Laboratoire Bordelais de Recherche en Informatique
UMR 5800 - Université Bordeaux I, 351, cours de la Libération,
33405 Talence CEDEX, France

Research Report `RR-1315-04`

A Linear Time Distributed Algorithm for Graph Decomposition

Bilel Derbel and Mohamed Mosbah

February, 2004

A Linear Time Distributed Algorithm for Graph Decomposition

Bilel Derbel and Mohamed Mosbah
LaBRI - Université Bordeaux I - ENSEIRB
351, Cours de la libération, 33405 - Talence
{derbel,mosbah}@labri.fr

February, 2004

Abstract

We present a linear time distributed algorithm for decomposing a graph into a disjoint set of clusters. This algorithm is truly parallel since many clusters can be contructed in parallel, which gives an anwser to a question asked by S. Moran and S. Snir in [24]. Moreover, no precomputed spanning tree is required for the computation of clusters. We apply the designed algorithm to construct covers for synchronizers γ_1 and γ_2. An implementation and a few experimental results are presented showing the practical efficiency of the result.

1 Introduction

Due to the constant growth of networks, it becomes necessary to find new techniques to handle related global informations, to maintain and to update these informations in an efficient way. Locality-Preserving (LP) network representation [26] can be considered as an efficient data structure that captures some topological properties of the underlying network and help in designing distributed algorithms to efficiently solve many distributed problems : synchronization [24, 27, 12], Maximal Independant Set (MIS) [6], routing [10], mobile users [11], coloring [25] and other problems [19, 20, 21, 16, 1].
Clustered representations are one type of many existing LP-representations of a graph. The main idea of clustered representations is to decompose the nodes of a graph into many possibly overlapping regions called clusters. This representation allows to organize the graph in a particular way that satisfies some desired properties. In general, the clusters satisfy two types of qualitative criteria. The first criteria attempts to measure the *locality level* of the clusters. Some parameters like the *radius* or the *size* of a cluster are usually used to measure the locality level of the clustered representation of a graph. The second criteria attempts to measure the *sparsity level*. This criteria gives an idea about how the clusters are connected to each others. When a clustered representation is made of disjoint clusters, the number of intercluster edges can for example be used as a parameter to express this criteria. In case of overlapping clusters, the (average or maximum) number of occurrences of a node in the clusters is usually used to express the sparsity (or the overlap) of the clustered representation. There are several algorithms that construct many types of clustered representations of a graph. In this papers, we focus on three types of decompostions needed for synchronizers γ, γ_1 and γ_2. In particular, we give distributed implementations of these decompositions while mainting a good time complexity bound.

1.1 The Model

We represent a network of n processes by an unweighted undirected connected graph $G = (V, E)$ where V represents the set of processes ($|V| = n$) and E the set of links between them. Here are some useful definitions :

Definition 1.1 *The neighborhood of a node v in V is :*

$$N(v) = \{v\} \cup \{u \in V; (u, v) \in E\}$$

Definition 1.2 *A cluster S is a subset of vertices of V, such that $G(S)$ (the subgraph induced by S in the graph G) is connected.*

Definition 1.3 *A cover of a graph G is a set $\mathcal{S}$ of clusters that contain all the vertices of G.*

Definition 1.4 *A partition of G is a cover $\mathcal{S}$ containing only disjoint clusters.*

Definition 1.5 *For a cluster S in $\mathcal{S}$:*

$$\begin{aligned}
\Gamma(S) &= \bigcup_{v \in S} N(v) \\
Rad(S) &= \min_{v \in S}\{\max_{w \in S}\{dist_{G(S)}(v, w)\}\} \\
Rad(\mathcal{S}) &= \max_{S \in \mathcal{S}}\{Rad(S)\}
\end{aligned}$$

Definition 1.6 *The distance between two clusters S and S' is :*

$$dist(S, S') = \min_{u \in S, v \in S'}\{dist_G(u, v)\}$$

We assume that each node v of a graph G has a unique identity Id_v. A cluster produced by the decompositions that we will introduce is always considered with a leader node and a spanning tree rooted at the leader. A unique identity Id_S is assigned to each cluster S which is the identity of its leader. Each node v of the graph knows its own identity Id_v and the identity Id_S of the cluster S it belongs to. The algorithms we will describe are based on a parallel construction of the clusters. When the construction of one cluster is finished, this cluster is called *final* or *finished*. It is definitively part of the decomposition we are constructing.
We also assume that a node is an autonomous entity of computation that can only do local computations. A node can only communicate with its neighbors by sending and receiving messages. The system is assumed to be free of failures.

1.2 Known Results

There are many techniques to construct a clustered representation of a graph. [26] and [17] give a complete idea about these different techniques. Awerbuch and Al. first introduced such a representation in [6] called *network decomposition*. There is also a simple and efficient randomized algorithm for constructing a network decomposition given in [22] which has been applied in other works mainly in [17]. At the same time, another and more powerful type of a network representation was introduced in [9]. Many papers related to clustered representations (including graph decompositions, covers and partitions) exist in the literrature [5, 21, 4, 3, 25, 9, 6, 16] and many features (sparsity, locality, complexity etc.) of these

representations have been studied in order to improve some existing distributed algorithms or to find new applications [24, 11, 27, 10, 7, 8, 12, 2]. Note that a clustered representation of a graph is always modeled in order to satisfy some properties that suit a particular application.
In general, some parameters for measuring the sparsity and the locality levels are fixed such as the vertex or cluster degree. The properties of the representation are given in terms of these parameters. The locality and the sparsity of a decomposition are in general tightly related and often goes in an opposite way : improving one of them usually yields in making the other one worst. These two levels are also related to the complexity measures : low cluster radius (locality level) implies in general lower time complexity and low sparsity implies lower memory. All algorithms that one can find always attempt to find a good compromise between these two levels. Another reccurent and important point is that in order to use clustered representations to improve and to design other distributed algorithms, it is necessary to add a preprocessing phase which constructs the partition or the cover. That's why it is important to design algorithms that construct a good clustered representation while maintaining good complexity bounds.

1.3 Our Main Motivation and Contribution

An interesting application of clustered representations is designing synchronizers in order to simulate a distributed synchronous algorithm on asynchronous networks. In general, synchronizers use a mechanism to generate pulses and then simulate a global clock. There exist many types of synchronizers. Some synchronizers require to partition the network into clusters and so use a clustered representation of the network to simulate a synchronous network. This preprocessing phase should be done efficiently to avoid increasing the complexity of the synchronizers themselves. In [24], Moran and Snir gave a method to improve some existing algorithms for constructing a decomposition of the network into clusters needed for some synchronizers and also gave a new cover algorithm for constructing a possibly overlapping clusters based on a classical merging (or coarsening) technique. In general, except in [22], the usual technique to construct covers is to consider a source cover (for example the neighborhoods of graph nodes) and to use this cover, by merging together some clusters, to construct another cover (or partition) that satisfies some particular properties. The technique used for decomposing a graph usually constructs clusters in a semi-sequential manner. In fact, starting from a single source cluster, each final cluster is constructed distributively by adding at each iteration a new layer. At each phase there is only one final cluster that is constructed. In the case of synchronizers, a source cluster may be a single node, any pair of two neighboring nodes, or any neighborhood of a node. Moran and Snir end their paper [24] saying : are there truly parallel constructions of the sparse covers (needed for their synchronizers) which have a polylogarithmic or sub-linear time complexity ? In this paper, we give an algorithm that constructs a sparse partition of a graph. This partition has the same properties in terms of sparsity and locality, as the partition needed for synchronizer γ. Our algorithm does not require a spanning tree of the graph. Indeed, each vertex initializes a cluster which grows until reaching the desired property. Many clusters are therefore constructed at the same time. We also give two variants of our algorithm that construct two covers used in designing synchronizers γ_1 and γ_2. The time and message complexity of our algorithms are in the worst case resp. $O(|V|)$ and $O(\Delta|V|^2)$ where Δ is the maximum degree of G.

