Sublinear Fully Distributed Partition with Applications*

Bilel Derbel†, Mohamed Mosbah‡, and Akka Zemmari‡

† LIFL, INRIA, Université des Sciences et Technologies de Lille
Bâtiment M3, 59655 Villeneuve d'Ascq Cedex - France
bilel.derbel@lifl.fr
‡LaBRI, Université Bordeaux I, ENSEIRB
351, Cours de la libération, 33405 - Talence - France
{mosbah,zemmari}@labri.fr

Abstract. We present new efficient deterministic and randomized distributed algorithms for decomposing a graph with n nodes into a disjoint set of connected clusters with radius at most $k-1$ and having $O(n^{1+1/k})$ intercluster edges.
We show how to implement our algorithms in the distributed $\mathcal{CONGEST}$ model of computation, i.e., limited message size, which improves the time complexity of previous algorithms [34,3,36] from $O(n)$ to $O(n^{1-1/k})$.
We apply our algorithms for constructing low stretch graph spanners and network synchronizers in sublinear deterministic time in the $\mathcal{CONGEST}$ model.

1 Introduction

Due to the constant growth of networks, it becomes necessary to find new techniques to handle related global information, to maintain and to update this information in an efficient way. A *Locality-Preserving (LP) network representation* [36] can be considered as an efficient data structure that captures topological properties of the underlying network and helps to design distributed algorithms for many fundamental problems: synchronization [34,41,7], Maximal Independent Set (MIS) [4], routing [5], mobile users [6], coloring [35] and other related applications [24,25,28,13,1]. In order to provide *efficient* solutions for these problems, it is important to construct LP-representations in a distributed way while maintaining good complexity measures.

The main purpose of this paper is to give an overview of some LP-representations of special interest and to show how to construct them efficiently in the distributed setting. More precisely, we focus on one important type of LP-representations called *clustered representations*. The main idea of a clustered representation is to decompose the nodes of a graph into many possibly overlapping regions called clusters. This decomposition allows us 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 criterion 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 a clustered representation. The second criterion attempts to measure the *sparsity level*. This criterion gives an idea about how the clusters are connected to each others. For instance, in the case of disjoint clusters, the number of intercluster edges is usually used to express the sparsity level. In the case of overlapping clusters, the average/maximum number of occurrences of a node in the clusters is usually used to express the sparsity (or the overlap) of the clustered representation.

* Some materials presented in this paper appeared in two extended abstracts published in the proceedings of IPDPS06 (20^{th} IEEE International Parallel & Distributed Processing Symposium) [20] and PDCS04 (16^{th} IASTED International Conference on Parallel and Distributed Computing and Systems) [18].

In general, the locality and the sparsity levels of a clustered representation are tightly related and often go in an opposite way. For instance, one can take the whole graph to be one cluster C. In this case, the sparsity level is good (the degree of C is 0), but the locality level is bad (the radius of C is the radius of the whole graph). In opposite, one can take a representation in which each node forms a cluster. In this case, the locality level is good (the radius of each cluster is 0), but the sparsity level is bad (the degree of a cluster may be Δ where Δ is the maximum degree of the graph).

The complexity of many applications (using clustered representations as a communication structure) is also tightly related to the sparsity and locality levels. In fact, a good locality level implies in general a low time complexity, and a low sparsity level implies low message/memory complexity. All the clustered representations one can find always attempt to find a good compromise between the sparsity and the locality levels.

1.1 Goals and related works

In this paper, we focus on an important clustered representation called *Basic Partition* ([36] Chapter 11). Our interest in this *Basic Partition* comes from its good sparsity-locality compromise. In fact, given an n-node graph, the *Basic Partition* provides a set of *disjoint connected* clusters such that the radius of a cluster is at most $k-1$ and the number of intercluster edges is $O(n^{1+\frac{1}{k}})$ where k is a given integer parameter. Our goal is to design time efficient algorithms for constructing a *Basic Partition* of a graph in a distributed model of computation where *nodes can only communicate with their neighbors by exchanging messages of limited size.*

The *Basic Partition* was first used in [3] in order to design efficient network synchronizers. The idea of producing a clustered representation satisfying a good compromise between the locality level and the sparsity level was then studied in [10]. The results of [10] inspired many other applications and generalizations [17,8,9]. In particular, Awerbuch *et al* [8] studied two important types of clustered representations:

1. The first one called *network decomposition* aims at partitioning the network into *disjoint* colored clusters with either *weak* or *strong* small radius and using a small number of colors. For *weak*-network decompositions, a cluster does not necessarily need to be connected and its radius is computed using paths which may shortcut through neighboring clusters. For *strong*-network decompositions, a cluster must be connected and its radius is computed in the network induced by this cluster.
2. The second one called *network covers* constructs a set of *possibly overlapping* clusters with the property that for any node v, there exists a cluster which contains the t-neighborhood of v, i.e., the neighbors at distance at most t from v where t is an integer parameter. The quality of such covers is measured using the strong radius of clusters and the cluster overlap, i.e., the maximum number of clusters a node belongs to.

In addition to design new network decompositions satisfying some desirable properties, many works studied the problem of distributively constructing these representations in an efficient way. For instance, Awerbuch *et al* [8] gave a deterministic (resp. randomized) distributed algorithm to construct a $(k, t, O(kn^{1/k}))$-neighborhood cover in $O(tk \cdot 2^{c\sqrt{\log n}} + tk^2 \cdot 2^{4\sqrt{\log n}} \cdot n^{1/k})$ (resp. $O(tk^2 \cdot \log^2 n \cdot n^{1/k})$) time for some constant $c > 0$. A (k, t, d)-neighborhood is a set of possibly overlapping clusters such that *(i)* the strong radius of a cluster is $O(kt)$, *(ii)* each node belongs to at most d clusters, and *(iii)* the t-neighborhood of each node is covered by at least one cluster. Moreover, a remark in [8] claims that it is possible to translate this

neighborhood cover into a strong-network decomposition of comparable parameters by using some techniques from [17,10].

On one hand, the strong radius of the cover constructed in [8] is $2k-1$ which is worse (by a factor 2) than the one of the *Basic Partition*. On the other hand, the distributed model considered there does not take into account the congestion created at various bottlenecks in the network (see Section 3.4 of [8]). In fact, the network model used in [8] is the Linial's *free* model [29,30] also known as the $\mathcal{LOCAL}$ model (see [36] Chapter 2). The $\mathcal{LOCAL}$ model assumes that nodes can communicate by exchanging messages of *unlimited size*. This assumption focuses on the locality nature of distributed problems, i.e., what can be computed distributively provided that every node knows its whole neighborhood at some distance?

From a practical point of view, since clustered representations are in the basis of many applications, it is crucial to design fast algorithms to construct such representations in practical distributed models. From a more theoretical point of view, it is also interesting and challenging to design fast algorithms assuming only some weak distributed assumptions, e.g., see [37].

In [34], Moran and Snir gave a distributed algorithm that computes a *Basic Partition* in $O(n)$ time in a distributed model where the size of a message is at most $O(\log n)$ bits, i.e., $\mathcal{CONGEST}$ model (see [36] Chapter 2). The algorithm of [34] improves the previous constructions of [3,41], and allows us to obtain more efficient algorithms for designing network synchronizers γ, γ_1 and γ_2. The algorithm of [34] is semi-sequential: Each cluster is constructed around some node in a distributed and layered fashion. Nevertheless, the clusters are constructed sequentially. In other words, the clusters are constructed one after the other: at each iteration, a new node is selected and the next cluster is constructed.

Moran and Snir end their paper [34] saying:

[34] Question:	Are there *truly parallel algorithms* which construct a Basic Partition in *polylogarithmic or sublinear time complexity* in the $\mathcal{CONGEST}$ model?

1.2 Contribution

In the following, we answer the [34] question. In fact, we give new sparse partition algorithms with $O(n^{1-1/k})$ time complexity, using messages of size at most $O(\log n)$.

More precisely, we give a fully distributed deterministic algorithm Dist_Part with no precomputation step. The idea is to let the clusters grow spontaneously in parallel in different regions of the graph, breaking ties using node identities. We give a detailed implementation of algorithm Dist_Part using small messages and we analyze its efficiency. The time complexity of algorithm Dist_Part is only linear. However, the technique of algorithm Dist_Part is used as a *black box* in order to design a new synchronous deterministic algorithm (Sync_Part) with sublinear time complexity. The main idea to break the linear time barrier is to privilege the construction of clusters in the dense region of the graph which allows us to finish the distributed construction in constant time once the graph becomes sparse. This idea is then adapted in order to run in an asynchronous setting and we obtain algorithm Fast_Part. Our new asynchronous algorithm is even faster than the synchronous one for many particular graphs.

We also give a randomized distributed algorithm (Elect_Part) which is based on a local election technique (LE_k) in balls of radius k. This k-local election technique is a generalization

of the algorithms given in [33] and can be of an independent interest. For general graphs, our randomized construction has the same sublinear time complexity as the deterministic one, but it provides improved bounds for many particular graphs. In fact, the analysis of algorithm Elect_Part enables us to express analytically the degree of parallelism of our construction and to compute the expected number of clusters constructed in parallel.

The basic partition can be applied for designing network covers, network synchronizers and also graph spanners. Hence, we obtain new fast algorithms for all of these applications. For instance, we obtain new $O(n^{1-1/k})$ time deterministic algorithm for constructing optimal spanners with size $O(n^{1+1/k})$ and stretch $2k-1$ which improves on previous constructions. Fast construction of sparse spanners is a real challenge and has been intensively studied over many years and are of special interest for many useful graph structures such as shortest paths, distance oracles and routing [21,23,22,12,11,42,40,15,2]. We should note that the best known distributed algorithms for constructing graph spanners with optimal stretch-size tradeoffs are either randomized [11] or use unbounded size messages [19].

One should finally note that at each round of our distributed constructions the number of messages exchanged by nodes can be order of the number of edges which is rather large. In this paper, we focus only on providing time efficient constructions. Improving the message complexity of our algorithms remains an open research field as it will be pointed later.

Outline In Sections 2 and 3, we give some definitions and we review the Basic_Part algorithm for constructing the *Basic Partition* in a semi-sequential manner. In Section 4, we give a detailed implementation and analysis of the fully distributed algorithm Dist_Part. In Sections 5 and 6, we describe algorithms Sync_Part, Fast_Part and Elect_Part, and we analyze their time complexity. In Section 7, we apply our algorithms to construct sparse graph spanners.

The application of the basic partition to network covers and network synchronizers γ, γ_1 and γ_2 is given in Appendix A. In appendix B, we also give a constructive analysis of our algorithms in the case of *Circulant graphs* and we obtain logarithmic time complexity.

2 Model and definitions

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. We consider the distributed model of computation used in [34,3] and known as the $\mathcal{CONGEST}$ model. More precisely, we assume that a node can only communicate with its neighbors by sending and receiving messages of size $O(\log(n))$ bits. Each node processes messages received from its neighbors, performs local computations, and sends messages to its neighbors in negligible time. In a synchronous network, all nodes have access to a global clock which generates pulses. A message which has been sent in a given pulse arrives before the next pulse. In a synchronous network, the time complexity of an algorithm is defined as the worst-case number of pulses from the start of the algorithm to its termination. In an asynchronous network, there is no global clock and a message delay is arbitrary but finite. In the latter case, the time complexity is defined as the worst-case number of time units from the start of the algorithm to its termination, assuming that a message delay is at most one time unit (this assumption is introduced only for the purpose of performance evaluation).

A cluster C is a subset of V such that the subgraph induced by C is connected. A cluster is always considered with a *leader* node and a BFS spanning tree rooted at the leader. We

also assume that each node v of a graph G has a unique identity Id_v (of $O(\log(n))$ bits). The identity Id_C of a cluster C is defined as the identity of its leader.

For every pair of nodes u and v of a graph G, $d_G(u,v)$ denotes the distance between u and v in G (we also write $d(u,v)$ when G is clear from the context). For any node v of a graph G, $\mathcal{N}(v) = \{u \in V \mid d_G(u,v) \leqslant 1\}$ denotes the neighborhood of v. For any cluster C of a graph G, $\Gamma(C) = \bigcup_{v \in C} \mathcal{N}(v)$ denotes the neighborhood of C. For any cluster C of a graph G, $Rad(C)$ denotes the radius of the cluster C, i.e., the radius of the subgraph induced by C in G. Similarly, for any set $\mathcal{C}$ of clusters, $Rad(\mathcal{C}) = \max_{C \in \mathcal{C}} Rad(C)$ denotes the radius of $\mathcal{C}$.

