sum of the \( c \) fields in the acknowledgment messages from its sons in \( \tilde{S} \), by the induction assumption, equals the number of nodes explored in \( T \) by \( S \) in the subtree of \( \tilde{S} \) 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
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 \((u, u_1, \cdots, u_{k-1}, v)\) 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 \( 1 \leq j \leq k-1 \), \( u_j \) 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 forwarded only on edges of \( S \), and the fact that these messages are initiated only by nodes in source clusters which intersect \( T[i-1] \), nodes which do not belong to such clusters will not receive expansion messages during iteration \( i \), and hence are not explored during this iteration. \( \square \)

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

**Proof:** Each node \( v \in \tilde{S} \) which is not in \( T[i-1] \) has exactly one father 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 father in \( S \), thus such a node must be a root. Finally, \( u \) is the father of \( v \) in \( \tilde{S} \) only if it is a root or it received an \( S-T \) expansion message before \( v \) did. Thus, \( \tilde{S} \) contains no cycles. The claim follows \( \square \)

We define a **string of messages** \((m_0, m_1, \cdots, m_k)\) as a sequence of messages, generated at nodes \((u_0, u_1, \cdots, u_k)\) respectively, where for \( 1 \leq i \leq k \), 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. \( \square \)

**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 \( \tilde{S} \), on which the \( S-T \) acknowledgments converge to the originators of these messages in the standard convergentcast 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 \( T[i] \)
Figure 3: An illustration of the cover-coarsening algorithm

sets the $f$ bit to 1, to establish $v$ as its father in $T$. When $v$ receives this $S_1$-$T$ acknowledgment message, it sets the $c$ and $f$ fields in it as described next, and forwards it to $w$. $v$ sets $c$ 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 father in $S_1$. $v$ sets the $f$ bit to 1 to establish $w$ as its father in $T$, since $w$ is marked as $v$'s father in $T$.

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 father 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 father in $T$. $v$ will also send a delete father message to $w$, informing $w$ that it is not its father in $T$ anymore, the tree edges that were added to $T$ during this process are all the edges in $S_1$ and $S_2$, except the edge $(w, u)$.

There are two important points that this example illustrates: first, that the change of father is essential in order to keep the incremental addition to the diameter of $T$ bounded by $d_S$. 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 father in the $S_1$-cluster). It appears that these two properties are not satisfied by the algorithm in [SS94]; 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 father 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 Construction’s Correctness

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 $d_S$, and the root of $T$
3. mark \( u \) as its father 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:** \( v \not\in T[i-1] \), 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 father in \( S \).
  2. increment \( d \) by one.
  3. If \( d < d(T) \) then it marks \( u \) as its father 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 father in \( S \), say \( u \), then it also sends an \( S\)-\( T \) acknowledgment message to \( u \). The count field \( c \) and the father 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 father 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 father in \( T \), it might be the case that \( v \) had previously established a different father in \( T \), say \( x \). In this case, \( v \) has to send \( x \) a delete father message announcing that it is no longer its father 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 father 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 sons in \( T[i-1] \) and acknowledgments 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

Before proving the properties of Algorithm \( \zeta \), we describe a specific scenario which illustrates some of its properties. This example is depicted in Figure 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 figure 3). Now let the \( i \)-th 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 father 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 father 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 father in \( S_1 \). \( x \) also
\[ \gamma_2 \] presented in Section 4, since it sends cluster identities in its messages. Also, \( \zeta \) does not imply synchronizers which have constant messages size and constant memory per edge, like \( \gamma_1 \) and \( \gamma_2 \).

### 5.1 Description of the Cover-Coarsening Algorithm \( \zeta \)

Algorithm \( \zeta \) 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 \( S \in S \) is eliminated from \( S \) and becomes layer 0 of \( T \). In iteration \( i > 0 \), each cluster \( S \in S \) which intersects \( T[i-1] \) is eliminated from \( S \) and merged into \( T[i] \) 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 \( d_S \). 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 \( T \) is less than \( (K-1)|V| \). Thus, \( Vol(T) \leq K|V| \). 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 \( d_S \cdot \log_K |V| \).