1.4 Structure of this paper

In Section 2, we give a formal overview of the technique used to construct a network cover used for synchronizers as in [24]. In Section 3, we introduce a new distributed algorithm for constructing a graph partition in a parallel way and discuss its complexity. In Section 4, we give two algorithms for constructing covers needed for synchronizers γ_1 and γ_2. In Section 5, we give some experimental results.

2 A Basic Algorithm For Constructing A Sparse Partition

Let k be an integer parameter. Algorithm $Basic\_Part$ [26] in Figure 1 constructs a decomposition of a graph G. It satisfies the following properties :

Theorem 2.1 ([26]) *The output $\mathcal{S}$ of algorithm $Basic\_Part$ is a partition of G and :*

1. $Rad(\mathcal{S}) \leqslant k-1$ *(locality level)*
2. *There are at most $n^{1+1/k}$ intercluster edges (sparsity level)*

Set $\mathcal{S} \leftarrow \varnothing$
while $V \neq \varnothing$ **do**
  Select an arbitrary vertex $v \in V$
  Set $S \leftarrow \{v\}$
  while $|\Gamma(S)| > n^{1/k}|S|$ **do**
    $S \leftarrow \Gamma(S)$
  end while
  Set $\mathcal{S} \leftarrow \mathcal{S} \cup S$ and $V \leftarrow V - S$
end while
return $\mathcal{S}$

Figure 1: Algorithm $Basic\_Part$

The partition produced by algotihm $Basic\_Part$ is used as a data structure for synchronizer γ. This application of network decomposition was first introduced in [2] and improvements were given in [27, 24]. Algorithm $Basic\_Part$ operates in many phases. At each phase, a leader vertex is selected from nodes which do not belong yet to a cluster. Then a cluster is constructed in many iterations.
There are many implementations of algorithm $Basic\_Part$. All of these implementations begin by electing a leader in the network which will be the center of the first cluster. Then the cluster is constructed in a layered fashion. At each iteration a new layer is added to the cluster. The construction of a cluster ends when there are no new layer to add (*i.e* : each node in $\Gamma(S)$ is already in some cluster) or if the sparsity condition $|\Gamma(S)| > n^{1/k}|S|$ is no longer satisfied. Once the construction of one cluster is finished, a new leader is elected from unproceeded nodes and a new cluster grows up around this leader using broadcast and convercast techniques. The main difficulity in these algorithms is to elect the next leader. In [24], a spanning tree T of the graph G is first constructed. This allows to elect the next leader by achieving a DFS traversal of T and by the same way improving considerably the complexity bounds of the decomposition : $O(|E|)$ messages and $O(|V|)$ time. However, the algorithm given in [24] still constructs clusters in a sequentiel way (one after the other). In [22], an efficient distributed algorithm for constructing a partition is introduced. The algorithm still operates in phases but at each phase only some clusters are constructed in parallel. The techniques used consist in allowing clusters to grow while avoiding collisions between them. However, the algorithm produces a partition that is different from the one produced by algorithm $Basic\_Part$. In the next section, we introduce a new algorihm that both produces the same partition as algorithm $Basic\_Part$ and constructs clusters in parallel.

3 A Deterministic Distributed Sparse Partition Algorithm

3.1 The Main Idea

In this section we give the general framework of algorithm $Dist\_Part$ which emulates in a distributed way algorithm $Basic\_Part$. The main idea of our algorithm is to allow clusters to grow in parallel and in a concurrent way. We do not avoid cluster collisions but manage the conflicts that can occur using cluster identities. Consider a single cluster S at some moment of the algorithm execution. This cluster will compete against other clusters in order to add a new layer. Either, the cluster enters in conflict with an adjacent one, say S_1 or with another cluster, say S_2 at distance two (see Figure 2). In the first case, cluster S tries to invade some nodes that belong to cluster S_1. In the second case, cluster S will compete for adding the nodes that separate it from cluster S_2. Symmetrically, clusters S_1 and S_2 will also compete against cluster S and against all clusters at distance one or two of them. In all cases, the clusters having the biggest identities always win against their neighboring clusters and succeed in adding new layers. The main feature of our algorithm which makes possible such a parallel construction is that at any moment at least the cluster having the biggest identity in all the graph always wins and so can not be stopped by another cluster. Once a cluster can not add a new layer because there are no layers to add or because it violates the sparsity condition (*i.e* : $|\Gamma(S)| < n^{1/k}|S|$), this cluster definitively stops adding new layers : it is a terminal (or final) cluster (it is definitively part of the decomposition). To sum up, clusters are always competing against each others in order to expand themselves. They may (respec. may not) succeed in adding new layers and they can invade (resp. be invaded by) neighboring clusters according to their identities. In our algorithm, a node v in the graph G can be in a *root*, *leaf*, *relay*, *orphan* or *final* state.

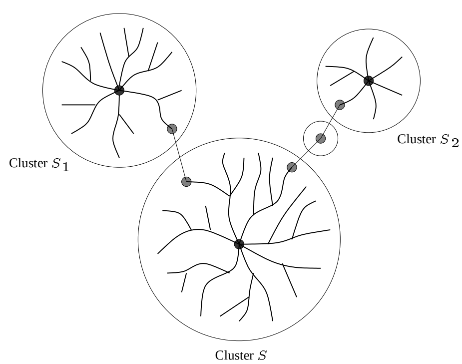


Figure 2: Two types of conflicts between clusters

Initially, all nodes are orphans. Each orphan node forms a cluster which we call *orphan cluster*. An orphan node is both the root and the leaf of its cluster. The identity of an orphan cluster is the identity of its single orphan node. Each orphan cluster will try to add a new layer. Thus, it must win against all the clusters at distance one or two. Consider an orphan cluster that succeeds in adding a new layer, this orphan node becomes the root node in the new formed cluster (radius 1) and the new nodes that join the new cluster become leaf nodes in this cluster. The root node marks the new nodes as its children and the new nodes mark the root node as their father. Now the new cluster will try to add a second layer (cluster of radius 2). The leaf nodes will try to win against all their neighbors using their cluster identity. Each leaf node informs its father weither it has locally won or lost. The cluster globally wins if all of its leaf nodes have locally won. Suppose that the cluster succeeds in adding a second layer. The leaf nodes mark their children and become relay nodes. On the other side, each newly added node mark its father (one new relay node) and becomes a leaf node in its new cluster. Each leaf node will again fight

locally against the neighboring clusters to allow their cluster to win new layers. The relay nodes enable to forward informations from the leaf nodes to the root node and conversely.
Note that as long as new layers are added a BFS tree spanning the cluster and rooted at its root node is constructed. The leaf nodes of this tree are the leaves of the cluster and the relay nodes are the nodes in the inside of the tree. The relay nodes are the links between the leaf nodes and the root node. The decisions of adding, removing or preserving a layer are made by the root node according to the informations forwarded from the leaf nodes. All communications between nodes inside a cluster are done using the constructed tree.

3.2 An example of cluster growth in algorithm $Dist\_Part$

In Figure 3, we have four clusters 1, 2, 3, 4 and 5 with identities $Id_1 > Id_2 > Id_3 > Id_4 > Id_5$. Cluster 5 is an orphan cluster because it has only one node. When a new exploration begins the leaves of cluster 1 will win the leaves of their neighboring clusters 5 and 3 and they both inform the root that the new exploration has succeeded. Cluster 2 wins against cluster 4 which is its neighbor but can not win cluster 5 because cluster 1 is stronger and it is a neighbor of cluster 5. Thus, cluster 2 can not add a new layer. Cluster 4 loses against both clusters 2 and 3 but it won't be invaded because both clusters 2 and 3 can not grow. Cluster 3 wins against cluster 4 but loses against cluster 1. Cluster 3 will be invaded by cluster 1 which wins against all clusters at distance two (cluster 5, 2 and 3). Thus, cluster 3 will lose its last layer. The node connecting it with cluster 4 will be an orphan cluster with the identity of its orphan node. The node connecting cluster 3 with cluster 1 will be a leaf in cluster 1. Note that in other regions of the graph, there are other clusters which are fighting against other clusters. Thus, many clusters can grow in parallel.

