Fast Distributed Graph Partition and Application (Extended Abstract)

Bilel Derbel, Mohamed Mosbah, and Akka Zemmari

LaBRI, Université Bordeaux I
351, Cours de la libération
33405 Talence, France
{derbel, mosbah, zemmari}@labri.fr

Abstract

This paper presents efficient deterministic and randomized distributed algorithms for decomposing a graph with n nodes into a disjoint set of connected clusters with small radius and few intercluster edges. Our algorithms can be easily implemented in the distributed $\mathcal{CONGEST}$ model of computation i.e., limited message size, improving the time complexity of previous algorithms [27, 3, 29] from linear to sublinear. One important application of our algorithms is efficient construction of sparse graph spanners. In fact, given a parameter k, we show that there exists a sublinear deterministic distributed algorithm that constructs a graph spanner of stretch $2k-1$ with at most $\mathcal{O}(n^{1+1/k})$ edges in the $\mathcal{CONGEST}$ model.

Keywords: *distributed algorithm, distributed model, time complexity, sparse partition, graph spanner.*

1. Introduction

1.1. Goals

The purpose of decomposing a graph into *subsets of nodes*, called *clusters*, is to produce a data structure which uses a small amount of information to represent the graph in a relevant and accurate way. Many examples in the literature show that graph decomposition is at the bottleneck of many algorithms that *efficiently* solve distributed fundamental problems including synchronization [27, 32], maximal independent set (MIS) [6, 20, 21], coloring [6, 28], routing [8], spanners [14, 18], shortest paths [33, 12], mobile users [9]. Often, one tries to have clusters of small radius in such a way that the induced graph, i.e., the graph obtained by contracting each cluster into a single node, has few edges. An important graph decomposition, called *Basic Partition* ([29] Chapter 11), aims at partitioning an n-node graph into *disjoint connected* clusters such that the radius of a cluster is at most $k-1$ and the number of the intercluster edges is $\mathcal{O}(n^{1+\frac{1}{k}})$ where k is a given parameter. In this paper, we are interested in designing 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.*

1.2. Related work and motivation

The *Basic Partition* that interests us, was first used in [3] in order to design efficient network synchronizers. The idea of producing a decomposition satisfying a good compromise between the locality level of the decomposition (e.g., the radius of clusters) and the sparsity level (e.g., the number of edges connecting the clusters) was then studied in [7]. The results there have inspired many other works [15, 4, 5]. In particular, [4] surveys the different formulations of network decompositions and covers. One can find two main types of such network representations. The first one 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 need necessarily 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. The second type of clustered-based representations called network covers constructs a set of *possibly over-*

1-4244-0054-6/06/$20.00

lapping clusters with the property that for any node, there exists a cluster in the cover that contains its t-neighborhood (neighbors at distance at most t) with t 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 provide decompositions satisfying some desirable properties, it is important to efficiently construct these representations. In [4], the authors give a deterministic (resp. randomized) distributed algorithm to construct a $(k, t, \mathcal{O}(kn^{1/k}))$-neighborhood cover in $\mathcal{O}(tk \cdot 2^{c\sqrt{\log n}} + tk^2 \cdot 2^{4\sqrt{\log n}} \cdot n^{1/k})$ (resp. $\mathcal{O}(tk^2 \cdot \log^2 n \cdot n^{1/k})$) time for some constant $c > 0$. A (k, t, Δ)-neighborhood consists of possibly overlapping clusters with strong diameter at most $\mathcal{O}(kt)$ and such that *(i)* each node belongs to at most Δ clusters and *(ii)* the t-neighborhood of each node is covered by at least one cluster. In addition, the authors in [4] remark at the end of their paper that it is possible by using techniques from [15, 7] to translate their cover into a strong-network decomposition of comparable parameters, i.e., slightly worst than the *Basic Partition* we are interested in. Although, the algorithm one can obtain using the method of [4] is sublinear, the distributed model of computing considered there does not take into account the congestion created at various bottlenecks in the network (see Section 3.4 of [4]). In fact, the network model used in [4] is the Linial's *free* model [22, 23] (known as the $\mathcal{LOCAL}$ model, see [29] Chapter 2). This model assumes that nodes can communicate by exchanging messages of *unlimited size*. This assumption focuses on the locality nature of some distributed problems, i.e., what can be computed distributively assuming every node knows its whole neighborhood at some distance? In order to illustrate the mathematical power of such assumption, we give two examples used in [4]. First, in that model, assuming the nodes of the graph have unique identities, all distributed tasks can be computed in a time bounded by the diameter of the graph; by electing a leader and letting the leader learn the topology of the whole graph. Second, given a graph G, an integer d and a distributed task which is τ time consuming, computing the same task on the logical graph G^d (the graph obtained by adding an edge between any two nodes at distance d or less in G) is only $d \cdot \tau$ time consuming. Although, the Linial's model allows to express the locality of some problems, it fails to take into account some other important distributed features such as the message congestion. Moreover, when analyzing the algorithm of [4], it appears that no trivial methods enable to derive new algorithms in a distributed model of computation where the message size is limited, without dramatically deteriorating the time complexity.