The construction of a target cluster \( T \) starts in iteration 0, when a cluster \( S \in S \) is eliminated from \( S \) 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 father-son relation in cluster \( S \) is ignored, and a new father-son relation is created, as follows. A node \( v \) in \( S \setminus T[i-1] \) which receives an \( S-T \) expansion message for the first time, marks the sender of this message as its new father in \( S \). In addition, each such node \( v \) also establishes a father in \( T \). Thus, the following structures are maintained during iteration \( i \):

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

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 \( \tilde{S} \) rooted at \( v \), and a bit \( f \) which is set to 1 if \( v \) establishes \( u \) as its father in \( T \). \( v \) may later replace its father 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 father in \( T \).
edges belong to \( C_E(v) \) or to \( C_T(v) \) (or both), which can be done by two bits per edge. Similarly, each tree edge entering \( v \) may belong to \( C_E(v) \), or to \( C_T(v) \) (or both), or to a tree spanning a different cluster (in this last case \( v \) must be a leaf of that tree). This information can be kept in 3 additional bits per edge. For each outgoing tree edge \((v \rightarrow u)\) which belongs to \( C = C_T(v) \), \( v \) keeps a record whether \( u \) is explored in \( C \) or a tenant of \( C \) (these bits are set during the construction of \( C \)).

We now show that constant message size suffices to implement \( \gamma_2 \). Using the above structure, a safe message sent by a node \( v \) to its father \( u \) 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 father in one of these two trees only when \( u \) and all its sons (if any) on that tree are safe. Similarly, since the information kept in each node \( u \) enables \( u \) to identify its sons in \( C_T(u) \) which are either explored by or tenants in \( C_T(u) \), \( u \) need no additional information in order to forward the go-ahead messages it receives from its father in \( C_T(u) \) only to these nodes, as required.

4.2.6 \( \gamma_2 \) 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 \( C_i \), all nodes except the root send one safe message, which gives total of \( \sum_C |C| \leq K|V| \) messages per pulse. In addition, each node receives one go-ahead 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 4.2.5 The memory requirement of \( \gamma_2 \) is \( O(1) \) per adjacent edge at a node, and control messages size is also \( O(1) \).

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**: An integer \( K \), and a source cover \( S = \{S_1, S_2, \ldots, S_m\} \) of a graph \( G = (V, E) \), where each cluster \( S_i \in S \) 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 \( d_S \).

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

We present here a cover-coarsening algorithm, named \( \zeta \). Assuming that the network has a leader, the communication and time complexities of \( \zeta \) are \( O(Vol(S)) \) and \( O(\min(Vol(S), d_S) \cdot V) \) respectively. Our algorithm improves a similar algorithm given in [SS94] in several aspects: It correct a problem in the cluster-construction procedure presented there, as we explain in details later, and it is simpler and more efficient in its complexity measures. Algorithms related to \( \zeta \) are also studied in [Pel89, AP92]: [Pel89] present a centralized cover-coarsening algorithm. In [AP92], algorithms for the related task of coarsening a cover comprises of the \( m \) neighborhoods of every \( v \in V \) for \( 1 \leq m \leq \log |V| \) are given. The techniques used there are different from ours, and the algorithms performances are incomparable with ours.

Note that \( \zeta \) can serve as a preprocessing algorithm for \( \gamma_1 \), by letting the input cover \( S \) be the set of all edges in \( E \) (thus \( Vol(S) = 2|E| \) and \( d_S = 1 \)), and for \( \gamma_2 \), by letting \( S \) include for each node \( v \) the cluster composed of \( v \) and its neighbors (thus \( Vol(S) = 4|E| \) and \( d_S = 2 \)). We note, however, that the bit complexity of \( \zeta \) is higher than that of the preprocessing algorithms for \( \gamma_1 \) and
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 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 $\gamma_2$ construction in a network with a leader are $O(|E|), O(|E| + |V| \log |V|)$ and $O(|V|)$ respectively.