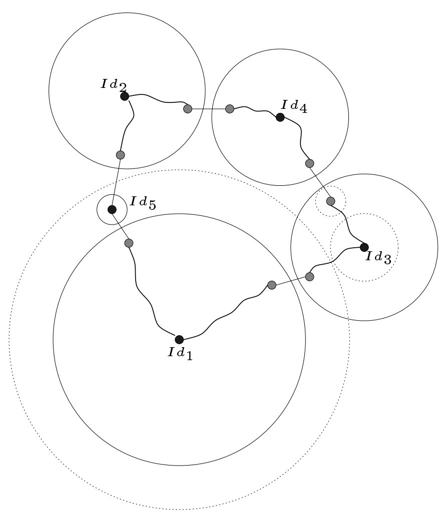


Figure 3: An example of competition between clusters

3.3 Detailed Description and Implementation of the Algorithm

In this section, we give an implementation of our algorithm using messages. We also give a complete description of how the clusters manage conflicts between each others. In next paragraphs, we give actions to be performed by each node according to its state. One shall keep in mind three points. First, a cluster is constructed in a layered fashion. Second, a cluster can add a new layer if its identity is bigger than the identities of all clusters at distance one or two from it and if the new layer satisfies the sparsity condition. Third, a root node can not know what's happening on the borders of its cluster, that's why, it always waits for informations from the leaves before taking any decision. However, cluster leaves can not know the global state of their cluster, that's why they always inform the root of the result of the computations done locally and waits for orders from their root.

Preliminary remarks

First, each node of the graph may use some variables when doing a computation step. The most significant variable is *State* which indicate the state of a node and so the actions the node must do. This variable may have five values : *Root*, *Leaf*, *Orphan*, *Relay* and *Final*.
The other variables a node may need are :

- *Brothers* : neighbors belonging the last layer of the node cluster. This avoids sending exploration messages to neighbors that are already in the current cluster.
- *Uncles* : neighbors belonging layer before the last of the node cluster (node that could have been fathers. This avoids sending exploration messages to neighbors that are already in the current cluster.
- *Nephews* : neighbors that are sons of a brother.
- *Conquerors* : A node may be explored by more than one node belonging to a conqueror cluster. In case of the neighbor cluster wins globally and invade the current cluster, the leaf chooses randomly one node between the conquerors to be its father and marks the other nodes as its uncles.
- *Sons* : chldren of a node in its cluster.
- *Final_States* : this variable identifies all neighboring nodes that belong to a final cluster.
- *Run* : This variable indicates that the node is still alive and isn't yet part of a finished cluster. The algorithm still runs.
- *Root_Id* : the root (cluster) identity.
- *Father* : the link identifing the node father in the cluster.
- *Winner_Root* : indicates for a leaf the identity of the potentiel winner cluster. This serves only if the node looses against a neighbor.
- *Winner_Father* : the link that identifies the potential winner father. This serves only if the node looses against a neighbor.

Figure 6 depicates the algorithm scheme.

Root nodes

A root node has three activities. First, it evaluates at each iteration the sparsity condition. Second, it informs the leaf nodes if the sparsity condition always holds. If so, the root node broadcats a notification message *NEW* to all its leaf nodes in order to begin a new exploration. On the other hand, if the sparsity condition does not hold, the root node knows that the construction of the cluster is finished and informs both the leaf and the relay nodes by broadcasting a *BACK* message. Third, if a new exploration has been initiated (after broadcasting a *NEW* message), a root node waits for the response of its leaves in order to know if the new exploration has succeeded. There are three possible cases :

1. If the leaves didn't find any new node to add (all neighboring nodes are in final states or there are no more nodes in the graph), the root node only receives *STOPPED* messages. In this case, the root nodes broadcast a *STOP* message informing all the nodes that the cluster construction is terminated.

2. If the new exploration has succeeded (*i.e* : all leaf nodes have locally won), the root informs the leaves that the cluster has globally won and broadcasts an *UP* message. In this case, the new layer will be added and the root will re-evaluate the sparsity condition for the cluster with the new added layer.

3. If the new exploration has not succeeded (*i.e* : at least one leaf node has lost against a neighboring cluster), the root informs all the leaf nodes by broadcasting an *ANNUL* message. Then, the root node waits for the leaf responses. There are two cases : Either at least one leaf node is invaded by another cluster and it informs its root by sending back a *BYE* message. Or all leaf nodes have resisted to neighboring cluster attacks (none of the neighboring cluster has succeeded in invading the current cluster), and they all send back to their root an *OK* message. In the first case, the cluster must give up its last layer, so the root brodcasts a *DOWN* message ordering all leaves to become *orphans*. In the second case, it broadcasts an *OK* message saying that none of the leaves was invaded.

Note that Figure 7 depicates the actions to be performed by the root.

Leaf nodes

First, we precise that a leaf node does not always belong to the last layer of a cluster. For example, a leaf node may not have neighbors belonging to other clusters that can be explored. Locally, this leaf can not add new nodes to its cluster, but other leaves belonging to the same cluster can continue to explore new layers. So globally, the construction of the cluster can continue even if some leaves cannot explore new regions. Each node has a variable h that indicates the width of the node in the tree spanning the cluster. If h equals 1 then the leaf nodes know that they belong to the last layer otherwise they know that they can not expand anymore.
When a node becomes a leaf in a new cluster, it only knows that its new cluster has won all its neighboring clusters but it doesn't know if the sparsity condition is verified. So a new leaf node always begins by sending the number 1 to its parent (which was a leaf and becomes a relay node). Thus, the parent computes the number of its children and gradullay the root node can compute $|\Gamma(S)|$. If the sparsity condition does not hold, the root broadcats a *BACK* message to its leaves. When receiving this message, a leaf node which belongs to the final layer knows that it must leave its new cluster and it becomes an orphan cluster (with its own identity). The leaves that do not belong to the last layer know that the construction of the cluster is finished. On the other hand, if the sparsity condition still holds, then the leaves must receive a *NEW* message. This message indicates that the root node is ordering the leaves to begin a new exploration. Such an exploration always begins by determinating whether the identity of the cluster is the biggest among those at distance 2. This is done using an election technique in a ball of radius 2 [23] : A leaf node sends its cluster identity to its neighbors and it waits for the identities of its neighboring clusters. Then, it computes the maximum identity of its neighbors including the identity of its own cluster and sends this maximum to its neighbors. Conversely, it waits for the maximum identity of its neighbors. If all the identities sent by the neighbors are equal to the identity of the node cluster, then the node has locally won against all neighbors at distance 2. More precisely, three cases may occur :

1. The leaf can not explore new regions. It sends back a *STOPPED* message to the root.

2. The leaf wins against neighboring clusters. It sends back to the root a *YES* message.
3. The leaf loses and sends back to the root a *NO* message.