In all our algorithms, clusters are constructed in a layered and concurrent fashion. In other words, a cluster may grow and explore a new layer but it may also lose its last layer. Some clusters may disappear because they lost all their layers and some others may be newly formed. A cluster is called *finished* if it belongs to the final decomposition that we are constructing. A node belonging to a finished cluster is also called finished. A node is called *active* if it does not belong to a finished cluster.

3 A basic algorithm for constructing a sparse partition

```
1: Set 𝒞 := ∅
2: while V ≠ ∅ do
3:    Select an arbitrary vertex v ∈ V
4:    Set C := {v}
5:    while |Γ(C)| > n^{1/k}|C| do
6:       C := Γ(C)
7:    end while
8:    Set 𝒞 := 𝒞 ∪ C and V := V − C
9: end while
10: return 𝒞
```

Fig. 1. Algorithm Basic_Part [36]

Let $k \geqslant 1$ be an integer parameter. Typically, k is taken to be small compared with n ($k \leqslant \log n$). Let us consider algorithm Basic_Part (Fig. 1) as given in Peleg's book [36] (Chapter 11, page 130). Algorithm Basic_Part was first used in [3] as a data structure for synchronizer γ, then some improvements were given in [41,34]. The algorithm operates in many phases. At each phase, a node is selected from the set of nodes which are not yet covered by a cluster. Then a new cluster is constructed in many iterations according to the sparsity condition of line 5, i.e., $|\Gamma(C)| > n^{1/k}|C|$. It is important to note that the graph G changes in line 8 of the algorithm and the notations in the while loop correspond to the new graph G obtained after the deletion of the corresponding nodes.

Algorithm Basic_Part constructs a *Basic Partition*. In fact, we have the following:

Theorem 1 ([36]). *The output $\mathcal{C}$ of algorithm* Basic_Part *is a partition of G which satisfies the following properties:*

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

Proof. On one hand, once the construction of a cluster C is finished, the nodes of C are definitely removed from the graph G. Thus, the clusters constructed by the algorithm are disjoint. On the other hand, the algorithm terminates once no node remains uncovered. Thus, the final output $\mathcal{C}$ is a partition of G.

Using the sparsity condition, if a cluster C adds i layers, then the size of C satisfies $|C| > n^{i/k}$. Hence, a cluster cannot add more than $k-1$ layers and the first property of the partition holds.

Let $G_{\mathcal{C}}$ be the graph induced by the clusters of the partition $\mathcal{C}$: the nodes of $G_{\mathcal{C}}$ are the clusters of $\mathcal{C}$ and there is an intercluster edge between two clusters if the clusters are at distance 1 from each others. Now, consider a cluster $C \in \mathcal{C}$. Once the construction of C is finished, there are at most $n^{1/k}|C|$ nodes in G at distance 1. Thus, there will be at most $n^{1/k}|C|$ neighboring clusters that will be constructed after C. Thus, there are at most $n^{1/k}|C|$ intercluster edges that can be added to the graph $G_{\mathcal{C}}$ after the construction of C is finished. Thus, the number of intercluster edges is bounded by $\sum_{C\in\mathcal{C}} n^{1/k}|C|$. Since $\mathcal{C}$ is a partition, $\sum_{C\in\mathcal{C}} |C| = n$ and the second property of the partition holds. □

There are many distributed implementations of the Basic_Part algorithm. All of these implementations are semi-sequential. First, they distributively elect a new leader in the network which corresponds to the center of a new cluster. Then, the cluster is constructed in a distributed way by adding the layers in many iterations. The construction of the cluster ends when there are no new layers to add or when the sparsity condition is no longer satisfied. Once the construction of the cluster is finished, a new leader is elected from unprocessed nodes and a new cluster grows up around this leader.

The main difficulty in these algorithms is to *distributively elect* the next leader. In [34], a preprocessing is used to overcome this difficulty. First, a spanning tree T of the graph G is constructed. Then, the next leader is elected by achieving a DFS traversal of T. This technique allows us to improve the complexity bounds of the decomposition: $O(|E|)$ messages and $O(|V|)$ time.

In the next sections, we introduce a new algorithm with no precomputation step and no next leader election step.

4 A deterministic fully distributed basic partition algorithm

4.1 Overview of the algorithm

The main idea of algorithm Dist_Part is to allow clusters to grow in parallel in different regions. In fact, consider two nodes u and v such that $d_G(u,v) \geqslant 2k$ where k is the same parameter as that in algorithm Basic_Part. Then, it is possible to grow two clusters respectively around u and v without any interference. Based on this observation, we initially let each node of the graph be a singleton cluster. Then, we allow the clusters to grow spontaneously. The main difficulty here is to guarantee that the clusters do not share any nodes.

We do not avoid cluster collisions but we try to manage the conflicts that can occur. For instance, consider some region of the graph and suppose that some clusters have independently grown as shown in Fig. 2. The clusters cannot add a new layer simultaneously without overlapping. Thus, we make each cluster compete against its neighbors in order to win a new layer. There are two critical situations. Either, a cluster enters in conflict with an adjacent one

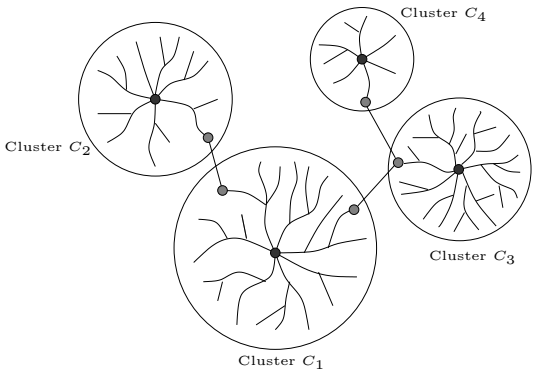


Fig. 2. An example of conflicts between clusters at distance 1 or 2

or with another cluster at distance two. For instance, in the example of Fig. 2, cluster C_1 tries to invade some nodes that belong to cluster C_3 and C_2 at distance 1. Thus, the neighboring cluster C_1, C_2 and C_3 are in conflicts. Similarly, cluster C_4 tries to invade some nodes in cluster C_3. Nevertheless, these nodes are also required for the new layer of cluster C_1. Thus, the two clusters C_1 and C_4 (at distance 2) are also in conflict. To resume, each cluster must compete against all clusters at distance 1 or 2 in order to add a layer. In addition, a layer not satisfying the sparsity condition of algorithm Basic_Part must be rejected.

```
 1: continue := True
 2: while continue do
 3:    execute the Exploration Rule
 4:    if success of the Exploration Rule then
 5:       add the new layer
 6:       execute the Growth Rule
 7:       if Non success of the Growth Rule then
 8:          reject the last explored layer
 9:          switch to a finished cluster
10:          continue := False
11:       end if
12:    else
13:       execute the Battle Rule
14:    end if
15: end while
```

Fig. 3. Algorithm Dist_Part: code for a cluster

In order to manage the conflicts and the cluster growth, we use the following rules:

1. *Exploration Rule*: a cluster is able to add a new layer if its *identifier* is bigger than those of not finished neighboring clusters at distance one or two. If a cluster wins in exploring a new layer then it must apply the *Growth Rule*, otherwise it must apply the *Battle Rule*.
2. *Growth Rule*: If the sparsity condition is satisfied then a cluster adds the last explored layer and tries to apply the *Exploration Rule* once again. Otherwise, the cluster construction is finished and the cluster rejects the last explored layer. The nodes in the rejected layer are re-initialized to singleton clusters with their initial identifiers.

3. *Battle Rule*: a cluster loses its *whole* last layer if at least one neighboring cluster at distance one has successfully applied the *Exploration Rule*. The nodes lost by a cluster are re-initialized to singleton clusters with their initial identifiers.

Based on the three previous rules, we obtain the fully distributed algorithm Dist_Part described in a high level way in Fig. 3.

Remark 1. It is important to choose a unique identifier for each cluster. For instance, the identifier of a cluster can be chosen to be the identity of its leader. This is implicitly assumed in the rest of this section. However, we can also choose the couple $(|C|, Id_v)$ as the identifier of a cluster C with a root v, and the lexicographical order to compare cluster identifiers.

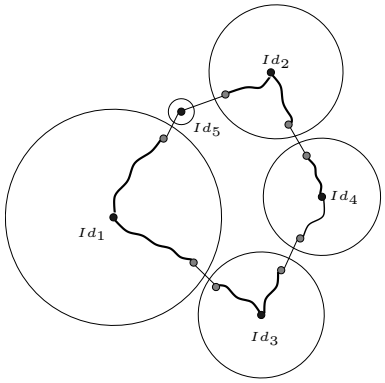
Example: Let us consider the concrete example of Fig. 4. We have five clusters 1, 2, 3, 4 and 5 with identities $Id_1 > Id_2 > Id_3 > Id_4 > Id_5$. Assume that the identifier of each clusters corresponds to the identity of its leader node. When a new exploration begins, cluster 1 wins against clusters 5 and 3. Cluster 2 wins against clusters 4 and 5 but loses against cluster 1 which is at distance two. Thus, cluster 2 cannot add a new layer. Cluster 4 loses against both clusters 2 and 3 but it will not be invaded because both clusters 2 and 3 cannot 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 becomes a singleton cluster with its initial identity Id_6. The node connecting cluster 3 with cluster 1 becomes a leaf in cluster 1. Now, suppose that the sparsity condition for the new enlarged cluster 1 is not satisfied. Then, cluster 1 rejects the last explored layer and its construction is finished. Hence, the nodes in the rejected layer become singleton clusters. Then, the remaining active clusters spontaneously continue new explorations. In our example, both cluster 2 and cluster 3 will succeed their explorations and add a layers. Note that in the other regions of the graph, there are other clusters which are fighting against each others. Hence, many clusters can grow in parallel.

4.2 Detailed description and implementation

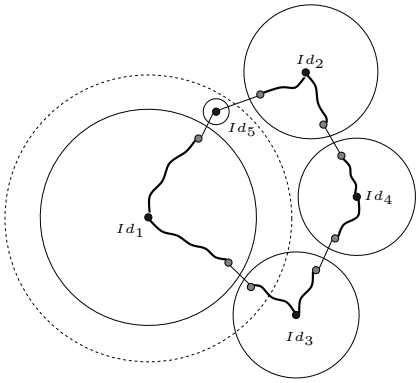
In this section, we give a complete description of how to implement the three rules of the Dist_Part algorithm using message passing. For the clarity of our algorithm, we assume that the identifier of a cluster is the identity of its root. The main difficulty when implementing the three rules of algorithm Dist_Part is to coordinate the center of a cluster with its leaves, i.e., nodes at the border of the cluster. On one hand, the center of a cluster cannot see what is happening on the borders of its cluster. Hence, it must always wait for pieces of information from the leaves before making a decision. Symmetrically, the leaves cannot see the global state of their cluster. Hence, they must also wait for information from the center of their cluster.

To apply the three rules, the nodes in a cluster must collaborate. Each node can be in five states *root*, *leaf*, *relay*, *orphan* or *final*. At the beginning, all nodes are orphans and they form *orphan clusters*, i.e., cluster with only one node. If a node is in a *final* state, then it belongs to a finished cluster and thus it does not make any computation. Roughly speaking, if a node v is in a *root* state, then it is the leader and it makes decisions for its cluster. If v is in a *leaf* state, then it tries to invade new nodes and it informs its root. If v is in a *relay* state, then it forwards information from the leaves to the root.

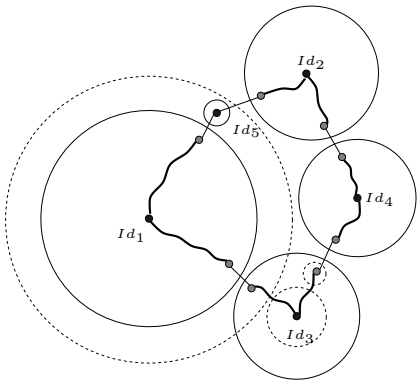
As long as new layers are added (resp., removed) to (resp., from) a cluster, the nodes in the cluster maintain a layered BFS spanning tree. The root of the tree corresponds to the



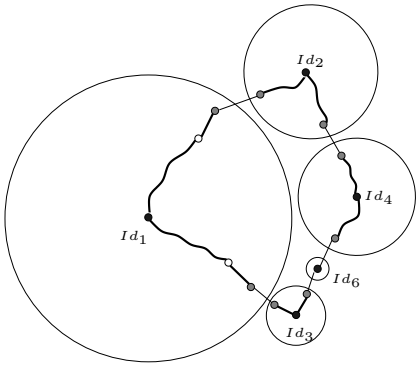
(a) starting configuration.