4.2.3 $\gamma_2$ operation

The simulation of a pulse by $\gamma_2$ is done in two phases, as in $\gamma_1$, with the following changes: the goahead messages are forwarded from a node $u$ in a cluster $C$ to its son $v$ 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 sons.

4.2.4 $\gamma_2$ correctness

For $\gamma_2$ 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 sons 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 sons, it broadcasts a goahead message to its cluster, and this message is forwarded by every node in $C$ to all its sons 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 father is a tenant in $C$, each node explored in $C$ will receive a goahead message from its father in $C$.

4.2.5 An efficient implementation of $\gamma_2$

In this section we show that synchronizer $\gamma_2$ 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. Note that since a node $v$ may participate in more than one cluster, $v$ needs to distinguish between the different (trees spanning the) clusters it belongs to. Thus, a straightforward implementation of $\gamma_2$ may require $O(\log |V|)$ bits per tree edge in order to keep the identity of the cluster (or clusters) which use 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 need to identify which outgoing tree
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. \hfill \Box

before analyzing $\gamma_2$ 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 father 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 father is in layer $j$ of $C$, for $j \leq i - 1$. If $v$ was explored in the second sub-iteration of iteration $i$, then its father is in layer $i - 1$ of $C$. In both cases, $v$’s father is not in the last layer of $C$, and hence it is a tenant of $C$. \hfill \Box

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. \hfill \Box

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 $i$-th iteration of its construction, for $i \geq 1$. Since by Claim 4.6 $v$ and all its neighbors were explored before $C'$ was constructed, neither $v$ nor any of its neighbors join $C'$ in the first sub-iteration of iteration $i$. Finally, since none of $v$’s neighbors join $C'$ in the first sub-iteration of iteration $i$, $v$ does not join $C'$ also in the second sub-iteration of that iteration. \hfill \Box

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. \hfill \Box

4.2.2 Complexity measures of $\gamma_2$ preprocessing phase

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 $O(|E|)$, which gives total of $O(\sum C |C|)$ when summing up over all the clusters. Since a node $u$ can be a son 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\textsuperscript{4}. Hence, the volume $\sum C |C|$ is bounded by $O(|E|)$. Thus, the communication complexity of the preprocessing phase is $O(|E|)$.

\textsuperscript{4}Using Claim 4.6, one can improve this somewhat, and show that for each edge $(u, v)$ the father-son relationship can exist at most twice in one direction and once in the other.
a node is a tenant in at most one cluster. Thus we get that the sum of the sizes of the clusters satisfies: \( \sum_c |C| = \sum_c (t_c + l_c) < \sum_c K \cdot t_c = K \sum_c t_c \leq K|V| \). Hence, the total number of control messages per pulse is less than \( 2K|V| \).

The overhead in time complexity in synchronizer \( \gamma_1 \) 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 convergecasted from the leaves to the root, while the goahead messages are broadcasted in opposite direction.

The memory requirement of \( \gamma_1 \) is \( O(1) \) per adjacent edge at a node. By lemma 4.1 a node \( v \) can have sons 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 sons in \( C_v \), \( v \)'s father in \( C_v \), and \( v \)'s fathers in other clusters. This can be done by three bits per edge. Also, since a node \( v \) may be a father 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.

### 4.2 Variant 2: Synchronizer \( \gamma_2 \)

#### 4.2.1 \( \gamma_2 \) cluster creation

The preprocessing phase of \( \gamma_2 \) guarantees that for each node \( v \) there is a cluster \( C \) such that \( v \) and all its neighbors belong to \( C \). To achieve this, a node \( v \) marks itself as explored by cluster \( C \) after \( v \) and all its neighbors join cluster \( C \), and the algorithm terminates only after all nodes are explored; initially all nodes are unexplored.