Then, the leaf waits for orders from the root node in order to know what happens globally on the cluster. There are three cases :

1. None of the cluster leaves has succeeded in initiating a new exploration : a leaf node receives a *STOP* message. Thus, this leafs knows that the cluster construction is finished and switches its state to *final*.
2. All leaves have succeeded the new exploration and won all neighbors at distance 2 : each leaf node receives an *UP* message from the root. In this case, the leaf sends a *JOIN* message to all the neighboring nodes (leaves in other clusters) in order to inform them that they must join the new cluster and then switches to a relay state.
3. At least one leaf has not succeeded the exploration (*ANNUL* message from the root). Thus, the leaf sends an *ANNUL* message to all neighboring nodes in order to inform them that they are not invaded by the current cluster. The leaf then waits to know if it will be really invaded by a neighboring cluster. In this case, it must receive a *JOIN* message from the neighboring winner cluster and it sends back a *BYE* message to its roots, waits for an acknowledgment from the root and then joins its new cluster as a leaf. On the other hand, if none of the neighboring cluster has succeeded in invading the leaf (*ANNUL* message), the leaf sends back to its root an *OK* message. At this stage, leaves (except those who have received a *JOIN* message) do not know whether their cluster is being invaded or not (only the root does so). That's why, leaves wait for either an *OK* or a *BYE* message from the root. In the former, the leaf state won't change and the node is still a leaf in the same cluster. In the latter, the leaves that have not been invaded know that their cluster was invaded and become *orphan* nodes once again.

Note that Figures 8 and Figures 9 depicate the leaf actions.

Orphan nodes

An orphan node acts both like a leaf and a root node. It tries to invade neighboring nodes. If it succeeds, it becomes a root node in its new cluster. If it is invaded by an other cluster (*JOIN* message), it becomes a leaf in the new winner cluster. Otherwise, it re-tries to invade its neighbors. If there are only neighbors belonging to finished clusters, the node switches to a final state.
Note that Figure 10 depicates the action to be performed by an orphan node which are very simiar to those performed by a root node.

Relay nodes

The main activity of a relay node is to forward informations between the root and the leaves. We precise that each node knows locally who are its children and its father.
When a relay node receives a message from its father, it simply forwards it to its children. If the message is a *BACK* or a *STOP* message, then the node knows that the cluster construction is finished and it switches to a final state. If the message is an *UP* message, the node knows that there is a new layer that joins its cluster and thus the width of the node is incremented by one ($h = h + 1$). If the message is a *DOWN* message then the relay nodes know that their cluster was invaded and it has lost the last layer ($h = h - 1$). In this case, if a relay node belongs to the layer before the last one ($h = 2$) then the relay node switches

its state to a leaf state.
On the other hand, if a relay node receives a message from its children, it can know at which step the leaves are (exploration of a new layer : *YES* or *NO* or *STOPPED* messages, resistance against neighbors attacks : *OK* or *BYE* messages, and computation of the sparsity condition). In all cases, the relay node can easily know which type of message it must forward to its root.
Figure 11 gives a more detailed idea about the actions to be performed by a relay node.

Final nodes

When a node is in a final state, it does not participate anymore in the algorithm computations because the construction of its cluster is finished. Each node knows which of its neighbors belong to a finished cluster and thus stops communicating with these final nodes.

3.4 Analysis of the Algorithm

Let us first prove the correctennes of the previous algorithm. The first two proofs are briefly sketched.

Theorem 3.1 *Our algorithm emulates the* $Basic\_Part$ *algorithm.*

Proof By construction, before definitively adding a new layer, the root node always verify if the sparsity condition of algorithm $Basic\_Part$ is correct. Conversely, if a cluster leaf is invaded, the cluster loses all of its other leaves : it loses the whole last layer. The resulting not yet finished cluster still respects the sparsity condition. Thus, the constructed partition satisfies the sparsity and locality properties of algorithm $Basic\_Part$. Furthermore, once the construction of a cluster is finished, each cluster node switches to a final state and definitively stops taking part of the partition construction. In other words, a finished cluster can not be invaded by other actif clusters : the decomposition we construct is a partition. ∎

Theorem 3.2 *Our algorithm terminates.*

Proof By construction, when a node switches to a final state, it stops communicating with all of its neighbors and it can not switch to another actif state. Each node also knows which of its neighbors are in a final state and does not communicate with these neighbors. So, to prove that our algorithm terminates, we prove that all nodes must be at some moment in a final state. To enter in a final state a node must be at some moment in a cluster which reaches the sparsity condition or which has only finsihed clusters around it (*i.e* : there is no new layer to add). Note that the sparsity condition will not be statisfied if the cluster radius is $k-1$. After a cluster adds $k-1$ layers it must stop growing : its construction is finished. So a cluster can not grow indefinitely. By construction, the cluster having the biggest identity in the graph always wins against its neighbors. This cluster always succeeds adding new layers until it can not grow any more. Thus, the nodes inside this cluster must switch to a final state at some moment. Now, the cluster having the biggest identity in the subgraph obtained by considering only not yet finished clusters always succeeds in its explorations because the only cluster that could have stopped it has finished. To sum up, at any moment, we are sure that the cluster having the biggest identity in the subagraph of active nodes always wins. If there are new added nodes, then these nodes can no longer be invaded by other clusters . Thus, the nodes in this cluster finish beeing in a final state. Any node of the graph finish by being invaded at some moment by the cluster having the biggest identity and so it is sure to enter definitively in a final state. As long as the construction goes on, the number of active nodes decreases up to the moment where there are no more active nodes : the algorithm terminates. ∎

Message and Time Complexity

Theorem 3.3 *In the worst case, the message complexity of algorithm $Dist\_Part$ is :*

$$Message(Dist\_Part) \; = \; O(\Delta|V|^2)$$

Proof The technique used in our algorithm is similar to the technique used for network synchronization with synchronizer γ in the sens that a cluster can not invade a neighbor before its neighbors give their acknowledgements. A leaf must first inform its current cluster that it is being invaded then waits for the acknowledgment of its root before joining a new cluster. In other words, there is a kind of synchronization between clusters. This allows us to decompose our algorithm in many virtual phases in order to compute the message complexity. At each phase, the graph is decomposed into many clusters each of them can add a new layer, go back (lose its last layer), or preserve it against neighbors attacks. If we consider a single cluster $|S|$ at some phase p, we can decompose the activity of a cluster into two groups. First, there are messages going from the root to leaves and conversely in order to brodcast or forward informations. This is done using the BFS spanning tree constructed as long as the cluster grows. Second, there are messages used by each leaf to try invading its neighbors. So, the number of message used at a phase p is :

$$\sum_{S \; not \; final} O((|S|-1) + |Edges\_Ext(S)|)$$

with $|Edges\_Ext(S)|$ the number of edges connecting the cluster S with its neighbors and allowing the cluster to communicate with them. A first bound can be obtained considering that $\sum_S |S| \leq |V|$ and $\sum_S |Edges\_Ext(S)| \leq |E|$. So, the total message complexity of our algorithm is :

$$Message(Dist\_Part) \; = \; \sum_p O(|V|) \quad + \quad \sum_p O(|E|)$$

To evaluate the number of phases needed before the algorithm terminates, we consider r_f (resp. n_f) the radius (resp. the number of nodes) of the cluster S_f obtained at the end of the algorithm. Let SF_p be the set of all finished clusters at phase p and SF the set of cluster constructed at the end of our algorithm. In the worst case, the number of phases needed to construct the decompostion is $\sum_{S_f \in SF} r_f$. Since $r_f \leq n_f$ and $\sum_{S_f \in SF} n_f = |V|$, we can conclude that, in the worst case, the number of phases is equal to $|V|$ and we obtain the following message complexity :

$$Message(Dist\_Part) \; = \; O(|V|^2) \quad + \quad O(|V| * |E|)$$

A more careful analysis of our algorithm consists in considering node degrees and the contribution of each node in the message complexity. Let us focus only on messages exchanged on the border of a cluster which cost much more than communications in the interior of the cluster. Let S_{max} be the cluster with the biggest identity. In the worst case, the contribution of active nodes on one phase is :

$$\sum_{v \notin S_f, v \notin S_{max}} O(degree(v))$$

We can note that, at each phase, there is at least one node that is definitively part of a futur final cluster S_{max}. This node is definitively inside of cluster S_{max} and won't fight no more against other nodes. So, it won't send any further messages on the edges connecting it with its neighbors. Let Δ be the maximum degree of the graph nodes, the number of messages used in the border of active clusters in the worst case is : at phase one : $O(\Delta|V|)$, then at phase two : $O(\Delta(|V|-1))$, more generally at phase p : $(\Delta(|V|-p+1))$. Thus, we obtain the following message complexity :

$$\begin{aligned} Message(Dist\_Part) &= O(|V|^2) + \sum_p O(\Delta(|V| - p + 1)) \\ &= O(|V|^2) + O(\Delta|V|^2) \end{aligned}$$