From a practical point of view and because network decompositions are in the basis of many fundamental distributed problems such as synchronizers and routing, it is crucial to be able to design fast algorithms to construct such representations using a more realistic distributed model. From a theoretical point of view, it is also interesting and challenging to design fast distributed algorithms assuming less strong distributed models e.g., see [30]. In [27], Moran and Snir gave a distributed algorithm that computes a *Basic Partition* in $\mathcal{O}(n)$ time in a distributed model where the message size is at most $\mathcal{O}(\log n)$ bits, i.e., $\mathcal{CONGEST}$ model, [29] Chapter 2. The algorithm of [27] improves previous constructions used in [3, 32] which allows to obtain more efficient algorithms for designing network synchronizers γ, γ_1 and γ_2. However, The algorithm given in [27] is semi-sequential: Each cluster is constructed around some node in a distributed and layered fashion (a layer is added to the cluster if it increases the size of the cluster by a constant fraction). Nevertheless, the clusters forming the decomposition are constructed one after the other which requires to select at each iteration the next node around which the next cluster will grow. More recently, a fully distributed algorithm for constructing a *Basic Partition* was given in [17]. The algorithm there is based on a simple and practical framework to grow clusters in parallel and independently, and by applying some rules to break the symmetry. The algorithm has $\mathcal{O}(n)$ time complexity in the worst case and uses messages of size at most $\mathcal{O}(\log n)$.

1.3. Our results

This paper presents new techniques allowing to achieve sublinear time constructions (using message of size at most $\mathcal{O}(\log n)$) which answers a question asked by Moran and Snir in [27]. Our algorithms are fully distributed and do not need any precomputation step (such as computing a spanning tree cf. [27]). In fact, we let the clusters grow in parallel and in a concurrent way breaking ties using node identities. In our approach, we privilege the construction of clusters in the dense regions of the graph. This allows to finish the distributed construction in constant time as soon as the graph becomes sparse (it contains few edges). We obtain a deterministic synchronous distributed algorithm *Sync_Part* with $\mathcal{O}(k^2 \cdot n^{1-\frac{1}{k}})$ time complexity in the worst case. We also give an improved deterministic algorithm *Fast_Part* which does not make use of any global clock. Furthermore, we design a random-

ized distributed algorithm *Elect_Part* which is based on local elections in balls of radius k. This technique is a generalization of the algorithms given in [26] which can be of an independent interest. Using randomization enables us to focus on the degree of parallelism of our construction and allows to obtain an improved time complexity for particular graphs. Finally, our algorithms allow to obtain the first sublinear deterministic algorithm to construct a sparse graph spanners assuming a $\mathcal{O}(\log n)$ limitation on the message size. More precisely, we obtain a deterministic distributed construction of a $(2k-1)$-multiplicative stretch spanner with $\mathcal{O}(n^{1+1/k})$ edges in $\mathcal{O}(n^{1-\frac{1}{k}})$ time (for small k).

1.4. Outline of the paper

The paper is organized as follows. In the first part of Section 2, we define the model and the basic notations that will be used in the remainder of the paper. In the second part, we review past algorithms to construct the *Basic Partition* we are interested in. In Section 3, we present two deterministic sublinear time distributed algorithms (*Sync_Part* and *Fast_Part*). In Section 4, we present a randomized algorithm *Elect_Part* and we analyze its expected time complexity. In Section 5, we present a sublinear deterministic construction of sparse graph spanners. Section 6 concludes the paper.

2. Preliminaries

2.1. 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 [27, 3] which is also 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 $\mathcal{O}(\log(n))$ 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 that generates pulses. A message that have 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 this 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 S is a subset of V such that the subgraph induced by S is connected. A cluster is always considered with a leader node and a spanning tree rooted at some node: the *leader* or the *root* of the cluster. A cluster that contains only one node is called a *single-node cluster*. We also assume that each node v of a graph G has a unique identity ID_v (of $\mathcal{O}(\log(n))$ bits). If node identities are not provided then each node can select an identity at random from a large set of integers which guarantees that identities are unique with high probability. This is a classical and relevant assumption made in related past works. Under this assumption, each node v of the graph knows its own identity ID_v and the identity ID_S of the cluster S it belongs to. The identity of a cluster is defined as the identity of its leader.