Potential leaders are unexplored nodes. 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 \( i \geq 1 \), 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, \( v \) adds to layer \( i \) of \( C \) all its neighbors which are unexplored.

A node in layer \( i - 1 \) which completed the first sub-iteration of iteration \( i \) forwards this fact to the root. After learning that the first sub-iteration is completed, the root instructs the nodes that joined layer \( i \) in the first sub-iteration to start a second one. In this second sub-iteration, each node \( v \) which was added to \( C \) in the first sub-iteration and is unexplored yet, performs 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 \( \text{dist}_{G(C)}(r, v) \).

**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: 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 \( l \)-th 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 \( l \)-th iteration, then \( u, v \) are in the same cluster by the induction hypothesis. Otherwise, \( u \) is added to \( C \) in the \( l \)-th iteration, if it was not added to \( C \) earlier. In both cases, after \( v \) became a tenant, it shares a common cluster with \( v \).

We note that the preprocessing phase of \( \gamma_1 \) is very similar to the improved version of the preprocessing phase of synchronizer \( \gamma \), as presented in Section 3. The main difference is that in \( \gamma_1 \) the “last layers” are not rejected, but added to the cluster being constructed. Thus, the communication, bit and time complexities of the preprocessing phase of \( \gamma_1 \) are the same as those of the improved version of the preprocessing phase of \( \gamma \), described in Section 3.

### 4.1.1 \( \gamma_1 \) operation

The simulation of a pulse in \( \gamma_1 \) consists of two phases:

1. A node \( v \) which belongs to a cluster \( C \), sends a **safe** message to its father in the tree spanning \( C \) after it becomes safe and it receives **safe** messages from all its sons in this tree.

2. When the root is safe and it receives **safe** messages from all its sons, 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 fathers and **safe** from all its sons in \( C \). A non-root node generates the next pulse after receiving a **goahead** from all its fathers.

### 4.1.2 \( \gamma_1 \) 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 fathers in the clusters it belongs to. If \( v \) is a root of a cluster \( C \), it has also to receive **safe** from all its sons 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**.  

### 4.1.3 \( \gamma_1 \) 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 last layer in p. 6). Also, by Lemma 4.1,
4 Improved Variants of Synchronizer $\gamma$

We present here two new synchronizers, $\gamma_1$ and $\gamma_2$ that differ from synchronizer $\gamma$ in avoiding the need to use (and hence to elect) preferred edges. Instead, the inter-cluster communication is obtained by having neighboring clusters share common nodes (thus, the clusters used are not disjoint). The first synchronizer has communication and time overheads per pulse of $2K|V|$ and $2\log_K |V|$ respectively, while in the second synchronizer these overheads are $(K + 2)|V|$ and $4\log_K |V| + 1$. These synchronizers preserve the communication and time overheads of two similar variants which appear in [SS94], but outperform them in memory requirements and/or in the size of the messages used.

In the preprocessing phases of both $\gamma_1$ and $\gamma_2$, the clusters are constructed sequentially, along the lines described in Section 3. 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). During the construction of $C$, A tree spanning $C$ is also constructed, as described below. A node $v$ may be added to layer $i$ of $C$ by receiving a request to join $C$ from one or more nodes which are already in $C$; if $v$ joins $C$, then it sends a positive acknowledgment to one of these requests, and makes the sender of this request its father in the tree spanning $C$. At the end of iteration $i$, each node informs its father 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 is smaller than $K - 1$ times the current size of cluster $C$, then the construction of $C$ terminates, and layer $i$ is the last layer of $C$. Thus, every cluster contains a last layer, which might be empty. By requiring that in each iteration, except the last one, the cluster size is multiplied by at least $K$, we bound the number of iterations by $\log_K |V|$. At the end of each cluster construction, the root informs all nodes in the cluster about the construction termination. 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. 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 $C$.