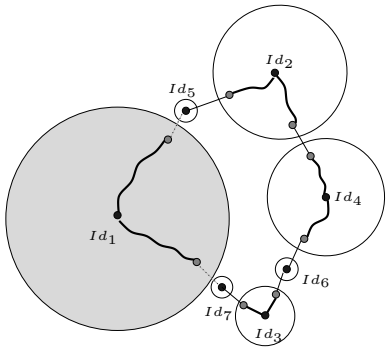
(b) Cluster 1: *Exploration rule* success.



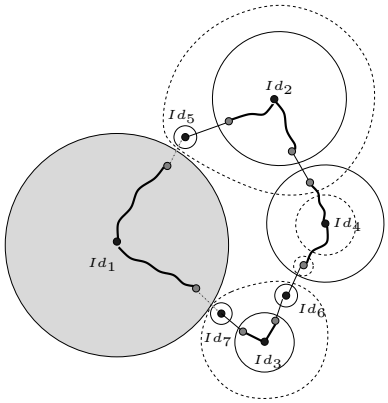
(c) Cluster 3: *Battle rule.*



(d) Cluster 1: *Growth rule*, Cluster 3: *Battle rule.*



(e) Cluster 1: *Growth rule.*



(f) Clusters 3,2: *Exploration rule.* Cluster 4: *Battle rule.*

Fig. 4. An example of algorithm Dist_Part

root of the cluster, the leaves of the tree correspond to the leaves of the cluster and the nodes in the interior of the tree correspond to relay nodes. The decisions of adding or removing a layer are broadcasted by the root node according to the information forwarded by the leaves all along the constructed BFS tree. Each time that some new nodes join a cluster, the BFS spanning tree is enlarged by making each new node choose a parent among the leaves of the already constructed tree.

In next paragraphs, we detail the actions to be performed by each node according to its state. Notice that the state of a node can change several times. For instance, the state of a node can be relay at some time, then becomes leaf, after that orphan, and at last final.

Remark 2. In the pseudo-code of our algorithms, a node uses the function *Send* to send a message to a (or some) neighbor(s). The function *Receive* allows a node to receive a message from a (or some) neighbor(s). The receive function is blocking, that is, a node cannot execute the next instruction in the algorithm unless the receive action is terminated, i.e., all messages were arrived.

Root nodes The algorithm executed by a root node is given in Fig. 5. First, the root verifies if the sparsity condition is satisfied and it informs the leaves (*Growth rule*). More specifically, if the sparsity condition is satisfied, then the root broadcasts a notification message *NEW* to the leaves in order to begin a new exploration. Otherwise, it broadcasts a *REJECT* message saying that the construction is finished.

After broadcasting a *NEW* message, the root waits the response from its leaves. There are three possible cases:

1. The root receives only *STOPPED* messages from its leaves. This means that the leaves did not find new nodes to explore. In this case, the root broadcasts a *STOP* message informing all the nodes that the cluster construction is finished.
2. The root does not receive any *LOST* message, i.e., only *WIN* (or *STOPPED*) messages. This means that the exploration was globally successful. Thus, the root broadcasts a *SUCCESS* message to the leaves. Then, the root waits to learn the size of the new enlarged cluster.
3. The root receives at least one *LOST* message. This means that the exploration was not successful (at least one leaf has lost against a neighboring cluster). Thus, the root informs the leaves by broadcasting a *FAILURE* message. Then, the root waits for the leaves responses. There are two cases:
   - At least one leaf is invaded by another cluster. Thus, the root must receive at least one *BYE* message. In this case, the cluster must reject its last layer (*Battle rule*). Hence, it broadcasts a *DOWN* message asking the leaves to become *orphans*.
   - All leaves have resisted to neighbor's attacks. Thus, the root must receive only *SAFE* messages, i.e., no neighboring cluster has succeeded in invading the current cluster. In this case, the root broadcasts an *OK* message saying that the cluster is not invaded and asking for a new exploration.

Leaf nodes The algorithm executed by a leaf is given in Fig. 6.

Remark 3. We remark that a leaf does not always belong to the last layer of a cluster. For instance, a leaf node may have only final neighbors belonging to finished clusters. Hence, it cannot add new nodes to its cluster. Nevertheless, other leaves belonging to the same cluster can continue exploring new nodes. Therefore, the construction of the cluster can continue even if some leaves cannot locally explore new nodes. In order to handle this situation, for each node we use a local variable h which corresponds to its depth in the BFS-spanning tree of its cluster. If $h = 1$ then the leaf belongs to the last layer and it can compete to add new layers, otherwise the leaf cannot explore any new layer.

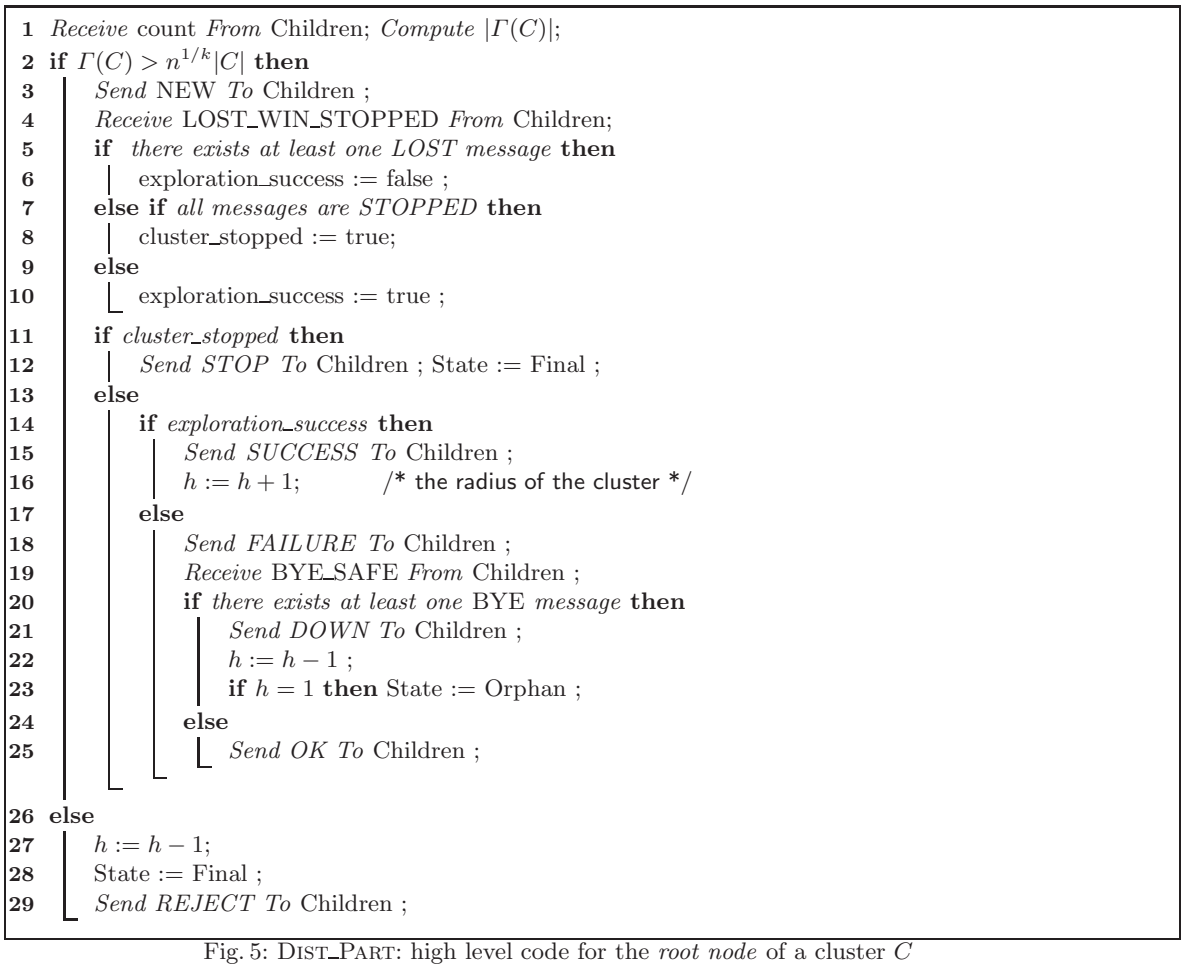


Fig. 5: DIST_PART: high level code for the *root node* of a cluster C

Since the exploration of a new layer is done before verifying the sparsity condition, whenever a node u becomes a leaf in a new cluster, it sends 1 to its parent v in the new cluster and waits for a message from its parent. The parent node v sends back the number of its children and so on. Thus, by a convergecast process, the root will be able to compute the size of the new cluster and to broadcast its decision to the leaves.

If the leaf u receives a *REJECT* message from its parent, then u leaves its new cluster (*Growth rule*) and it becomes an orphan cluster. Otherwise, u receives a *NEW* message from its parent. This means that a new layer must be explored. If u cannot explore new regions, then u sends back a *STOPPED* message. Otherwise, u begins a new exploration using an election technique in a ball of radius two: First, u sends its cluster identifier to its neighbors. Symmetrically, it waits for the identifiers of the neighboring clusters. Second, u computes the maximum of the neighbor identifiers (including the identifier of its own cluster) and sends it again to the neighbors. Symmetrically, it waits for the maximum identifiers sent by neighboring leaves. If all the identifiers received by u are equal to the identifier of u's cluster, then u has locally succeed its exploration and it sends back a *WIN* message. Otherwise, u sends back a *LOST* message.

Remark 4. Since the clusters have unique identifiers, then two neighboring leaves can easily decide whether they belong to the same cluster, e.g., when exchanging their identifiers in a new exploration.

Once the exploration is finished, the leaf node u waits for the decision of its root. There are three cases :

1. If u receives a *STOP* message from the root, then none of the leaves can explore new nodes. Thus, u becomes a *final* node, i.e., the construction of the cluster is finished.
2. If u receives an *SUCCESS* message from the root, then all leaves have succeeded their local explorations, i.e., they won against all neighbors at distance 1 or 2. Thus, u sends a *JOIN* message to neighboring leaves asking them to join its cluster. Then, u switches to a relay state.
3. If u receives a *FAILURE* message from the root, then at least one leaf has not succeeded the exploration. Thus, u sends a *STAY* message to neighboring leaves informing them that they will not be invaded by u's cluster. Then, u waits to know if the neighboring clusters succeeded their explorations. There are two cases:
   - If u receives at least a *JOIN* message from a neighboring leaf (in a different cluster), then it sends back a *BYE* message to its root, waits for an acknowledgment (*DOWN* message) and it joins the new cluster.
   - Otherwise, if none of the neighboring cluster has succeeded in invading the leaf (*STAY* message), the leaf sends back to its root a *SAFE* message. At this stage of the algorithm, the leaves (except those who have received a *JOIN* message) do not know whether their cluster is being invaded or not (only the root globally knows what is happening at its frontiers). Thus, the leaves wait for either an *OK* or a *DOWN* message from the root. If a leaf receives an *OK* message, then it remains in the same cluster and it begins a new exploration once again. Otherwise, it receives a *DOWN* message and it becomes an orphan node.

Orphan nodes An orphan node acts like a root and like a leaf node. In fact, it makes decisions for its singleton cluster and it fights against neighboring nodes. If an orphan node

Fig. 6: DIST_PART: high level code for a *leaf node* in a cluster C

succeeds an exploration, it becomes a root node in a new cluster of radius 1. If it is invaded by a cluster, it becomes a leaf. Otherwise, it re-tries to invade its neighbors (new exploration). If it has only neighbors belonging to finished clusters, then it switches to a final state. The type of messages that must be sent by an orphan node to neighbors can be deduced from the previous discussion.

Relay nodes The main role of a relay node is to forward information from the root to the leaves. If a relay node receives a message from its parent, it simply forwards it to its children. If the message is a *REJECT* 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 a *SUCCESS* message, then the node knows that there is a new layer that will join the cluster. Thus, the depth of the node is incremented by one. If the message is a *DOWN* message then the relay node knows that its cluster was invaded and lost the last layer. In this case, if a relay node belongs to the layer before the last one ($h = 2$) then the relay node becomes a leaf.

On the other hand, if a relay node receives a message from its children, it can deduce which step the leaves are executing (exploration of a new layer: *WIN*, *LOST* or *STOPPED* messages, resistance against neighbors attacks: *OK* or *BYE* messages, and computation of the sparsity condition: integer message). In all cases, the relay node can easily compute what kind of message it must forward to its root.