In all the algorithms we will introduce, 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 said *finished* if it belongs to the final decomposition we are constructing. A finished cluster is no longer taken into account in the algorithm computations. A *finished* node is part of a finished cluster. A node is said *active* if it is still not in any finished cluster. In the sequel, we will denote by V_f the set of finished nodes. Furthermore, at each step of an algorithm execution, we are interested in active nodes in $V - V_f$, hence the degree of a node v is defined as its degree in the graph $G_{|V-V_f}$ induced by $V - V_f$. We also define the l-neighborhood $N_l(v)$ of a node v as the set of nodes at distance at most l from v in $G_{|V-V_f}$.

2.2. Algorithms *Basic_Part* and *Dist_Part*

One of the most basic algorithms to construct a sparse partition of a graph is algorithm *Basic_Part* which was introduced in [3] and then improved in [27]. A formal description of this algorithm is given in Figure 1 where $\mathcal{P}$ is the set of constructed clusters, k is an integer parameter and $\Gamma(S)$ is the set of nodes in the cluster S together with their neighbors, i.e., $\Gamma(S) = \bigcup_{v \in S} N_1(v)$.

Algorithm *Basic_Part* is inherently sequential as it constructs clusters one by one starting from a single node and adding a layer at each iteration. A layer is

```
Set P ← ∅
while V ≠ ∅ do
  Select an arbitrary vertex v ∈ V
  Set S ← {v}
  while |Γ(S)| > n^{1/k}|S| do
    S ← Γ(S)
  end while
  Set P ← P ∪ S and V ← V − S
end while
return P
```

Figure 1. Algorithm $Basic\_Part$ **[29]**

```
continue ← True
while continue do
  execute the Exploration Rule
  if success of the Exploration Rule then
    add the new layer
    execute the Growth Rule
    if Non success of the Growth Rule then
      reject the last explored layer
      switch to a finished cluster
      continue ← False
    end if
  else
    execute the Battle Rule
  end if
end while
```

Figure 2. Algorithm $Dist\_Part$ **[17]: code for a cluster**

added only if the sparsity condition $|\Gamma(S)| > n^{\frac{1}{k}}|S|$ is verified. Once a cluster is constructed, a new leader is chosen from the set of active nodes and a new cluster grows up around this center node. It is obvious that clusters constructed by algorithm $Basic\_Part$ are disjoint. Furthermore, the sparsity condition ensures that a cluster can not add more than $k-1$ layers. Hence, the following result holds:

Theorem 1 ([29]) *The output $\mathcal{P}$ of algorithm $Basic\_Part$ is a partition of G and:*

1. *$Rad(\mathcal{P}) \leqslant k-1$. ($Rad(\mathcal{P})$ is the maximum radius of any cluster in $\mathcal{P}$)*
2. *The graph induced by the clusters of $\mathcal{P}$ contains at most $n^{1+1/k}$ intercluster edges.*

In [17], the authors have proposed a new fully distributed algorithm $Dist\_Part$ that emulates the $Basic\_Part$ algorithm. One of the most interesting features of that algorithm is to allow clusters to grow in parallel and in a concurrent way. The algorithm starts with single-node clusters, then each cluster tries to add new layers according to some rules. These rules both control the cluster growth and manage the conflicts occurring between neighboring clusters using node identities to break the symmetry. Figure 2 gives a high level description of algorithm $Dist\_Part$ which is based on the following three rules:

1. *Exploration Rule*: a cluster is able to add a new layer if its identity 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.
3. *Battle Rule*: a cluster must give up its last layer if at least a neighboring cluster at distance one has successfully applied the *Exploration Rule*.

The *Exploration Rule* can be viewed as an attack hold by a cluster against its neighboring clusters in order to win a new layer. If a new layer is explored by a cluster S, then S is sure that no other clusters are sharing some nodes in this layer. A cluster may lose exploring a new layer, and it may be invaded by some neighboring clusters which have successfully performed their exploration. In this case, the invaded cluster loses its whole last layer because the sparsity condition may no longer hold. Note that a layer is first explored and then added to the cluster if it verifies the sparsity condition (*Growth Rule*).

In past sequential implementations [27, 29] of algorithm $Basic\_Part$, a spanning tree of the graph was precomputed and a breadth first search was used in order to find the next center around which a new cluster will grow. In opposite, the three rules of algorithm $Dist\_Part$ allow us to avoid electing a new center each time the construction of a cluster is finished. In addition, many clusters may be constructed independently in parallel. It is also easy to see that using classical convergecast and broadcast techniques, the three rules can be implemented using messages of size $\mathcal{O}(\log n)$ bits. For a more detailed description of algorithm $Dist\_Part$, its implementation and its analysis, the reader can refer to [17]. In particular, it has been proved that the time complexity of algorithm $Dist\_Part$ is $\mathcal{O}(n)$ in the worst case. In the next sections, we show new techniques inspired by [17] allowing

us to construct the decomposition produced by algorithm *Basic_Part* in sublinear time.