Finally, the message complexity of algorithm $Dist\_Part$ is :

$$Message(Dist\_Part) = O(\Delta|V|^2)$$

Theorem 3.4 *In the worst case, the time complexity of algorithm $Dist\_Part$ is :*

$$Time(Dist\_Part) = O(|V|)$$

Proof First, note that our algorithm can be implemented in a synchronous and in an asynchronous environment : we don't use any global clock. In this paragraph, we suppose that the algorithm runs on a synchronous environment. Thus, the time complexity is the maximum number of pulses from the start of the algorithm to its termination. Let SF be the set of all finished clusters S_f and let r_f be the radius of a finished cluster S_f. We denote by $Time(S_f)$ the number of pulses from the moment the root node of cluster S_f becomes the biggest over the whole active graph nodes (*i.e* : not in finished clusters) up to the termination of S_f construction. From now, we consider that clusters are never constructed at the same time. This is the worst case in terms of time complexity. Then, the time complexity is :

$$Time(Dist\_Part) = \sum_{S_f \in SF} Time(S_f)$$

To decide if a layer will be added or not, a cluster must be traversed at most six times. In addition, a node can join a new cluster only if it informs its root and receives an acknowledgement. Let i, $0 < i \leq r_f$ be the number of layers in the cluster S_f before its construction is finished and r_{max_i} be the maximum radius over all S_f neighboring clusters at step (iteration) i of the cluster construction. Thus :

$$Time(S_f) \leq \sum_{0 < i \leq r_f} O(i + r_{max_i})$$

Using Theorem 2.1, we have $r_f \leq k - 1$ and $r_{max_i} \leq k - 1$ and we can conclude that :

$$Time(S_f) = O(r_f^2 + kr_f) = O(kr_f)$$

Using the fact that $\sum_{S_f \in SF} r_f \leq |V|$, we get :

$$Time(Dist\_Part) = O(k|V|)$$

To improve this bound, we make the following modification in our algorithm : a cluster can win only if it first has the biggest radius and second the biggest identity (we use the lexicographical order). This does not affect the message complexity and improves the time complexity. In fact, with this modification, we have $r_{max_i} \leq r_f$. Note also that using the sparsity condition and in the relevant range $k \leq log(n)$ (after which $n^{1/k} = O(1)$), we have the following :

$$n_f \geq n^{r_f/k} \Rightarrow r_f \leq \frac{k}{log(n)} log(n_f) \Rightarrow r_f \leq log(n_f)$$

Finally,

$$
\begin{aligned}
Time(Dist\_Part) &= \sum_{S_f \in SF} \sum_{0<i\leq r_f} O(i + r_{max_i}) \\
&\leq \sum_{S_f \in SF} O(r_f^2) \\
&\leq \sum_{S_f \in SF} O(log(n_f)^2) \\
&\leq \sum_{S_f \in SF} O(n_f) \\
&\leq O(n)
\end{aligned}
$$

Note that using both the cluster radius and identity to break the symmetry, we improve the time complexity and we privilege the construction of clusters with bigger radius. We also can use cluster size with the radius and the identity to resolve conflicts between adjacent clusters. ∎

Note that, in the worst case there is only one node that definitively stops participating in new explorations (stops being at the border). The complexity should be better if there are many nodes that definitively stop participating in new explorations (in other words, if there are many nodes at the new layers and if there are many clusters that grows at the same time).
It would be interesting to compute the average message complexity. This may be a hard task because the execution of our algorithm depends both on the graph topology and on the distribution of node identities. We have no criterion to say how the clusters are contructed or how many clusters are constructed at the same time. Note that there are some configurations for which only one cluster is constructed at once and for which the worst case is achieved. This is the case for the graph given in Figure 4 with a bad distribution of node identities. For this graph we prove that the average message complexity is still in $O(n^2)$. Computing the average complexity for general graphs and for any distribution of node identities remains an open question.

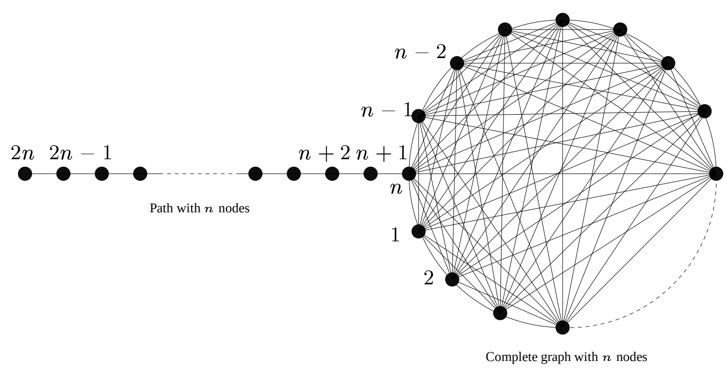


Figure 4: An example of bad node distribution

Anonymous Graphs

The main idea of algorithm $Dist\_Part$ is to allow clusters to grow in parallel and to break the symmetry using node identities. This is no more possible when considering an anonymous graph.
One idea is to attemp to turn out our algorithm into a randomized one for anonymous graphs where at each new exploration, the root node of each cluster generates randomly an identity and forwards it to its leaves. If we suppose that each node selects an identity uniformly at random from a large set of integers, we

can suppose that the identities generated by neighboring clusters are different with high probability. Our algorithm remains the same (the only modification is to forward the generated identity at each iteration) and the symmetry could be broken in the same way as in the deterministic algorithm.

4 Distributed Deterministic Sparse Cover Algorithms for Synchronizers γ_1 and γ_2

The decompostion given in Section 2 can be used for synchronizer γ. The main idea of this synchronizer is to synchronize first the nodes belonging to one cluster. This is done using a broadcast convergcast technique upon the tree spanning the cluster. Then, neighboring clusters are synchronized using the links between them. The root node of each cluster is used to control all of these synchronization steps. As the decompostion algorithm $Basic\_Part$ produces clusters with low radius and low inter connexion edges, the message complexity of the synchronization can be improved while mainting a reasonable time complexity (logarithmic). There are also other important synchronizers derived from synchronizer γ mainly synchronizers γ_1 and γ_2 [24, 27, 12] which use two different types of covers. In the next two sections, we show how it is possible to construct the covers needed for synchronizers γ_1 and γ_2 in a truly parallel way.

4.1 Synchronizer γ_1

The first synchronizer γ_1 needs a sparse cover where each two neighboring nodes must belong to a cluster and where the clusters may overlap. We can distributively construct such a decompostion using our technique. In fact, in our algorithm, layers are first added, then the cluster root computes the sparsity condition and decides if the last layer is actually added or rejected. So, the last layer is always computed. To construct the cover needed for synchronizer γ_1, it suffices to allow the last layer to be part of the cluster when the sparsity condition is no longer satisfied. By simply doing the modifications listed below, we can construct a cover needed for synchronizer γ_1 in a trully parallel way while mainting the same message and time complexity :

1. When the construction of a cluster is finished, because of the sparsity condition, nodes which are inside of the cluster (relay nodes) switch to final states but still mark children in the rejected last layer.

2. In the same way, rejected nodes in the last layer mark themselves as part of the current cluster (they still know how are their fathers in the curent cluster), and then switch to an orphan state.