Remark 5. Each node can easily know which of its neighbors belongs to a finished cluster. It is sufficient to make each node (which becomes final) send a message to its neighbors to inform them. However, we can avoid these extra communication messages as following. When a node v is explored by a cluster C, it uses the Ids sent by neighbors in order to compute a set $\mathcal{F}_C$ of neighbors belonging to the layer before the last one. Then, if v receives a *REJECT* message from the root of C, i.e., the sparsity condition for the last layer of C is not satisfied, then v marks its neighbors in $\mathcal{F}_C$ as finished. Now, consider any edge (u, v). Suppose that u and v do not belong to the same cluster at the end of the algorithm. W.l.o.g., suppose that u becomes in a final state before v. Thus, the cluster containing u must have explored v. Thus, v must have received a *REJECT* message from its parent. Thus, v can decide that u switched to a final state. The case where nodes u and v ends up in the same cluster is trivial since both the two nodes stop communicating.

Remark 6. Although our algorithm is completely asynchronous, we remark that there is a kind of synchronization in our implementation which is close to the one used in synchronizer γ (see Appendix A). On one hand, the root nodes control the execution of the algorithm and give the starting signal to all the actions of the leaves using the relay nodes. Hence, the actions of nodes inside one cluster are synchronized. On the other hand, the decisions made by the roots in neighboring clusters are synchronized since a root must wait for some information concerning those neighboring clusters. The leaves in different clusters synchronize also their actions to execute the decisions of their roots.

4.3 Analysis of the algorithm

Theorem 2. *Algorithm* Dist_Part *terminates.*

Proof. From the algorithm description, the cluster having the biggest identifier in the graph always succeeds the *Exploration rule*. Thus, it always succeeds adding new layers until the sparsity condition is violated. Thus, after at most $k-1$ layers, the nodes in the biggest cluster

are in final states. Now, the remaining cluster with the biggest identifier always succeeds its new explorations and so on until all the nodes are in final states. □

Theorem 3. *Algorithm* Dist_Part *emulates the* Basic_Part *algorithm.*

Proof. From the algorithm description, once the construction of a cluster is finished, the cluster cannot be invaded by any other active cluster. Hence, the constructed clusters are disjoint.

In addition, a new layer is added if and only if it verifies the sparsity condition (*Growth rule*). Symmetrically, if a cluster is invaded, then it loses its whole last layer. Hence, the layers of finished clusters satisfy the sparsity condition.

Thus, the constructed partition satisfies the sparsity and locality properties of algorithm Basic_Part. □

Theorem 4. *In the worst case, the time complexity of algorithm* Dist_Part *is:*

$$Time(\text{Dist\_Part}) = O(n)$$

Proof. In the following proof, we consider the clusters in an increasing order of the time of their construction. Let C be a cluster in the final partition $\mathcal{C}$. Let r be the radius of C. We remark that the construction of a cluster always ends with a sequence of successive successful explorations, and in all these explorations except for the last one the growth condition holds. Consider the first time t when the cluster C starts to successively completes all its explorations. Let $Time(C)$ be the number of time units from t to the end of C's construction. In other words, $Time(C)$ is the duration of the successive successful explorations. Let j be the radius of C at time t. For any $i \in \{j, \cdots, r\}$, we consider the time when C contains i layers, and we denote by r_{max_i} the maximum radius of the neighboring clusters of C.

In order to decide if a layer is added or not, the cluster C must be traversed at most a constant number of times. In addition, before a node joins a new cluster, it informs its previous root and waits for the acknowledgment of this root. Thus, $Time(C) \leqslant \sum_{0<i\leqslant r} O(i + r_{max_i})$. Using Theorem 1, we have $r \leqslant k-1$ and $r_{max_i} \leqslant k-1$. Thus, $Time(C) = O(kr)$.

In the worst case, two clusters are never constructed in parallel. Thus,

$$Time(\text{Dist\_Part}) = \sum_{C\in\mathcal{C}} Time(C)$$

Hence, using the fact that $\sum_{C\in\mathcal{C}} r \leqslant n$, we get

$$Time(\text{Dist\_Part}) = O(k \cdot n)$$

The previous analysis is not sufficient to prove the theorem if the parameter k is not a constant. Nevertheless, it gives us a precious remark. In fact, we remark that it can be interesting to take the couple $(Rad(C), Id_v)$ to be the identifier of a cluster C rooted at a node v and the lexicographical order to compare cluster identifiers (which do not change the overall implementation). In this case, we have $r_{max_i} \leqslant r$. Thus, for the relevant range of $k \leqslant \log(n)$, we have:

$$|C| \geqslant n^{r/k} \Rightarrow r \leqslant \frac{k}{\log(n)} \log(|C|) \Rightarrow r \leqslant \log(|C|)$$

Thus,

$$\begin{aligned} Time(\textsc{Dist\_Part}) &= \sum_{C\in\mathcal{C}} \sum_{0<i\leqslant r} O(i + r_{max_i}) \leqslant \sum_{C\in\mathcal{C}} O(r^2) \\ &\leqslant \sum_{C\in\mathcal{C}} O(\log(|C|)^2) \\ &\leqslant \sum_{C\in\mathcal{C}} O(|C|) \end{aligned}$$

Since $\mathcal{C}$ is a partition, the theorem holds. □

Remark 7. Since at each time unit nodes could exchange order of $|E|$ messages in a worst case scenario, the message complexity of our algorithm can be rather large. However, this does not take into account the fact that finished clusters stop communicating with their neighbors. In this paper, we are mainly interested in the time complexity and we will not consider improving the message complexity measure.

Remark 8. One shall remark that our theoretical analysis is still sequential. In fact, in our analysis we consider that clusters are never constructed in parallel. This can actually happen for instance if the graph contains a path with nodes having a decreasing sequence of identities. In Fig. 7, such a bad scenario is illustrated. In fact, at each round of the algorithm execution there will be only one cluster constructed. At the beginning, all nodes but node number $2n$ fail to explore a new layer. Then, node number $2n$ switches to a final state, and only node number $2n-1$ wins the exploration and so on. Thus, it takes $O(n)$ time to terminate the construction. Also in the example of Fig. 7, there will be order of $O(n^2)$ messages exchanged at each new exploration round. In fact, the nodes of the clique will not be able to switch to final states before the last round. Hence, the message complexity is large. Nevertheless, by considering a random permutation of node identities, one can see that the path in Fig. 7 is likely to be broken in many pieces rather quickly allowing more than only one cluster to grow in parallel. This leads to a better time and message complexity in practice. For the general case of any graph, we think that the average complexity of our algorithm can be much better than the worst case bound of Theorem 4. It would be very nice to prove this claim analytically. This could be a hard task since one have to consider any connected graph with n nodes and all possible permutations of node identities.

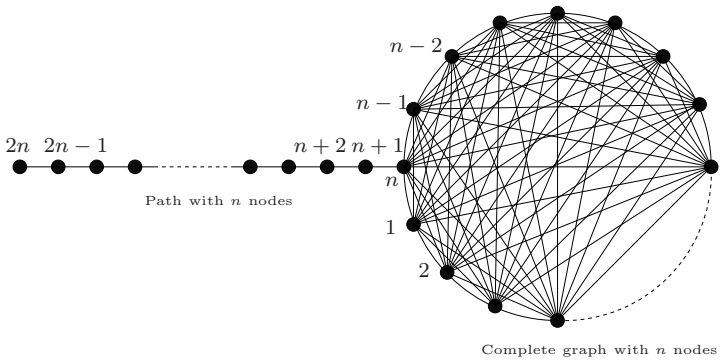


Fig. 7. An example of bad node distribution

5 Sublinear deterministic distributed partition

In the following, we show how to improve algorithm Dist_Part in order to obtain sublinear time algorithms for constructing a basic partition. First, we describe and analyze a new synchronous algorithm called Sync_Part. Then, we show that the synchrony of the network is not important to achieve a sublinear time construction, and we provide a new asynchronous algorithm called Fast_Part.

In the remainder, we denote by V_f the set of finished nodes, i.e., nodes in a finished cluster. Furthermore, we are interested in active nodes in $V - V_f$, hence the degree d_v of a node v is defined as its degree in the graph G_{V-V_f} induced by $V - V_f$.

5.1 A synchronous deterministic algorithm

In this section, we assume that the network is synchronous, i.e., there exists a global clock. At any time t, A_t denotes the set of active nodes (nodes not in V_f at time t), and $R_t = \{v \in A_t \mid d_v > n^{\frac{1}{k}}\}$ denotes the set of active nodes having high enough degrees at time t.

We remark that the sparsity condition for a singleton cluster rooted at some node v is $d_v > n^{\frac{1}{k}}$. Hence, a singleton cluster rooted at some node in $A_t \setminus R_t$ cannot grow any layer. Thus, at any time t, we only let the nodes in R_t compete in order to grow some clusters. Once R_t becomes empty, we just let the remaining active nodes be finished singleton clusters.

The new algorithm Sync_Part works in two stages. The first stage is performed until time $\mathcal{T} = O(k^2 n^{1-1/k})$ is reached. The second stage begins at time $\mathcal{T}$ and lasts $O(1)$ time units.

In the next paragraphs, we give the details of algorithm Sync_Part and discuss its correctness and its complexity.

First stage of the algorithm During this stage, all nodes execute algorithm Dist_Part with the following additional exploration rules:

- *If a node $v \in A_t$ is no longer in R_t, i.e., $v \in A_t \setminus R_t$, then v sets its identity to $-\infty$.*
- *Singleton clusters rooted at nodes in $A_t \setminus R_t$ do not explore any layer.*

Notice that the previous modifications are made only by singleton clusters that do not verify the sparsity condition. We use the same three rules of algorithm Dist_Part to manage the growth of other clusters rooted at any node in R_t.

Let us consider a singleton cluster rooted at $v \in A_t \setminus R_t$. Then, when applying the new rules, v sets its identity to $-\infty$. Hence, v has the lowest identity among all other possible identities. Therefore, node v will not stop the growth of another cluster rooted at a node of R_t. In fact, v can only be a part of other neighboring dense clusters (if it is asked to join). If the neighborhood of v is also in $A_t \setminus R_t$, then the cluster behaves as if it has the lowest identity, i.e., it does not explore any layer. In a practical implementation, a node needs to know whether it is in R_t or not. Since at any moment of algorithm Dist_Part a node is aware of its finished neighbors (see Remark 5), there are no further communications to be done by a node in order to know if it is still in R_t.

Second stage of the algorithm At time $\mathcal{T}$, all remaining active nodes in A_t stop computing and just decide to be finished singleton clusters.

Proposition 1 (Correctness). *Algorithm* Sync_Part *emulates algorithm* Basic_Part.

Lemma 1. *Let C be the cluster with the biggest identity among active nodes at some moment of the algorithm. We need $O(k^2)$ time in the worst case before the construction of C is finished.*

Proof. On one hand, the communications performed by the algorithm are done using a broadcast convergecast process inside the BFS spanning tree of each cluster. Since a cluster has a radius at most $O(k)$, a broadcast (or a convergecast) costs at most $O(k)$ time units.

On the other hand, using the *Exploration rule*, cluster C always wins against its neighboring clusters and it always succeeds in exploring new layers. In the worst case, there will be at most $k-1$ new explored layers. Thus, it takes at most $O(k \cdot k)$ time before the construction of C is finished. □

Lemma 2. *For any time t either $|A_{t+O(k^2)}| < |A_t| - n^{1/k}$ or $R_{t+O(k^2)} = \varnothing$.*

Proof. Consider a given time t. If $R_t = \varnothing$ then $R_{t+O(k^2)} = \varnothing$ (because a finished node will never be active again). In the remainder of the proof we consider the less trivial case where $R_t \neq \varnothing$.

Consider a node v in R_t, i.e., v has more than $n^{1/k}$ active neighbors at time t. Node v must belong to some cluster C. Let u be the root of C. Suppose that u is active at time t, then u must be in R_t (because sparse nodes cannot explore new layers). The cluster having the biggest identity will end up having radius at least 1 and size at least $n^{1/k}$. The vertices of that cluster become inactive and so at least $n^{1/k}$ vertices become inactive. Using Lemma 1, the construction of the cluster having the biggest identity is terminated within at most $O(k^2)$ time. Thus, $|A_{t+O(k^2)}| < |A_t| - n^{1/k}$.

Suppose now that for each node $v \in R_t$, the root of the cluster containing v at time t is inactive. This may happen since it may take some time for a root node to inform the other nodes in its cluster that the construction of the cluster is finished. This information takes at most $O(k)$ time to reach a node v. Thus at time $t' = t + O(k)$, either v becomes inactive or $v \in R_{t'}$ or $v \in A_{t'} \setminus R_{t'}$. Thus, we have two relevant cases:

- No node v becomes in $R_{t'}$. This means that $R_{t'} = \varnothing$ and the second property of the lemma holds.
- At least one node v becomes in $R_{t'}$. By considering the cluster having the biggest identity at time t' and using Lemma 1, we have that $|A_{t'+O(k^2)}| < |A_{t'}| - n^{1/k}$. Since $|A_{t'}| \leqslant |A_t|$ and $t' + O(k^2) = t + O(k) + O(k^2) = t + O(k^2)$, we can conclude that $|A_{t+O(k^2)}| < |A_t| - n^{1/k}$. Thus, the first property of the lemma holds.