4.1 Variant 1: Synchronizer $\gamma_1$

In this version, the preprocessing phase guarantees that for each edge $e = \{u, v\}$ in $G$, there is a cluster $C$ which includes both $u$ and $v$. In the preprocessing phase of this version, a potential leader is a node $v$ which is not a tenant yet. Initially, all nodes are not tenants. 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 $C$ first 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 is at least $K - 1$ times the number of nodes in layers $1, \ldots, i - 1$, 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.

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 sons in a cluster $C$ only if it is a tenant in $C$. $\Box$

Lemma 4.2 The construction algorithm terminates, and when it terminates, any two neighboring nodes $u$ and $v$ are members in a common cluster.
While there are potential leaders in the network do
1. Find a node $v$ which is a potential leader
2. Build a cluster whose leader is $v$

In previous works, in step 1 above a potential leader is elected from the currently terminated cluster or from the last layer that was rejected from that cluster. If no node can be found from that last cluster, the next cluster’s leader is elected from the most recently created cluster with a nonempty set of potential leaders. This task in [Awe85] requires communication and time complexities of $O(|V|^2)$ and $O(|V| \log |V|) \text{ respectively}$, while at [SS94] communication and time complexities are both $O(|V| \log |V|)$, but memory requirements could be $O(|V|)$ per node. Another method mentioned by [AP92] traverses the last created cluster in a Depth First Search (DFS) manner [Eve79], seeking for a potential leader, and when this cluster is exhausted, it backtracks to the cluster from which this one was created. This method has both communication and time complexities of the volume of the output cover (defined in page 1), since on each edge of each output cluster’s spanning tree, two messages are sent.

In this work we use the following simple method for finding the next potential leader: first, a minimal spanning tree (MST) 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 [Awe87, FM95]. All the nodes in this tree are marked as potential leaders. Now, the first cluster is constructed, starting from the root of the MST. 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 $\gamma$, 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 the MST, 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 MST 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 $\gamma$ in [Awe85] from $O(K|V|^2)$, $O(K|V|^2 \log |V|)$ and $O(|V| \log |V|)$ 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 $O(|V| \log |V| + |E|)$, $O((|V| \log |V| + |E|) \cdot \log |V|)$ 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 page 4), 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 page 4) 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 [Awe85]. (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 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; each such message is of size at

---

\[3\] The fact that there are only $O(|V|)$ such messages follows from an argument similar to the one given in Section 4.2.2.
least $K - 1$ times the magnitude of the already existing cluster, where $K \geq 2$ is a parameter. The depth (height) of each cluster is bounded by $\log_K |V|$, and the number of preferred edges used for inter cluster communication is bounded by $(K - 1)|V|$. The construction stage is composed of four tasks with the following complexities:

1. Electing a leader in the network. This requires communication and time complexity of $O(|V| \log |V| + |E|)$ and $O(|V|)$ respectively ([Awe87, FM95]). We note that the preprocessing phase in [Awe85] used for this task the earlier algorithm of [GHS83], whose communication and time complexities are $O(|V| \log |V|)$ and $O(|V| \log |V|)$ respectively.

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.

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

A simulation of pulse $p$ by synchronizer $\gamma$ is done as follows:

1. A node that becomes safe and receives safe messages from all its sons, sends a safe message to its father. 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 sons (and after 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 father and over all adjacent preferred edges, it initiates a convergecast of ready messages. A node sends a ready message to its father after it receives such a message from all its sons, and a cluster safe message over all its preferred edges. This phase has $\log_K |V|$ time and $|V|$ communication complexity.

4. Finally, after receiving ready messages from its sons and cluster safe messages over adjacent preferred edges, the root broadcasts go-ahead 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 $\gamma$ is worthwhile only when $2K|V| < |E|$, as otherwise the simpler synchronizer $\alpha$ is more efficient.