4.2 Synchronizer γ_2

The second synchronizer γ_2 uses a sparse cover that guarantees that for each node v, there is a cluster S which contains $N(v)$ (*i.e* : v and all its neighbors). To construct such a cover, we allow clusters to grow in parallel and we use the identities of clusters to manage conflicts. However, we must consider at each iteration two layers at the same time. In fact, at the beginning of the execution of the algorithm each cluster tries to add two layers before it begins verifing the sparsity condition. This enables each cluster to have two reserved layers. First, there is a layer that will allow the cluster to compute the sparsity condition : layer l_i. This is the same layer that we count in the basic distributed sparse decomposition algorithm $Dist\_Part$. But now, it is no longer the last layer of the cluster but the layer before the last one.

Second, the last layer l_{i+1} enables a cluster to compete against neighboring clusters and it ensures that the neighborhoods of all nodes in layer l_i are in the current cluster. There are three important points :

1. If the sparsity condition is satisfied, then the cluster can begin a new exploration. Leaves on layer l_{i+1} try to invade a new layer. If this is possible, layer l_{i+1} becomes layer $l_{i'=i+1}$ and the new added layer becomes the new $l_{i'+1}$ layer. Otherwise, if the sparsity condition on layer l_i is not satisfied, then the construction of the cluster is finished. The finished cluster contains not only all layers $l_{j<i}$ but also the two layers l_i and l_{i+1}. The main difference between this three kind of layers is that layers $l_{j<i}$ are definitively out of any computation : they switch to a final state. At this point, the neighborhoods of all nodes in layer l_i are covered by the finished cluster. Thus, the nodes in layer l_i verify the property needed for synchronizer γ_2 and their neighborhood does not need to be covered by any other cluster. But, neighborhoods of nodes at layer l_{i+1} may not yet be covered by a cluster, that's why nodes at layer l_i must continue participating in the algorithm computations but should not be leaders in their cluster. So nodes at layers l_i become Orphan clusters with identity -1 in order to allow other not finished clusters to grow and to cover the neighborhoods of nodes in layer l_{i+1}. In other words, nodes in l_i must not be an obstacle to the growth of other clusters. On the other side, nodes in layer l_{i+1} become Orphans. Thus, their neighborhoods can be covered by other clusters.

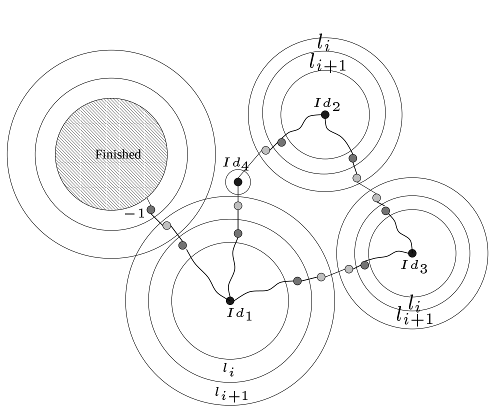
2. If a new exploration fails *i.e* there is a cluster at distance 1 or 2 that has a biggest identity than the current clusters. Either the winner lost against another neighboring cluster and the current cluster is not invaded or the current cluster is invaded. In the first case, the cluster does nothing and simply retries a new exploration. In the second case, the current cluster must yield its last layer. Invaded nodes in layer l_{i+1} become part of the last layer $l_{i_{win}+1}$ of the winner cluster. Nodes in layer l_{i+1} which have not been invaded become orphan nodes and begin a new exploration using their own identities. Layer l_i becomes the last layer $l_{i'+1=i}$ and layer l_{i-1} becomes layer $l_{i'=i-1}$ in the current cluster (the one who loses). Then the current cluster can begin a new exploration again.

3. When the construction of a cluster is finished, nodes at distance at least 2 from the border of the cluster (*i.e* : layers $l_{j\leq i-1}$) switch to final states. In fact, layer l_{i-1} of a finished cluster acts as a barrier that protects the finished cluster. Layers $l_{j\leq i-1}$ are commonly called the *Kernel* of the cluster. This is a reccurent classical feature and technique used for constructing coarsening covers [9, 10, 26].

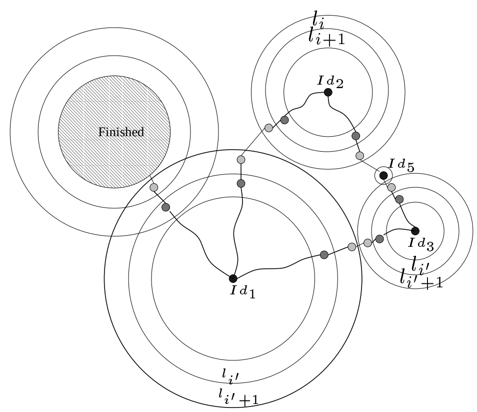
Note that this algorithm constructs the same decomposition (needed for synchronizer γ_2) as in [24]. The main difference is that clusters are constructed in parallel. The time and message complexity of this algorithm remains the same as for the $Dist\_Part$ algorithm. In fact, the sparsity condition is computed in the same way and the new exploration too. The main difference is that the computations are done on the two last layers (we just need to mark nodes belonging at the last layer as part of the cluster, before verifying the sparsity condition).

An example of cluster growth for synchronizer γ_2

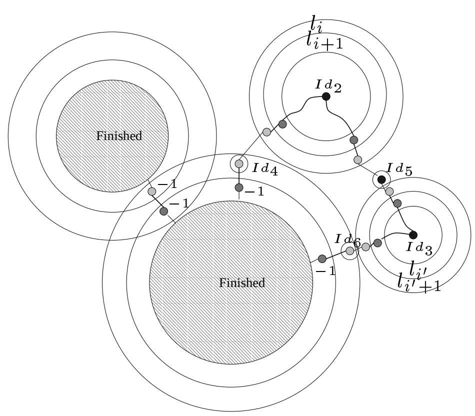
In Figure 5, we give an illustration of how the cover is constructed for synchronizer γ_2. Note that there are four not yet finished clusters : 1, 2, 3 and 4 with identities Id_1, Id_2, Id_3 and Id_4. There is also a finished cluster neighboring cluster 1. The finished cluster has stopped its expansion because the sparsity condition was not satisfied. The nodes at layer l_i of the finished cluster (first part of Figure 5) still participate in the computation with identity -1, all other nodes before this layer are definitively in a final state. There is also a node in layer l_{i+1} of the finished cluster that is part of layer l_{i+1} of cluster 1. Suppose now that the layer l_i of each cluster satisfies the sparsity condition, then all clusters (except the finished one) will try to grow. Suppose that cluster 1 have the biggest identity. Cluster 2 can not grow because cluster 1 is at distance two of it and has a bigger identity but it will not be invaded. Cluster 1 will invade both clusters 3 and 4. Cluster 4 is orphan and it simply joins the last layer of cluster 1. Cluster 3 will lose its last layer l_{i+1}. The invaded nodes of cluster 3 join cluster 1 and the other nodes which have not been invaded become orphan clusters. Note also that the node with identity -1 of the finished cluster is invaded by cluster 1. This guarantees that the neighborhood of the son of this node in the finished cluster is now covered by cluster 1. Now the new exploration is finished, cluster 1 will verify the sparsity condition. If this condition is satisfied, a new exploration will begin and clusters 2 and 3 will be invaded. Note that nodes of the finished cluster which are in layers before l_i will not be invaded by cluster 1 : these nodes are definitively out of computations (it is as if they do not exist). If the sparsity condition is not satisfied which is the case in the third part of Figure 5, the construction of cluster 1 is finished. Nodes at layer $l_{i'}$ will take identity -1 and nodes in layer $l_{i'+1}$ will be orphan nodes except those who are already in layer l_i of an other finished cluster (those whose neighborhoods are covered by another cluster). Note that the two finished clusters we have constructed overlaps (they have a common edge). Now, the cluster having the biggest identity won't be stopped by cluster 1 and for sure will succeed in adding new layers.