  In all cases, either the first property or the second property of the lemma holds. □

Lemma 3. *For $t = O(k^2 n^{1-1/k})$, $R_t = \varnothing$.*

Proof. Using Lemma 2, at the beginning of the algorithm we have $|A_{O(k^2)}| < |A_1| - n^{1/k}$ or $R_{O(k^2)} = \varnothing$. If $R_{O(k^2)} = \varnothing$ then the lemma holds. Otherwise, we have $|A_{O(k^2)}| < |A_1| - n^{1/k}$. Let $t = O(k^2)$. Again by Lemma 2, we have $|A_{t+O(k^2)}| < |A_t| - n^{1/k}$ or $R_{t+O(k^2)} = \varnothing$. If $R_{t+O(k^2)} = \varnothing$ then the lemma holds. Otherwise, we have $|A_{t+O(k^2)}| < |A_t| - n^{1/k}$. Hence it is easy to see that each $O(k^2)$ time period p, either the set of active nodes decreases by at least $n^{1/k}$ factor or all nodes become sparse. Thus, by a simple induction, after at most $n^{1-1/k}$ periods p either set A becomes empty or set R becomes empty. Since at any time t, $R_t \subseteq A_t$, then we can conclude that $R_{O(k^2 n^{1-1/k})} = \varnothing$. □

Since the first stage of the algorithm costs $\mathcal{T} = O(k^2 n^{1-1/k})$ time units and the second one is performed in $O(1)$ time units, we get the following theorem:

Theorem 5 (Time Complexity). *The time complexity of algorithm* Sync_Part *is* $O(k^2\ n^{1-1/k})$.

Remark 9. We remark that since we are interested in small values of k (typically constant values of k), the time complexity of our algorithm is $O(n^{1-1/k})$. However, we can show that for any k verifying $k < \log n$, the value of $\mathcal{T}$ in the previous algorithm can be chosen to be equal to $O(n^{1-1/k})$. To do that, we slightly modify our algorithm by privileging the growth of clusters having the biggest couple ($Radius$,Id). The proof of this claim is technically similar to the proof of Theorem 4. In fact, consider a cluster C rooted at some node v in R_t. Consider the cluster C having the biggest couple (r,Id_v) where r is the radius of C and Id_v the identity of v. Similarly to the analysis of Theorem 4, it takes at most $O(r)$ time to explore a new layer. Moreover cluster C contains at least $n^{r/k}$ nodes that will be removed from set A_t once the construction of C is finished. Let $\ell \geqslant 1$ the radius of C in the final partition. Overall, it costs $O(\ell^2)$ time to construct cluster C. Moreover at least $n^{\ell/k}$ nodes become inactive at the end of the cluster construction. Now, for $k < \log n$, we have that $n^{\ell/k}/\ell^2 = \Omega(n^{1/k})$. Thus, by considering the clusters in an increasing order of their time construction, it is easy to see that the way that our algorithm removes nodes from set A_t is equivalent to removing at least $\Omega(n^{1/k})$ nodes each $O(1)$ time units. Thus, after at most $O(n^{1-1/k})$ time units, either set A_t becomes empty or R_t becomes empty. Thus, after $O(n^{1-1/k})$ time units, no clusters with radius at least 1 can be constructed. Hence, $\mathcal{T}$ can be chosen to be $O(n^{1-1/k})$ and the factor k^2 can be removed in the time complexity.

5.2 An asynchronous deterministic algorithm

Algorithm Sync_Part uses the property that the system is synchronous to find a bound on the time $\mathcal{T}$ before no nodes can grow a non zero radius cluster. The time $\mathcal{T}$ informs all remaining active nodes that there are no more active dense clusters in the graph. This compels us to wait $\mathcal{T}$ time units even if the input graph is sparse. Furthermore, algorithm Sync_Part cannot be run in an asynchronous system without using any synchronizers (see, e.g., Appendix A). In the following, we give a new asynchronous algorithm Fast_Part which does not use any global clock. The general idea of the algorithm is to allow sparse clusters to become finished without waiting until pulse $\mathcal{T}$. Our asynchronous algorithm shows that the key point for speeding up the construction does not rely on the global synchrony of the system, but rather on more local parameters.

Details of the algorithm Let us call a cluster C *dense*, if C has a radius at least 1 or if the single node v of C verifies $d_v > n^{\frac{1}{k}}$. We also define a *sparse cluster* to be a singleton cluster which is not dense (this corresponds to a node in $A_t \setminus R_t$ in algorithm Sync_Part).

Algorithm Fast_Part uses the three rules of algorithm Dist_Part with the following modifications:

- *A dense cluster can explore a new layer if it has an identity bigger than those of its active dense neighbors at distance one or two.*
- *A sparse cluster is not allowed to explore a new layer.*
- *A sparse cluster declares itself finished singleton cluster if:*
  - *all its neighbors are sparse,*

- *or if none of its dense neighbors has succeeded in exploring a new layer.*

Using these new rules, a sparse node is allowed to declare itself finished if it is not explored by any neighboring cluster. This occurs if all neighbors are sparse or if the dense neighbors have not succeeded their explorations. This simple idea enables us to improve the time complexity of the previous synchronous algorithm.

It is obvious that the new modifications can be implemented using messages of size at most $O(\log(n))$ using the same techniques than in algorithm Dist_Part. For instance, we can use a couple *(Id,Dense)* for the cluster identifiers, where *Dense* is a boolean variable indicating whether a cluster is dense or sparse.

Proposition 2 (Correctness). *Algorithm* Fast_Part *emulates algorithm Basic_Part.*

Let Λ be the number of clusters of radius at least 1 at the end of algorithm Fast_Part. Then, the following theorem holds:

Theorem 6 (Time Complexity). *The worst case time complexity of algorithm* Fast_Part *satisfies:*

$$Time(\text{Fast\_Part}) = O(k^2\,\Lambda) = O(k^2\,n^{1-\frac{1}{k}})$$

Proof. The new rules guarantee that a dense cluster is never stopped by a sparse one. In the worst case, no two dense clusters are constructed in parallel. Thus, let us consider the finished dense clusters in a *decreasing* order of their time construction.

The construction of a cluster costs at most $O(k^2)$. Thus, after at most $O(k^2\Lambda)$ time, it only remains active sparse clusters in the graph. In two rounds, all remaining sparse clusters detect that their neighbors are sparse. Thus, using the new rules, they become finished clusters and the algorithm terminates. Thus, the first part of the theorem holds. In addition, since the cluster are disjoint, it is obvious that for any graph and for any execution of the algorithm, Λ is bounded by $n^{1-\frac{1}{k}}$ which completes the proof. □

Remark 10. Note that we can apply Remark 9 for the asynchronous algorithm Fast_Part in order to remove the k^2 factor and obtain a $O(n^{1-\frac{1}{k}})$ time complexity.

Remark 11. The bound $O(k^2\,\Lambda)$ becomes of special interest in the case of graphs where Λ can be shown to be small compared to $n^{1-\frac{1}{k}}$, e.g., see Appendix B for the case study of Circulant graphs.

Remark 12. A nice property of algorithm Fast_Part is to privilege the clustering of dense regions of the graph. For instance, if we consider a graph with only some few dense regions, e.g., some cliques connected by some paths. Our algorithm will automatically capture the topology of the underlying graph and the clustering will have a high priority at those dense regions. In the example of Fig. 7, algorithm Fast_Part constructs a basic partition in constant time whereas algorithm Dist_Part needs $O(n)$ time.

6 Sublinear randomized distributed partition

Although, the previous deterministic algorithms allow us to construct clusters in parallel, their analysis is still sequential. In this section, we give a new randomized algorithm enabling us to compute a lower bound of the number of clusters constructed in parallel.

6.1 Randomized local elections

In [33], a randomized algorithm called L_2-*election* (LE_2 for short) is introduced in order to implement distributed algorithms described with Closed Star (CS for short) relabeling systems. Relabeling systems can be considered as a formal tool to describe and to prove distributed algorithms independently of the underlying model of communication. The reader can refer to [31,32,26,14] for a review on the mathematical foundations of relabeling system. Roughly speaking, a distributed computation step in the CS relabeling system formalism consists in relabeling the nodes attached to a star according to a precise relabeling rule. This encodes the fact that local computations done by a node in a distributed environment can be viewed as a function of the states (labels) of its neighbors. The execution of a distributed algorithm is then described as a sequence of relabeling steps. Each relabeling step changes the labels of a star in the graph according to some rule. The relabeling could be executed in parallel in different regions of the graph at the condition that the corresponding stars do not intersect. In this context, algorithm LE_2 [33] is a message passing algorithm used to implement any formal algorithm described using the relabeling system formalism. Algorithm LE_2 works in fact in rounds where at each round some nodes are elected centers of some stars. In other words, at each round, algorithm LE_2 computes some disjoint stars to be relabeled in parallel.

Algorithm LE_2 is based on the following simple idea. At each round, a node chooses a random number. If the number chosen by a node is bigger than the number chosen by its neighbors at distant 2, then the node is elected center of a star. Thus, the stars centered at the elected nodes can be relabeled in parallel since they are disjoint. Now, suppose that we want to construct the basic partition for $k = 2$. By Theorem 1, the radius of a cluster is at most 1. Thus, we remark that we can use algorithm LE_2 to first compute some disjoint stars. Since the elected stars are disjoint, the elected nodes can verify the sparsity of their corresponding stars in parallel without interfering with each other. Thus, whenever a node is elected center of a star, it computes the size of its corresponding star and then it decides to be either a radius 1 finished cluster or a finished singleton cluster. By repeating this process until each node gets clustered, we obtain the basic partition we want to compute.

In [33], the authors studied the number of nodes locally elected by their LE_2 algorithm, and they interpreted that as the degree of parallelism authorized by their algorithm. Thus, by applying the LE_2 algorithm for constructing the basic partition, we can study the number of clusters constructed in parallel in one round and for $k = 2$. Using that study, we can derive new upper bounds on the time needed to cluster all the nodes.

In the following we will argue that the local election algorithm of [33] can be extended to elect nodes which are centers of disjoint balls of radius $k \geqslant 2$. Our LE_k algorithm is then used as a sub-procedure in algorithm Elect_Part in order to construct the basic partition. These algorithms are described in next paragraphs.

6.2 Algorithm Elect_Part

Algorithm Elect_Part is depicted in Fig. 8 below. It runs in many phases until each node of the graph becomes part of a finished cluster. A phase of the algorithm is executed in two stages.

In the first stage, we construct disjoint balls of radius at most k using algorithm LE_k depicted in Fig. 9. Algorithm LE_k can be viewed as a variant of algorithm Dist_Part. It

works in at most k rounds. At each round, a node applies the exploration rule and tries to add a new layer. If the exploration is not successful, then the node executes the battle rule as in algorithm DIST_PART. Note however that whenever the exploration is successful then the new explored layer is added even if it does not satisfy the sparsity condition. At the end of algorithm LE_k, some nodes will succeed growing a cluster up to some distance $t \leqslant k$. Nevertheless, some layers of those clusters may not verify the sparsity condition.

The second stage allows us to compute finished clusters and to re-initialize the computations for a new phase. In fact, each cluster in the input of the second phase computes independently whether there is a layer that does not satisfy the sparsity condition (Step 2.a). This can be done distributively using convergecast and broadcast between the root and the leaves. If there exists a layer j violating the sparsity condition then the cluster rejects all layers $l \geqslant j$ and declares itself finished (Steps 2.b and 2.c). Otherwise, if all its neighbors are finished then the cluster declares itself finished (Step 2.d). This is because the cluster will not be able to grow any more. Finally, the remaining clusters are broken into singleton clusters in order to run a new phase (Step 2.e).

```
1: while There exist nodes not in a finished cluster do
2:    (0.) each node selects randomly an identity from a big set of integers.
3:    Stage 1: local election in balls of radius k
4:    (1.a) Each node v not in a finished cluster runs algorithm LE_k
5:    Stage 2: reinitialization
6:    (2.a) Each formed cluster C computes independently the sparsity condition for each layer j ⩽ k,
7:    if S contains a layer j violating the sparsity condition then
8:       (2.b) C releases all layers l ⩾ j and becomes a finished cluster,
9:       (2.c) nodes in released layers become singleton clusters.
10:   else
11:      if all neighbors are finished then
12:         (2.d) C becomes finished.
13:      end if
14:   end if
15:   (2.e) Break all non finished clusters and form new singleton clusters.
16: end while
```

Fig. 8. Algorithm ELECT_PART

```
1: Round ← 0;
2: while Round < k do
3:    execute the Exploration Rule;
4:    Round ← Round + 1;
5:    if Non Success of the Exploration Rule then
6:       execute the Battle Rule;
7:    end if
8: end while
```

Fig. 9. Algorithm LE_k: code for a cluster

Remark 13. Algorithm LE_k grows balls of radius k whereas a radius $k-1$ suffices. This allows us to mark edges connecting a cluster with the nodes in the last rejected layer and we avoiding the *preferred edge election step* needed for some applications. This step is discussed in Section 7.