### 3 The Decomposition Method

A decomposition of a network to clusters, needed for synchronizers or for other reasons, is traditionally done in a sequential manner, constructing one cluster at a time. We will present few variants of this scheme. During the construction, some nodes in the network are potential leaders, meaning that they can start a new cluster creation. Thus, the decomposition of the network into clusters is performed according to the following scheme:

---

2 This does not affect the complexity measures of the algorithm in [Awe85], which are dominated by the task of electing preferred edges.
connecting neighboring clusters. This approach was first used in synchronizer $\gamma$ introduced in [Awe85], which is described below.

The messages sent by the synchronizer $\gamma$ and its variants can be divided into two sets: original messages, which correspond to messages sent by the simulated synchronous algorithm, 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. We also assume, as previous works do, that every original message is acknowledged by the receiving node. It is obvious that these acknowledgments do not increase neither time nor communication asymptotic complexity.

**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 $\gamma$ is a combination of two synchronizers – $\alpha$ and $\beta$. In synchronizer $\alpha$, 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 $\beta$, a tree spanning the network is constructed in a pre-processing phase. Each node, upon becoming safe at pulse $p$ and receiving safe messages from all its sons, sends a safe message to its father in the tree. When the leader has received safe messages from all its sons, it sends goahead messages to all its sons. A node, upon receiving the goahead message, passes it on to its sons 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 $O(|V|)$.

In synchronizer $\gamma$, 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

<table>
<thead>
<tr>
<th>Synchronizer</th>
<th>Communication Complexity</th>
<th>Time Complexity</th>
<th>Message Size</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$</td>
<td>$(2K + 2)</td>
<td>V</td>
<td>$</td>
<td>$4 \log K</td>
</tr>
<tr>
<td>$\theta$</td>
<td>$(2K + 2)</td>
<td>V</td>
<td>$</td>
<td>$2 \log K</td>
</tr>
<tr>
<td>$\eta_1$</td>
<td>$(K + 2)</td>
<td>V</td>
<td>$</td>
<td>$4 \log K</td>
</tr>
<tr>
<td>$\eta_2$</td>
<td>$(K + 2)</td>
<td>V</td>
<td>$</td>
<td>$4 \log K</td>
</tr>
</tbody>
</table>

Figure 1: Synchronizers Overhead per Pulse

<table>
<thead>
<tr>
<th>Synchronizer</th>
<th>Communication Complexity</th>
<th>Time Complexity</th>
<th>Bit Complexity</th>
<th>Space per Edge</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>$\eta_1, \eta_2$</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
<td>$O(</td>
<td>V</td>
</tr>
</tbody>
</table>

Figure 2: Preprocessing Complexities, Assuming the Network has a Leader
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 [Awe85]. In this approach, there is a pre-processing phase in which the network is partitioned to clusters, which are used by the synchronizer \( \gamma \) introduced there. [Awe85] also presents a distributed algorithm for this pre-processing phase; the bottleneck of this algorithm is the next cluster’s leader election with communication and time complexities of \( O(|V|^2) \) and \( O(|V| \log K |V|) \) respectively, and electing the preferred edges for inter-cluster communication, with communication and time complexities of \( O(K|V|^2) \) and \( O(|V| \log K |V|) \) respectively.

[SS94] presents three synchronizers, named \( \eta_1 \), \( \eta_2 \) and \( \theta \), which are similar to \( \gamma \); \( \eta_2 \) reduces the communication overhead by half, \( \theta \) reduces the time overhead by half, and all of them eliminate the need to elect preferred edges. The pre-processing phases required for these three synchronizers are performed in [SS94] by a general distributed cover-coarsening algorithm. As we show in Section 5, their algorithm fails to perform correctly in certain scenarios. In this paper we present a correct and more efficient version of that algorithm.

[AP90a] presents 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 should be known beforehand. 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 [Awe85, SS94] mentioned above.