(a) Before cluster 1 expansion



(b) After cluster 1 expansion



(c) After construction of cluster 1 is finished

Figure 5: An example of a cluster expansion for cover needed for γ_2

Table 1: Experimental message and time complexity

Graph Size	Min Msg	Max Msg	Average Msg	Min Time	Max Time	Average Time
100	25342	40388	32945	66	204	123
200	23673	78610	43296	48	421	139
300	31536	68632	42766	72	446	166
400	33290	213901	61573	67	714	215
500	36188	482238	121576	103	2210	411
600	38654	389143	140040	125	1291	611
700	33759	764258	207606	84	2140	660
800	34917	288478	91488	91	1165	350
900	35472	462385	132698	94	1588	474
1000	37338	526368	149547	87	1652	505

5 Experimentation

Although our algorithm works in a complete parallel way (clusters are constructed in parallel), the time complexity $O(|V|)$ of our algorithm is the same as the one of algorithm in [24]. This is because we can find a particular distribution of node identities for which only one cluster grows at once. This is, for example, the case if the graph contains a path in which two neighboring nodes have repectively identities Id and $Id + 1$. In this case, only the cluster containing the strongest node at the border of the path can grow. Any other cluster containing any other node of the path can not add any new layer because it will lose against its neighbor. But, in general, we note that there are more than one cluster that can be constructed at the same time. So we have made several experiments to measure the time complexity of our algorithm $Dist\_Part$ in order to estimate in practice how fast it constructs a network partition.
We can make the same remarks for the message complexity. In fact, the worst case is when only few clusters are constructed in parallel. This means that only messages used by the strongest clusters really serves and all other messages used by weaker clusters do not serve at all. These messages increase considerably the message complexity. As we said before, the complexity depends on the distribution of node identities.

Our experiments have been done using a software platform for the Visualization and the Simulation of Distributed Algorithms : ViSiDiA [18, 15, 13, 14]. ViSiDiA is a practical tool which helps implementing distributed algorithms by providing basic primitives for sending and receiving messages. It also enables to visualize the execution of an algorithm on the fly. It gives some tools for counting messages. A distributed version of these software allows to distribute the computations using a network of machines which enables algorithm simulation on huge graphs for example. Our experimental resulats using this tool are regrouped in Table 1. We have used graphs with different sizes going from one hundred nodes to one thousand nodes. For each size, we have made several experiments changing both the number of edges, the maximum degree Δ and the identity distribution. Note that on average, the time complexity is in most cases lower than $|V|$. Note also that the message complexity both in the worst case, in the best case and on average is still lower than $|V|^2$. This ensures the fact that in practice our algorithm is efficient both in time and message complexity. These experiments also allows us to say that it is possible to refine the bound on the message complexity. We conjecture that on average the message and time complexity of our algorithm is respectively bounded by $O(D * |E|)$ and $O(D)$ where D is the diameter of the graph.

References

[1] Y. Afek and M. Ricklin. Sparser : a paradigm for running distributed algorithms. *Journal of Algorithms*, 14:316–328, 1993.

[2] B. Awerbuch. Complexity of network synchronization. *Journal of the Association for Computing Machinery*, 32:804–823, 1985.

[3] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Low-diameter graph decomposition is in nc. *Random Structures and Algorithms*, pages 441–452, 1994.

[4] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Fast distributed network decompositions and covers. *Journal of Parallel and Distributed Computing*, 39:105–114, 1996.

[5] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28:263–277, 1998.

[6] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Poltkin. Network decomposition and locality in distributed computation. *Symposium on Foundations of Computer Science*, pages 364–369, 1989.

[7] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. *Proceedings of the 24th annual ACM symposium on Theory of computing*, pages 571–580, 1992.

[8] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks. Adapting to asynchronous dynamic networks. *Proceedings of the 24th annual ACM symposium on Theory of computing*, pages 557–570, 1992.

[9] B. Awerbuch and D. Peleg. Sparse partitions. *31st IEEE Symposium on Foundations of Computer Science*, pages 514–522, 1990.

[10] B. Awerbuch and D. Peleg. Routing with polynomial communication-space trade-off. *SIAM Journal on Discrete Mathematics*, 5:151–162, 1992.

[11] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the Association for Computing Machinery*, 42:1021–1058, 1995.

[12] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. *IEEE Symposium on Foundations of Computer Science*, 2:503–513, October 1990.

[13] M. Bauderon, S. Gruner, and M. Mosbah. A new tool for the simulation and visualization of distributed algorithms. Technical Report 1245-00, LaBRI, 2000. Accepted in MFI'01, Toulouse, 21-23 May 2001.

[14] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI, 2002. http://www.labri.fr/visidia/.

[15] M. Bauderon and M. Mosbah. A unified framework for designing, implementing and visualizing distributed algorithms. *International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'02)*, 2002.

[16] F. Belkouch, M. Bui, L. Chen, and A. K. Datta. Self-stabilizing deterministic network decomposition. *Journal of Parralel and Distributed Computing*, 62:696–714, 2002.

[17] L. Cowen. *On Local Representations of Graphs and networks*. Ph. D Thesis, 1993.

[18] B. Derbel and M. Mosbah. Distributing the execution of a distributed algorithm over a network. *7th IEEE International Conference on Information Visualization, IV03-AGT.*, pages 485–490, 16-18 July 2003.

[19] I. Gaber and Y. Mansour. Centralized broadcast in multihop radio networks. *Journal of Algorithms*, 46:1–20, 2003.

[20] J.A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. COMPUT*, 27:302–316, February 1998.

[21] S. Kutten and D. Peleg. Fast distributed construction of small k-dominating sets and applications. *Journal of Algorithms*, 28:40–66, 1998.

[22] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. *Proceedings of the second annual ACM-SIAM Symposium on Discrete Algorithms*, pages 320–330, 1991.

[23] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Information Processing Letters*, 82:313–320, 2002.

[24] S. Moran and S. Snir. Simple and efficient network decomposition and synchronization. *theoretical computer science*, 243:217–241, 2000.

[25] A. Panconesi and A. Srinivasan. Improved distributed algorithms for coloring and network decomposition. *24th Annual ACM STOC*, pages 581–592, 1992.

[26] D. Peleg. *Distributed Computing, A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

[27] L. Shabtay and A. Segall. Low complexity network synchronization. *8th Internat. Workshop on distributed Algorithms*, pages 223–237, 1994.

Appendix A : Practical Implementation of Algorithm $Dist\_Part$

```
// variable initialization

Brothers = vecteur();
Uncles = vecteur();
Nephews = vecteur();
Conquerors = vecteur();
Sons = vector();
Final_States = vector();
State = Orphan;
Run = true;
Root_Id = MyId();
father = -1;

WHILE (Run) DO {
     Winner_Root = -1;
     Winner_father = -1;

     IF (State == Orphan) {
           // Actions
     } ELSE IF (State =  Root) {
           // Actions
     } ELSE IF (State =  Leaf) {
           // Actions
     } ELSE IF (State =  Relay) {
           // Actions
     } ELSE IF (State =  Final) {
          Run = false;
     }
}
```