6.3 Analysis of the algorithm

In this section, we compute a bound of the expected number of phases needed before algorithm Elect_Part terminates. The main idea of our analysis is to bound the number of nodes becoming part of a finished cluster in a phase, by using the number of clusters constructed in parallel in each phase.

In the sequel, we say that a node is *locally k-elected* if it succeeds the first stage of algorithm Elect_Part without losing against any other cluster, i.e., line 6 of algorithm LE_k is never executed by a locally k-elected node. We also use a parameter K such that: $\forall v \in V$, $\mathcal{N}_{2k}(v) \leqslant K$, where $\mathcal{N}_{2k}(v) = \{u \in V \mid d(u,v) \leqslant 2k\}$, i.e., K is an upper bound of the $2k$-neighborhood of any node.

It is not difficult to show that the probability that a node v is locally k-elected in a given phase is $\Omega(1/\mathcal{N}_{2k}(v))$. On the other hand, if we denote by X the random variable which counts the number of locally k-elected nodes, then X can be written as the sum of n random variables X_v (for each $v \in V$) such that $X_v = 1$ with probability $q = \Omega(1/\mathcal{N}_{2k}(v))$ and 0 with probability $1-q$. Hence, by the linearity of the expectation, we obtain $\mathbb{E}(X) = \sum_{v \in V - V_f} q$. Thus, the following lemma is straightforward:

Lemma 4. *The expected number of nodes locally k-elected in a phase is lower bounded by* $\Omega(\frac{|V-V_f|}{K})$.

Theorem 7. *Let T be the time complexity of algorithm* Elect_Part. *The expected value of T satisfies:*

$$\mathbb{E}(T) = O\left(k^2 \frac{\log(n)}{\log\left(\frac{K}{K-1}\right)}\right)$$

Proof. Let $i \geqslant 0$ be a phase of the algorithm and $(G_i)_{i \geqslant 0}$ the sequence of graphs such that $G_0 = G$ and for all $i \geqslant 1$, G_i is the graph obtained by removing the nodes (and the corresponding incident edges) belonging to a finished cluster from G_{i-1}. Obviously, G_i is the input graph of phase i.
Let X_i be the random variable which denotes the size of the graph G_i (the number of its nodes) for all $i \geqslant 0$, and let Y_i be the number of nodes locally k-elected in the i^{th} phase. It is clear from Lemma 4 that we have the following inequality:

$$\mathbb{E}(Y_i \mid G_i) \geqslant X(G_i)/K$$

It is also easy to see that $X_{i+1} \leqslant X_i - Y_i$ for all $i \geqslant 0$. Thus,

$$\mathbb{E}(X_{i+1} \mid G_i) \leqslant X_i - \mathbb{E}(Y_i \mid G_i) \leqslant X_i \cdot (1 - \frac{1}{K})$$

For $i \geqslant 0$, we define a new r.v. Z_i by $Z_i = X_i/(1-\frac{1}{K})^i$. Then, $\mathbb{E}(Z_{i+1} \mid G_i) \leqslant Z_i$. Thus, the r.v. Z_i is a super-martingale (see [43]), and then

$$\mathbb{E}(Z_{i+1}) = \mathbb{E}(\mathbb{E}(Z_{i+1} \mid G_i)) \leqslant \mathbb{E}(Z_i)$$

A direct application of a theorem from [43] chapter 9, yields $\mathbb{E}(Z_i) \leqslant Z_0 = n$. Thus

$$\mathbb{E}(X_i) = \left(1-\frac{1}{K}\right)^i \mathbb{E}(Z_i) \leqslant n\left(1-\frac{1}{K}\right)^i.$$

The algorithm terminates when $V_f = V$, i.e., $X_i = 1$. This implies that i is upper bounded by the ratio $\log(n)/\log(\frac{K}{K-1})$. Since both the first and the second stage of the algorithm take at most $O(k^2)$ time to be finished, the assertion in the theorem is proved. □

Remark 14. The bound given by Theorem 7 does not take into account the size of the finished clusters at each phase but only the number of clusters constructed in parallel. Furthermore, the number of clusters constructed in parallel is just lower bounded using the variable K which corresponds to the initial graph G and not to the subgraph in the input of each phase. It would be very interesting to take all this features into account in order to get a better bound on the number of phases needed to terminate algorithm Elect_Part.

6.4 Improvements

In algorithm Elect_Part, sparse nodes also participate in the computations and compete against other nodes in order to grow a ball. This slows down the construction because an elected sparse node will always form a finished singleton cluster. Thus, we can improve algorithm Elect_Part by allowing only dense nodes to compete in order to grow a ball of radius k.

Similarly to algorithm Fast_Part, we let a dense node win against a sparse one using a couple $(Id, Dense)$. In other words, we prohibit that a sparse node stops the growth of a dense cluster. We also let a sparse node declare itself finished if it is not invaded by any neighboring cluster, i.e., if dense neighbors lose their explorations or if all neighbors are sparse.

By considering the number of *dense nodes* at each phase and using the same arguments than in Theorem 7, we can find a bound on the expected number of phases needed to terminate the construction. Unfortunately, the theoretical analysis leads to the same bound than in Theorem 7. It is also easy (using the same reasoning than in Theorem 6) to prove that the complexity of the modified algorithm is bounded by $O(k^2\Lambda)$.

This new modified version of algorithm Elect_Part is particularly interesting because it has a sublinear time complexity for general graphs, and at the same time, it allows us to express the high degree of parallelism of our method. For instance, consider a graph G such that $K = O(n^\epsilon)$ with $\epsilon < 1$. This defines a large class of graphs for which we can achieve an improved time complexity, namely $O(\log(n)n^\epsilon)$.

In Appendix B, we show that the expected running time of the modified algorithm is $O(\log(n))$ in the case of *Circulant graphs*.

7 Application to graph spanners

In this section, we show how to efficiently construct graph spanner in the $\mathcal{CONGEST}$ distributed model using our previous algorithms. A subgraph H is an (α, β)-*spanner* of a graph

G if H is a spanning subgraph of G and $d_H(u,v) \leqslant \alpha \cdot d_G(u,v) + \beta$ for all nodes u, v of G, where $d_X(u,v)$ denotes the distance from u to v in the graph X. The couple (α, β) is called the *stretch* of H, and the *size* of H is the number of its edges.

One immediate application of algorithm Basic_Part is the construction of a $(4k-3, 0)$-spanner with $O(n^{1+1/k})$ edges for any n-node graph G. The spanner is obtained by considering the set of edges spanning each cluster and by selecting an inter-cluster edge for each pair of two neighboring clusters. The bounds on the stretch and the size of the spanner are a straightforward consequence of Theorem 1. In order to construct such a spanner *distributively*, we must first construct the basic partition, and second *select an edge between every two neighboring clusters*. However, we can both avoid this additional step of selecting preferred edges and at the same time improve the bound on the spanner size.

In fact, let us consider any cluster C under construction in algorithm Dist_Part. Before the construction of C is finished (just after the sparsity condition is no longer satisfied) and for every neighboring vertex u of C (u is on the last rejected layer of C), we select an edge from u to some v in the last layer of C and we add it to the spanner S. Moreover, we add the BFS spanning tree of each cluster C to the spanner S. It is well known that this idea allows us to construct a $(2k-1, 0)$-spanner with $O(n^{1/k})$ edges. This idea is in fact attributed to [27] in [36] (Exercise 3, page 188) and is used in [22] as a first step to construct $(1+\epsilon,\beta)$-spanners. The same idea is also used in [34] to improve the complexity of synchronizers γ_1 and γ_2. The time complexity of the algorithms used in [36,34] is $O(n)$ and it has not been improved since.

We remark that the last rejected layer is always explored in all the distributed algorithms described in previous sections. Hence, the edges connecting a cluster with nodes in the last rejected layer are implicitly computed by our algorithms without any extra communications. Hence, by the previous discussion, the following results are straightforward:

Theorem 8. *There is a deterministic distributed algorithm that given a graph with n nodes and a fixed integer $k \geqslant 1$, constructs a $(2k-1, 0)$-spanner with $O(n^{1+1/k})$ edges in $O(n^{1-1/k})$ time using messages of length $O(\log n)$.*

Corollary 1. *There is a deterministic algorithm that given a graph with n nodes constructs a $(3, 0)$-spanner with $O(n^{3/2})$ edges in $O(\sqrt{n})$ time using messages of length $O(\log n)$.*

One can find many papers concerning the construction of graph spanners. In [39,38,44], the reader can find excellent and recent reviews on graph spanners.

To our knowledge Theorem 8 provides the best time complexity for constructing $(2k-1, 0)$-spanners with $O(n^{1+1/k})$ edges in a *deterministic* manner and *using small messages*. The fastest deterministic algorithm was given very recently in [19]. It constructs $(2k-1, 0)$-spanners with $O(kn^{1+1/k})$ edges in $O(k)$ time using messages of polynomial size. Fast *randomized* algorithms using small messages exist. The best one is due to Baswana et al. [12,11]. The authors there gave a (*Las-Vegas*) randomized algorithm that computes a $(2k-1, 0)$-spanner with *expected* size $O(kn^{1+1/k})$ in $O(k)$ time using $O(\log(n))$ size messages. As mentioned in [8], a randomized solution might not be acceptable in some cases, especially for distributed computing applications. In the case of graph spanners, deterministic algorithms that *guarantee* a high quality spanner are more than of a theoretical interest. Indeed, one cannot just run a randomized distributed algorithm several times to guarantee a good spanner, since it is impossible to check efficiently the global quality of the spanner in the distributed model.

8 Open questions

In this paper, we focus on the time complexity of constructing sparse partitions in the practical $\mathcal{CONGEST}$ distributed model. One important motivation of our work is to understand and to study the effects of the congestion created by small messages into the overall time complexity of a distributed algorithm.

We left open the following questions:

1. Can we improve the time complexity of our algorithms from $n^{1-1/k}$ to $n^{1/k}$ in the $\mathcal{CONGEST}$ model? In particular, we remark that, in the case of small k $(2, 3, 4, 5)$, the locality level of the basic partition and the time complexity bound obtained using our technique are better than the bounds one can obtain by both assuming a more powerful distributed model, i.e., unlimited message size, and using techniques from [8]. This observation is intriguing and one can be interested in a lower bound on the time complexity of distributively computing the basic partition. Although, the case $k = 2$ seems hard to improve, we are optimistic that deterministic algorithms with better bounds exist for other values of k.
2. We have studied the sparse partition problem from a *locality* point of view. In other words, we only consider the problem of improving the time complexity. Can we improve the message complexity of our algorithms while maintaining the same time complexity?

References

1. Y. Afek and M. Ricklin. Sparser : a paradigm for running distributed algorithms. *Journal of Algorithms*, 14:316–328, 1993.
2. I. Althofer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9:81–100, 1993.
3. B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32:804–823, 1985.
4. B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Poltkin. Network decomposition and locality in distributed computation. 30^{th} *IEEE Symposium on Foundation of Computer Science (FOCS89)*, pages 364–369, 1989.
5. B. Awerbuch and D. Peleg. Routing with polynomial communication-space trade-off. *SIAM Journal on Discrete Mathematics*, 5:151–162, 1992.
6. B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 42:1021–1058, 1995.
7. B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. 31^{st} *IEEE Symposium on Foundations of Computer Science (FOCS90)*, 2:503–513, October 1990.
8. Baruch Awerbuch, Bonnie Berger, Lenore J. Cowen, and David Peleg. Fast distributed network decompositions and covers. *Journal of Parallel and Distributed Computing*, 39:105–114, 1996.
9. Baruch Awerbuch, Bonnie Berger, Lenore J. Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, February 1998.
10. Baruch Awerbuch and David Peleg. Sparse partitions. In 31^{th} *Symposium on Foundations of Computer Science (FOCS90)*, pages 503–513. IEEE Computer Society Press, October 1990.
11. Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. New constructions of (α, β)-spanners and purely additive spanners. In 16^{th} *Symposium on Discrete Algorithms (SODA05)*, pages 672–681. ACM-SIAM, January 2005.
12. Surender Baswana and Sandeep Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(n^{1+1/k})$ size in weighted graphs. In 30^{th} *International Colloquium on Automata, Languages and Programming (ICALP03)*, volume 2719 of Lecture Notes in Computer Science, pages 384–396. Springer, July 2003.
13. 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.
14. J. Chalopin and Y. Métivier. A bridge between the asynchronous message passing model and local computations in graphs. In *Mathematical Foundations of Computer Science (MFCS05)*, volume 3618 of *Lecture Notes in Computer Science*, pages 212–223. Springer-Verlag, aug 2005.

15. Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28(1):210–236, 1998.
16. Francesc Comellas, Javier Ozón, and Joseph G. Peters. Deterministic small-world communication networks. *Inf. Process. Lett.*, 76(1-2):83–90, 2000.
17. Lenore J. Cowen. *On Local Representations of Graphs and Networks*. Ph. D Thesis, MIT, 1993.
18. B. Derbel and M. Mosbah. A fully distributed linear time algorithm for cluster network decomposition. 16^{th} *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS04)*, pages 548–553, 2004.
19. Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In 27^{th} *Symposium on Principle Of Distributed Computing (PODC)*, pages 273–282, 2008.
20. Bilel Derbel, Mohamed Mosbah, and Akka Zemmari. Fast distributed graph partition and application. In 20^{th} *IEEE International Parallel & Distributed Processing Symposium (IPDPS06)*. IEEE Computer Society Press, April 2006.
21. Michael Elkin. Computing almost shortest paths. In 20^{th} *ACM Symposium on Principles of Distributed Computing (PODC01)*, pages 53–62. ACM Press, 2001.
22. Michael Elkin and David Peleg. $(1+\epsilon, \beta)$-spanner constructions for general graphs. *SIAM Journal on Computing*, 33(3):608–631, 2004.
23. Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1+\epsilon, \beta)$-spanners in the distributed and streaming models. In 23^{rd} *ACM Symposium on Principles of Distributed Computing (PODC04)*, pages 160–168. ACM Press, July 2004.
24. I. Gaber and Y. Mansour. Centralized broadcast in multihop radio networks. *Journal of Algorithms*, 46:1–20, 2003.
25. J.A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27:302–316, February 1998.
26. E. Godard, Y. Métivier, and A. Muscholl. Characterizations of classes of graphs recognizable by local computations. *Theory of Computing Systems*, 37:2:249–293, 2004.
27. S. Halperin and U. Zwick. Unpublished result. 1996.
28. Shay Kutten and David Peleg. Fast distributed construction of small k-dominating sets and applications. *Journal of Algorithms*, 28(1):40–66, 1998.
29. Nathan Linial. Distributive graph algorithms - Global solutions from local data. In 28^{th} *IEEE Symposium on Foundations of Computer Science (FOCS87)*, pages 331–335. IEEE Computer Society Press, October 1987.
30. Nathan Linial. Locality in distributed graphs algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.
31. I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Mathematical Systems Theory*, 28:41–65, 1995.
32. I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.
33. Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Information Processing Letters*, 82:313–120, 2002.
34. Shlomo Moran and Sagi Snir. Simple and efficient network decomposition and synchronization. *Theoretical Computer Science*, 243(1-2):217–241, 2000.
35. A. Panconesi and A. Srinivasan. Improved distributed algorithms for coloring and network decomposition. 24^{th} *ACM Symposium on Theory of Computing (STOC92)*, pages 581–592, 1992.
36. David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications, 2000.
37. David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM Journal on Computing*, 30(5):1427–1442, 2000.
38. Seth Pettie. Low distortion spanners. In 34^{th} *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 78–89, July 2007.
39. Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. In 27^{th} *Symposium on Principle Of Distributed Computing (PODC)*, pages 253–262, 2008.
40. Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In 32^{nd} *International Colloquium on Automata, Languages and Programming (ICALP)*, volume Lecture Notes in Computer Science, 2005.
41. L. Shabtay and A. Segall. Low complexity network synchronization. 8^{th} *Internatinal Workshop on Distributed Algorithms*, pages 223–237, 1994.

42. Mikkel Thorup and Uri Zwick. Spanners and emulators with sublinear distance errors. In 17^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 802–809, 2006.
43. D. Williams. *Probability with Martingals*. Cambridge University Press, 1993.
44. David P. Woodruff. Lower bounds for additive spanners, emulators, and more. In 47^{th} *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 389–398, October 2006.

A Application to Neighborhood Covers and Network Synchronizers

Neighborhood covers can be thought as a generalization of the basic partition. In fact, for any positive integer ρ, a ρ-neighborhood cover of a graph G can be defined as a collection of clusters $\mathcal{C} = \cup C$ such that for every $v \in V$, there exists a cluster $C \in \mathcal{C}$ such that $\mathcal{N}_\rho(v) \subseteq C$ where $\mathcal{N}_\rho(v) = \{u \in V \mid d(u,v) \leqslant \rho\}$ denotes the ρ-neighborhood of node v in the graph G. Hence, the basic partition described before is a 0-neighborhood cover of G having a low average degree, namely $O(n^{1/k})$. In Peleg's book [36] (Chapter 12), one can find a survey on different types of covers obtained on the basis on the basic partition algorithm. In particular, given an initial cover $\mathcal{S}$, it is shown in [36] how to extend algorithm Basic_Part to construct a *coarsening* cover $\mathcal{T}$ of $\mathcal{S}$, that is a cover that subsumed $\mathcal{S}$. By taking $\mathcal{S} = \cup_{v \in V} \mathcal{N}_\rho(v)$, it can be shown that a ρ-neighborhood cover with radius $O(k \cdot \rho)$ and average degree $O(n^{1/k})$ can be constructed on the basis of algorithm Basic_Part (the proof is by Theorem 12.2.1 of [36]). This type of neighborhood covers is also used in [34] as an auxiliary communication structure to design network synchronizers. More specifically, the authors in [34] described a distributed 1-neighborhood cover algorithm that is used to design a new efficient synchronizer. Based on the extended construction of [36], it is not difficult to extend our basic partition distributed algorithms to construct ρ-neighborhood covers distributively. Since we are mainly interested in the application of covers in the design of network synchronizers, we will briefly outline the modifications to be done to obtain the ρ-neighborhood cover used in [34]. Extending our technique for any ρ is left as an exercise.

A.1 Distributed construction of 1-neighborhood covers

In this section, we extend algorithm Dist_Part in order to cover the 1-neighborhood of each node. We use the same distributed techniques to manage cluster growth. However, we make a cluster explore two layers at the same time instead of only one. At each new exploration, each cluster fights to maintain two layers l_i and l_{i+1} with i the radius of the cluster. The first layer l_i allows the cluster to compute the sparsity condition (the same one than in algorithm Dist_Part). The second layer l_{i+1} (which is the last explored layer) guarantees that the neighborhoods of all nodes in layer l_i are in the current cluster. There are mainly five important modifications to do:

1. At the beginning of the algorithm, all nodes are orphans. An orphan node first explores *two consecutive* layers before starting computing the sparsity condition.
2. If the sparsity condition is satisfied for layer l_i, then a cluster begins a new exploration, i.e., the leaves in layer l_{i+1} try to invade new nodes. If the new exploration succeeds, then layer l_{i+1} becomes layer $l_{i'=i+1}$ and the new explored layer becomes the new $l_{i'+1}$ layer.
3. If the sparsity condition for 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}. Nevertheless, only nodes in layers $l_{j<i}$ are in a final state. The 1-neighborhoods of all nodes in layer l_i are covered by the finished cluster but they do not stop computing yet. In fact, the 1-neighborhoods of nodes in layer l_{i+1} may not be covered by a cluster. Hence, nodes in layer l_i become orphan clusters with identity $-\infty$ in order to allow other clusters to grow and cover the neighborhoods of nodes in layer l_{i+1}. On the other side, nodes in layer l_{i+1} become orphan clusters with their initial identifiers and continue competing in order to grow new clusters.

4. If a new exploration fails, i.e., there is a cluster at distance 1 or 2 (from layer l_{i+1}) with a bigger identifier, then:
   - either the winner lost against another neighboring cluster and the current cluster is not invaded. Hence, the cluster simply retries a new exploration.
   - or the current cluster is invaded and the cluster loses its last layer l_{i+1}. Hence, 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 identifiers. Layer l_i becomes the last layer $l_{i'+1=i}$ and layer l_{i-1} becomes layer $l_{i'=i-1}$. Then the cluster begins a new exploration once again.
5. When the construction of a cluster is finished, nodes at distance at least 2 from the border of the cluster, i.e., layers $l_{j\leqslant i-1}$, switch to final states. In fact, layer l_{i-1} of a finished cluster acts as a barrier that protects the finished cluster from future invasions. Layers $l_{j\leqslant i-1}$ are usually called the *Kernel* of the cluster.

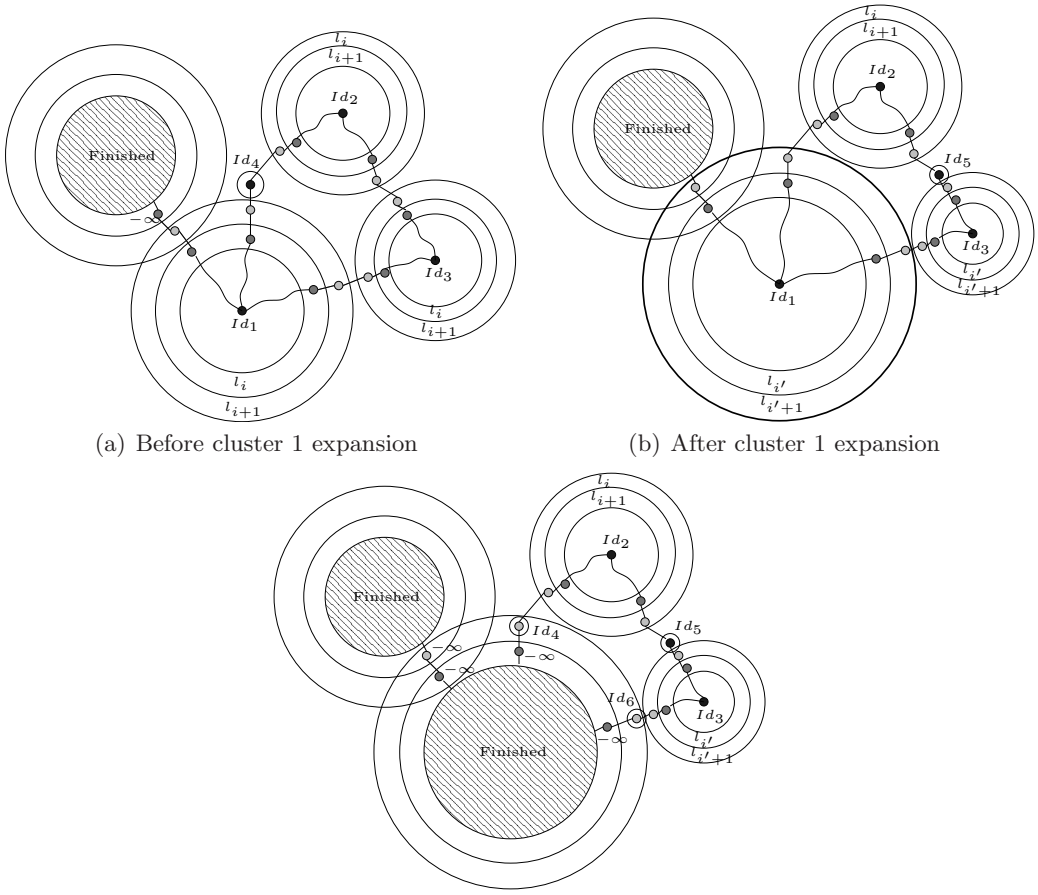
It is easy to see that the time and message complexity of the extended algorithm increases by only a constant factor due to the computation of the extra layer l_{i+1}. Notice also that it is not difficult to adapt the techniques of algorithms Fast_Part and Elect_Part in order to obtain sublinear time complexity.

An example of cluster growth In Fig. 10, we give an example of how the cover is constructed. In our example, there are four active clusters: 1, 2, 3 and 4 with identities $Id_1 > Id_2 > Id_3 > Id_4$. We suppose that there is a finished cluster in the neighborhood of cluster 1.

The nodes in layer l_i of the finished cluster (first part of Fig. 10) still participate in the computation with identity $-\infty$, all the nodes in the Kernel of the finished cluster are in a final state. There is also a node in layer l_{i+1} of the finished cluster which belongs to layer l_{i+1} of cluster 1.