[AP90b] presents a centralized cover-coarsening algorithm which achieves a lowest maximal and average node’s degree while maintaining the maximum cluster diameter relatively low. [AP92] presents a distributed algorithm for implementing the algorithms of [AP90b]. In an asynchronous setting, by means of the synchronizer described in [AP90a].

1.3 Our Contribution

In section 2 we review synchronizer \( \gamma \) of [Awe85] and its construction. In Section 3 we present a general method for network decomposition, and use it to construct an efficient pre-processing phase of a variant of synchronizer \( \gamma \). In Section 4 we use this method for presenting two new synchronizers, \( \gamma_1 \) and \( \gamma_2 \), which are variants of synchronizers \( \theta \) and \( \eta_2 \) of [SS94].

The performances of our synchronizers in overhead per pulse are compared with these of previous synchronizers in Figure 1, and the complexity measures of the pre-processing phases of these synchronizers in a network where a leader is already elected are compared in Figure 2.

In Section 5 we present a new efficient coarsening algorithm, which improves the coarsening algorithm in [SS94] in several ways.

2 The \( \gamma \) Synchronizer

As mentioned before, the synchronizers studied here are based on a pre-processing phase, in which the network is divided to clusters. A spanning tree is constructed for each cluster, and the synchronization messages are sent only on edges of these spanning trees, and possibly also on edges
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 algorithms on asynchronous networks, there is a need to synchronize the network. This is done by an algorithm named synchr onizer that simulates 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 radius 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 of clusters $\{C_i\}_{i=1..m}$ such that $\bigcup_{i=1..m} C_i = V$. A partition is a cover in which the clusters are mutually disjoint. Covers (partitions) are denoted by calligraphic letters (such as $S$, $T$, 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 $\text{dist}_G(u, v)$, is the length of a shortest path between $u$ and $v$ in $G$. For a node $v$ and a cluster $C$, $\text{dist}(v, C) = \min\{\text{dist}_G(u, v) | u \in C\}$. For two clusters $C$ and $C'$, $\text{dist}(C, C') = \min\{\text{dist}_G(u, v) | u \in C, v \in C'\}$. We say that two clusters $C$ and $C'$ are neighboring if $\text{dist}(C, C') \leq 1$. Two clusters $C$ and $C'$ intersect if they have at least one common node. A cover $T$ is said to coarsen a cover $S$, if every cluster in $T$ is a union of some clusters in $S$.

The volume of a cover $S = \{S_1, S_2, ..., S_m\}$, denoted as $\text{Vol}(S)$, is the sum $\sum_{i=1..m} |S_i|$. The diameter (radius) of a cluster $C$ is the diameter (radius) of the tree spanning $C$. The degree of a node $v$ in a cover $S$ is the number of clusters $S \in S$ which contain $v$.

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 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

\footnote{An alternative definition requires that every cluster in $S$ is included in some cluster in $T$.}
Simple and Efficient Network Decomposition and Synchronization

Shlomo Moran\textsuperscript{1} and Sagi Snir\textsuperscript{2}

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

February 27, 1997

Abstract

We present a simple and efficient method for the decompositions of a network into clusters. This method is used to perform the preprocessing phases needed for variants of the synchronizers in [Awe85, SS94] in $O(|V|)$ time and $O(|E| + |V|\log|V|)$ communication complexities, while maintaining constant messages size and constant memory per edge. The synchronizers resulted from our construction are more efficient than the original ones. For instance, they 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 $K \geq 2$ is a parameter, and $D$ is the diameter of the network.

We also present an efficient cover-coarsening algorithm, which improves the coarsening algorithm in [SS94] in several aspects.

\textsuperscript{1}E-address: moran@cs.technion.ac.il. The work of this author was supported in part by the Bernard Elkin Chair in Computer Science.
\textsuperscript{2}E-address: ssagi@cs.technion.ac.il