Figure 6: General scheme of algorithm $Dist\_Part$

```
RECEIVE count FROM ALL Sons ;
COMPUTE S(h,h+1);      // sparsity condition for radius h+1
IF ( S(h,h+1) ) {      // sparsity condition holds
   SEND NEW TO ALL Sons ;
   RECEIVE Yes_No FROM ALL Sons;
   COMPUTE LC2, Cluster_Stopped;
   IF (Cluster_Stopped) {
      SEND STOP TO ALL Sons ;
      State = Final ;
   } ELSE {
      IF (LC2) {
         SEND UP TO ALL Sons ;
         h = h+1;
      } ELSE {
         SEND ANNUL TO ALL Sons ;
         RECEIVE msg FROM ALL Sons ;
         Bye = False ;
         IF (it exist (msg == BYE)) {
            Bye = True ;
         }

         IF (Bye) {
            SEND DOWN TO ALL Sons ;
            h = h-1 ;
            IF (h == 1) {
               State = Leaf ;
               COMPUTE Sons ;
            }
         } ELSE {
            SEND OK TO ALL Sons ;
         }
      }
   }
} ELSE {
   h = h-1;
   State = Final ;
   SEND BACK TO ALL Sons ;
}
```

Figure 7: Root actions

```
SEND 1 TO father ;
RECEIVE msg FROM Father ;

IF ( msg == NEW ) {
   SEND Root_Id TO ALL neighb
                 NOT IN father, Uncles, Nephews, Final_states ;
   RECEIVE Root_neighb FROM ALL neighb
                           NOT IN father, Uncles, Nephews, Final_states ;
   COMPUTE Cluster_Stopped, Conquerors, Winner_father, Winner_Root ;

   IF (Winner_Root < Root_Id) {
      SEND Root_Id  TO ALL neighb
                    NOT IN Final_States, Uncles, Nephews;
   } ELSE {
      SEND Winner_Root TO ALL neighb
                       NOT IN Final_States, Uncles, Nephews ;
   }
   RECEIVE Root_neighb FROM ALL neighb
                           NOT IN Final_States, Uncles, Nephews ;
   COMPUTE Max_Root_neighbors;
   LC2 = False;
   IF (Max_Root_neighbors > Root_Id) {
      LC2 = True;
   }

   IF (Cluster_Stopped) {
      SEND Stopped TO father ;
   } ELSE IF (LC2) {
      SEND YES TO father ;
   } ELSE {
      SEND NO TO father ;
   }

   RECEIVE msg FROM father ;
   IF (msg == UP) {
      SEND UP TO ALL neighb
               NOT IN father, Uncles, Final_states ;
      RECEIVE msg FROM ALL neighb
                      NOT IN father, Uncles, Nephews, Final_states ;
      RECEIVE 1_Or_0 FROM ALL neighb
                         NOT IN father, Uncles, Nephews, Final_states ;
      COMPUTE Sons ;
      IF (Sons is not Empty) {
         h = h+1;
         State = Rely;
      }
   } ELSE IF (msg == STOP) {
      State = Final ;
```

Figure 8: Leaf actions

```
    } ELSE IF (msg == ANNUL) {
        SEND ANNUL TO ALL neighb
                    NOT IN father, Uncles, Nephews, Final_states ;
        RECEIVE msg FROM ALL neighb
                        NOT IN father, Uncles, Nephews, Final_states ;
        Bye = False ;
        IF (it exist (msg == BYE)) {
            Bye = True ;
        }

        IF (Bye) {
            Uncles = Conquerors ;
            SEND BYE To Father ;
            RECEIVE msg FROM father ;
            SEND 0 To Winner_father ;
            SEND 1 TO ALL Conquerors;
            Root_Id = Winner_Root ;
            father = Winner_father;
        } ELSE {
            SEND OK To Father ;
            RECEIVE msg FROM father ;
            IF (msg == DOWN ) {
                IF ( h == 1 ) {
                    State = Orphan ;
                    Root_Id = MyId();
                    father = -1;
                } ELSE {
                    h = h-1;
                }
            }
        }
    }
} ELSE {
    IF (h == 1) {
        COMPUTE Final_States ;
        father = -1 ;
        Root_Id = MyId();
        State = Orphan ;
    } ELSE {
        h = h-1;
        State = Final ;
    }
}
```

Figure 9: Leaf actions

```
SEND Root_Id TO ALL neighb
              NOT IN Final_States, Uncles, Nephews ;
RECEIVE root_neighbors FROM ALL neighb
                          NOT IN Final_States, Uncles, Nephews;
COMPUTE Winner_Root, Winner_father, Conquerors, Cluster_Stopped ;
IF (Cluster_Stopped) {
      State = Final ;
} ELSE {
   IF (Winner_Root < Root_Id) {
      SEND Root_Id TO ALL neighb
                    NOT IN Final_States, Uncles, Nephews;
   } ELSE {
      SEND Winner_Root TO ALL neighb
                        NOT IN Final_States, Uncles, Nephews;
   }
   RECEIVE Root_neig FROM ALL neighb
                        NOT IN Final_States, Uncles, Nephews ;
   COMPUTE Max_Root_neighbors;
   LC2 = False;
   IF (Max_Root_neighbors > Root_Id) {
      LC2 = True;
   }
   IF (LC2) {
      h = h+1;
      SEND UP TO ALL neighb
               NOT IN Final_States, Uncles, Nephews ;
      RECEIVE msg FROM ALL neighb
                      NOT IN Final_States, Uncles, Nephews ;
      RECEIVE 1_Or_0 FROM  ALL neighb
                     NOT IN Final_States, Uncles, Nephews ;

      COMPUTE Sons ;
      State = Root ;
   } ELSE {
      SEND ANNUL TO ALL neighb
                  NOT IN Final_States, Uncles, Nephews ;
      RECEIVE msg FROM ALL neighb
                  NOT IN Final_States, Uncles, Nephews ;
      Bye = False ;
      IF (it exists  (msg == UP)) {
         Bye = true ;
      }
      IF (Bye) {
         Uncles = Conquerors;
         SEND 0 TO Winner_father;
         SEND 1 TO ALL conquerors
                  NOT IN Final_States, Uncles, Nephews ;
         Root_Id = Winner_Root ;
         Father = Winner_father ;
         State = Leaf ;
         h = 1 ;
      }
   }
}
```

Figure 10: Orphan node actions

```
RECEIVE msg FROM ANY Door;
IF (Door == Father) {
   SEND msg TO ALL Sons ;
   IF (msg == UP) {
      h = h+1 ;
   } ELSE IF (msg == DOWN) {
      h = h-1 ;
      IF (h == 1) {
         COMPUTE Sons ;
         State = Leaf ;
      }
   } ELSE IF (msg == BACK) {
      h = h-1 ;
      State = Final ;
      IF (h == 1) {
         COMPUTE Sons ;
         State = Leaf ;
      }
   textbf} ELSE IF (msg == STOP) {
      State = Final ;
   }
} ELSE {
   IF ( (msg == BYE)    (msg == OK) ) {
      IF (msg == BYE) {
         Bye = True ;
      } ELSE {
         Bye = False ;
      }
      RECEIVE message FROM ALL Sons
                                    NOT IN Door ;
      IF (it exists (message == BYE)) {
         Bye = True ;
      }
      IF (Bye) {
         SEND BYE TO father ;
      } ELSE {
         SEND OK TO father ;
      }
   } ELSE IF ((msg == YES)
              (msg == NO)
              (msg == STOPPED)) {
      Stopped = True ;
      IF (msg == NO) {
         No = True ;
         Stopped = False;
      } ELSE {
         No = False ;
         Stopped = False;
      }
      RECEIVE message FROM ALL Sons ;
                                    NOT IN Door ;
      IF (it exists (message == NO)) {
         No = True ;
         Stopped = False ;
      } ELSE IF(it exists(message==YES)){
         Stopped = False ;
      }
      IF (No) {
         SEND NO TO father ;
      } ELSE IF (Stopped) {
         SEND STOPPED TO father ;
      } ELSE {
         SEND YES TO father ;
      }
   } ELSE {
      RECEIVE message FROM ALL Sons
                                    NOT IN Door ;
      COMPUTE Count ;
      SEND Count TO father ;
   }
}
```

Figure 11: relay node actions