Suppose that the layers l_i of the active clusters satisfy the sparsity condition, then these clusters will try to grow. Cluster 2 cannot grow because cluster 1 is at distance two of it and has a bigger identity. 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 (second part of Fig. 10). Note also that the node with identity $-\infty$ in the finished cluster is invaded by cluster 1. This guarantees that the neighborhood of the children of the $(-\infty)$-node in the finished cluster is covered by cluster 1.

Once the new exploration is finished, cluster 1 verifies the sparsity condition. If it is satisfied, a new exploration will begin and clusters 2 and 3 will be invaded. Note that nodes in the Kernel of the finished cluster will not be invaded by cluster 1. If the sparsity condition is not satisfied which is the case in the third part of Fig. 10, the construction of cluster 1 is finished. The nodes in layer $l_{i'}$ become orphans with identity $-\infty$. The nodes in layer $l_{i'+1}$ become orphan nodes except those which are already in layer l_i of another finished cluster (those whose neighborhoods are covered). Note that the two finished clusters we have constructed overlap (they have a common edge).



(a) Before cluster 1 expansion

(b) After cluster 1 expansion

(c) After construction of cluster 1 is finished

Fig. 10. An example of a cluster expansion for cover needed for γ_2

A.2 Application to network synchronizers

The basic partition and the 1-neighborhood covers constructed in previous sections are of special interest for designing network synchronizers γ, γ_1 and γ_2 [34]. In the following, we review the basic properties of these synchronizers.

Background Network synchronizers allow us to transform a synchronous algorithm into an asynchronous one. In general, one prefers to design a distributed algorithm in a synchronous model rather than an asynchronous model which is typically harder to grasp and to analyze ([36], Chapter 6). From a practical point of view, network synchronizers provide a uniform methodology for transforming synchronous distributed algorithms into asynchronous ones.

Generally speaking, the basic idea of network synchronizers is to simulate a global clock by using *local pulse generators*. If the local clock pulse of some node v is equal to p, then node v knows that the messages that it has sent at pulse $p-1$ have reached their destinations. Many simulation techniques were developed in order to guarantee this property:

1. The first basic technique is known as synchronizer α. The general idea of synchronizer α is to send an acknowledgment corresponding to each received message of the original syn-

chronous algorithm. Once, a node receives an acknowledgment of all the original messages corresponding to one pulse, the node informs its neighbors and then it generates the next pulse. This technique leads to an overhead of $O(|E|)$ messages in order to simulate a pulse. Assuming that a message delay is at most $O(1)$ (this is only for performance analysis), it also leads to the theoretical $O(1)$ time overhead per pulse.

2. The second basic technique called synchronizer β assumes a precomputed rooted BFS spanning tree T of G. Only the root of T have a pulse generator which controls all other nodes. In fact, once a node u receives the acknowledgments of the messages it has sent, the node u is ready for the next pulse and it informs its parent in the tree T. The parents forward this information until it reaches the root of T. Once the root learns that all the nodes are ready for the next pulse, it broadcasts a message saying *"it is time for the next pulse"*! Thus, synchronizer β implies an overhead of $O(|V|)$ messages and $O(D)$ time per pulse, where D is the diameter of G.
3. The third technique is an intermediate technique which provides a good time-message trade-offs. This technique implies three synchronizers γ, γ_1 and γ_2 ([34]). All of these three synchronizers use sparse covers. More precisely, synchronizer γ uses the basic partition as an auxiliary communication structure. Synchronizer γ_1 uses a cover based on the basic partition where each edge belongs to at least one cluster. This property is easily obtained by our partition algorithms by simply marking the last rejected layer as part of the cluster. Finally, synchronizer γ_2 uses the the 1-neighborhood cover described in the previous section.

A detailed description of synchronizers γ, γ_1 and γ_2 can be found in [34]. In the following, we just outline the basic ideas used in synchronizer γ (the two other synchronizers are based on the same general ideas). First, we assume the following:

1. A partition $\mathcal{C}$ of G is constructed.
2. A rooted BFS spanning tree T_C for each cluster $C \in \mathcal{C}$ is constructed.
3. A set $\mathcal{I}$ of intercluster edges is selected.

To simulate a pulse, we combine the techniques of synchronizers α and β. Roughly speaking, the root of each tree T_C first waits to learn that the nodes in C are ready for the next pulse (which costs $O(|C|)$ messages and $O(Rad(C))$ time for each cluster). Then, the cluster tries to synchronize with its neighbors using the intercluster edges. More precisely, the root of C broadcasts a notification message all along the tree T_C saying that all the nodes in its cluster are ready. When the leaves of T_C receive the notification message, they forward it to neighboring clusters using the selected intercluster edges (which costs $O(|\mathcal{I}|)$ message and $O(1)$ time). Symmetrically, the leaves receive a notification from their neighboring clusters. When receiving such a notification, they send it back to their root. Once the root receives the notification messages of its neighbors, it sends a message to the nodes in its cluster saying "it is time for the next pulse". Thus, the global overhead is $O(n + |\mathcal{I}|)$ messages and $O(Rad(\mathcal{C}))$ time per pulse.

Thus, if we take the basic partition as a communication structure, then the global overhead is $O(n^{1+1/k})$ messages and $O(k)$ time per pulse which gives a good compromise compared to synchronizer α and β.

Contribution The previous overhead is essentially optimal according to Lemma 25.1.7 in Peleg's book [36]. Synchronizers γ, γ_1 and γ_2 have the same performances up to a constant

factor. Hence, one remaining challenge is to improve the pre-processing step of constructing the required covers. Using our algorithms, the time complexity of this pre-processing step is reduced from $O(n)$ in previous implementations to $O(n^{1-1/k})$.

B Case Study: Circulant Graphs

In this section, we study the efficiency of algorithms FAST_PART and algorithm ELECT_PART in the case of *Circulant Graphs*. In fact, Circulant Graphs are *dense* enough to be interesting for the algorithm we are studying. They have enough large diameter in order to let the analysis non trivial and constructive. In addition, the analysis given here is interesting from a theoretical point of view. In particular, the proof of Theorem 12 below illustrate the improvements discussed in Section 6.4. The reader should also note that this class of graphs was studied in many past works and for different purposes. For instance, it is used in [16] as a basis for the construction of graphs having the small-world property.

Definition 1. *A circulant graph $\mathcal{C}ir_n(\mathcal{L})$ is a graph of n nodes $\{1, 2, ..., n\}$ in which a vertex i is adjacent to nodes $(i - j)$ and $(i + j)$ for each i and j in the list $\mathcal{L}$ (see Fig. 11 for an example).*

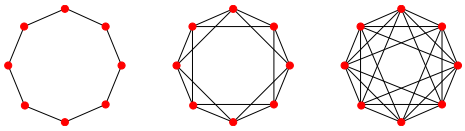


Fig. 11. An Example of $\mathcal{C}ir_n(\mathcal{L})$ graphs with n=8 and $\mathcal{L} \in \{\{1\}, \{1, 2\}, \{1, 2, 3\}\}$

Definition 2. *For every parameter ϵ such that $0 < \epsilon \leqslant 1$, we define the graph $\mathcal{C}ir_n^\epsilon$ to be the circulant graph $\mathcal{C}ir_n(1, 2, ..., \lfloor \frac{n^\epsilon}{2} \rfloor)$.*

In the sequel, we suppose that $\epsilon > \frac{1}{k}$. In fact, if $\epsilon \leqslant \frac{1}{k}$ then the graph is already sparse and all our algorithms terminate in $O(1)$ rounds.

Theorem 9. *For $k < \log(n)$ and for every graph $\mathcal{C}ir_n^\epsilon$, the time complexity of algorithm* FAST_PART *is bounded by $O(n^{1-\epsilon})$.*

Proof. For $k < \log(n)$, any constructed cluster has radius at most 1. It can also be shown that $\Lambda \leqslant 2\,\frac{n}{n^\epsilon} = O(n^{1-\epsilon})$. Thus, the theorem follows as a consequence of Theorem 6. □

Theorem 10. *Let T be the time complexity of algorithm* ELECT_PART. *Then, for every graph $\mathcal{C}ir_n^\epsilon$, the expected value of T satisfies:*

$$\mathbb{E}(T) = O(k^3 \log(n)\, n^\epsilon)$$

Proof. For any graph $\mathcal{C}ir_n^\epsilon$, it is easy to show that $K = 2k\, n^\epsilon$ (we recall that K is an upper bound of the $2k$-neighborhood of any node). Thus, $\log(1 - \frac{1}{K}) \leqslant -\frac{1}{K} = -\frac{1}{2kn^\epsilon}$ and the result follows immediately from Theorem 7. □

The two theorems 9 and 10 are immediate consequences of the analysis we have already made for algorithms Sync_Part and Elect_Part. In particular, we obtain a time complexity which is better than $O(n^{1-\frac{1}{k}})$. Nevertheless, using a more careful analysis, we obtain the following bounds:

Theorem 11. *For every graph $\mathcal{C}ir_n^\epsilon$, the expected time complexity T of algorithm* Elect_Part *satisfies:*

$$\mathbb{E}(T) = O\left(k^3 \log(n) + kn^{\frac{1}{k}}\right)$$

Theorem 12. *Using the improved version of algorithm* Elect_Part *described in Section 6.4, the expected time complexity T of algorithm* Elect_Part *satisfies:*

$$\mathbb{E}(T) = O(k^3 \log(n))$$

Proof. We prove the previous two theorems in two parts. The first part is common to the two theorems. The technical arguments are similar to those in the analysis made in Theorem 7 but the reasoning is different.

First Part of the proof Let $i \geqslant 0$ be a phase of algorithm Elect_Part and $(G_i)_{i\geqslant 0}$ be the sequence of graphs such that $G_0 = G$ and for all $i \geqslant 1$, G_i is the graph obtained by removing the nodes belonging to a finished cluster from G_{i-1}.
Let V_i be the set of nodes having a degree higher than $\lfloor \frac{n^\epsilon}{2} \rfloor$ in phase i. Let X_i be the random variable which denotes the number of nodes in V_i, and let Y_i be the number of nodes from V_i which are locally k-elected in the i^{th} step. The following inequality holds:

$$\mathbb{E}\left(Y_i \mid G_i\right) \geqslant \frac{X_i}{K},$$

One can show that if the node v belongs to V_i, then every active neighbor w of v is also in V_i. Hence, we can state the following:

$$\begin{aligned}
\mathbb{E}\left(X_{i+1} \mid G_i\right) &\leqslant X_i - \mathbb{E}\left(Y_i \mid G_i\right) \tfrac{n^\epsilon}{2} \\
&\leqslant X_i(1 - \tfrac{n^\epsilon}{2K}) \\
&\leqslant X_i(1 - \tfrac{1}{2 \cdot 2k}).
\end{aligned}$$

By induction and using the same arguments as in Theorem 7, the expected time such that $X_i = 1$ is bounded by:

$$O(k^2 \frac{\log(n)}{\log(\frac{4k}{4k-1})})$$

Let us consider the time after which all nodes in the graph have a degree less than $\lfloor \frac{n^\epsilon}{2} \rfloor$, i.e., the time such that $V_i = 0$) . One can show that the remaining nodes are grouped in many connected fragments that can be divided in two types: dense components with more than $n^{\frac{1}{k}}$ nodes and sparse components with no more than $n^{\frac{1}{k}}$ nodes. All theses components are disjoint and do not share any node. Thus, the algorithm runs independently on each component.

Let us consider a dense component C_d, i.e., $n^{\frac{1}{k}} < |C_d| < \lfloor \frac{n^\epsilon}{2} \rfloor$). In one phase, there will be exactly one elected node in C_d and the finished cluster constructed around this node will contain the whole component C_d. Thus, in $O(k)$ time, all nodes in C_d become finished.

Second part of the proof of *Theorem 11*: Let us consider a sparse component C_s, i.e., $|C_s| \leqslant n^{\frac{1}{k}}$). The nodes of such a component have a degree less than $n^{\frac{1}{k}}$. At each phase of algorithm ELECT_PART, there will be exactly one elected node in C_s which forms a finished singleton cluster. Thus, we need at most $O(n^{\frac{1}{k}})$ phases of $O(k)$ time units each before all nodes in C_s become finished.

To conclude, if V_i becomes empty then we need at most $O(kn^{\frac{1}{k}})$ time units before the algorithm terminates and Theorem 11 holds.

Second Part of the proof of *Theorem 12*: Let us consider a sparse component C_s, i.e., $|C_s| \leqslant n^{\frac{1}{k}}$). The nodes of such a component have a degree less then $n^{\frac{1}{k}}$. Thus, using from the improvements of algorithm ELECT_PART in Section 6.4, these nodes are allowed to be finished. Hence, in $O(1)$ time, they all become finished singleton clusters.

To conclude, if V_i becomes empty then we need at most $O(k)$ time units before the algorithm terminates and Theorem 12 holds. □