3. Sublinear Deterministic Distributed Partition

3.1. Algorithm *Sync_Part*

Preliminary In this section, we assume the network to be synchronous, i.e., there exists a global clock. At any time t, A_t denotes the set of active nodes, i.e., nodes in $V - V_f$, and $R_t = \{v \in A_t, d_v \geqslant n^{\frac{1}{k}}\}$ denotes the set of active nodes having high enough degrees.

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

In practice, the new algorithm *Sync_Part* works in two stages. The first stage is performed until time $\mathcal{T} = 10\ k^2 n^{1-1/k}$. The second stage begins at time $\mathcal{T}$. 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 $-\infty$ (this avoids to use additional labels and makes the algorithm easier to understand)*
- *Single-node clusters rooted at nodes in $A_t \setminus R_t$ do not explore any layer.*

These modifications concerns only single-node clusters which do not verify the sparsity condition. We use the same three rules of algorithm *Dist_Part* to manage the growth of clusters rooted at any node in R_t. Let us consider a single-node cluster rooted at $v \in A_t \setminus R_t$. When applying the new rules, v sets its identity to $-\infty$. Hence, v has the lowest identity among all other possible identities. In consequence, it will not stop the growth of another cluster whose leader is in R_t. In fact, v can only be 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 acts as if it has the lowest identity, i.e., it does not explore any layer. In practice, 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, 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 single-node clusters.

Lemma 1 *Considering a cluster with the biggest identity among active nodes, we need $10k^2$ time in the worst case before its construction is finished.*

Lemma 2 *At any time t such that $R_t \neq \varnothing$, $|A_{t+10k^2}| < |A_t| - n^{1/k}$*

Proof. (Sketch) At time t such that $R_t \neq \varnothing$, we consider the node v in R_t with the biggest identity. Let us denote S the cluster rooted at v. Using Lemma 1, at time $t' = t + 10k^2$, at least the $n^{1/k}$ nodes in the first layer of cluster S are not in $A_{t'}$. □

Lemma 3 *For $t = 10k^2 n^{1-1/k}$, $R_t = \varnothing$.*

Proof. Using Lemma 2, at some time t, if $R_t \neq \varnothing$ then after a $10k^2$ time units period p, the number of active nodes will decrease by at least $n^{1/k}$. By induction, after $i \geqslant 0$ periods p, if $R_t \neq \varnothing$ then $|A_t| < n - i\, n^{1/k}$. Using the fact that $R_t \subseteq A_t$, we have at most $i = n^{1-1/k}$ periods p such that $R_t \neq \varnothing$. □

Theorem 2 (Correctness) *Algorithm Sync_Part emulates the Basic_Part algorithm.*

Proof. (Sketch) The correctness of the algorithm relies on the fact that after time $\mathcal{T}$, no clusters of radius greater or equal to 1 can be constructed (Lemma 3). □

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

Theorem 3 (Time Complexity) *The time complexity of algorithm Sync_Part is $\mathcal{O}(k^2\ n^{1-1/k})$.*

Remark 1 *One can show that if we privilege the growth of clusters having the biggest couple (Radius,ID), then $\mathcal{T}$ can be chosen such that $\mathcal{T}$=$\mathcal{O}(n^{1-1/k})$ for relevant range of $k < \log(n)$. This requires to show that a finished cluster of radius l has at least $n^{\frac{l}{k}}$ nodes and that it was constructed in at most $\mathcal{O}(l^2)$ time, and thus we can prove that at each*

$\mathcal{O}(1)$ *time units, the number of nodes becoming part of the cluster having the biggest couple (Radius,ID) is at least* $n^{\frac{1}{k}}$*. For clarity of our algorithms and because* k *is taken to be small, we omit details and proofs.*

3.2. Algorithm *Fast_Part*

Preliminary Algorithm *Sync_Part* uses the property that the system is synchronous to find a bound on the time $\mathcal{T}$ after which there are no nodes able to 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 systems without using any synchronizer (see [29]). Hence, we propose a new asynchronous algorithm *Fast_Part* that does not use any global clock. The general idea of the algorithm is to allow sparse clusters to become finished without waiting until a pulse $\mathcal{T}$.

Details of the algorithm Let us call a cluster S *dense* if S has a radius at least 1 or if the single node v of S is such that $d_v > n^{\frac{1}{k}}$. We also define a *sparse cluster* to be a single-node cluster which is not dense (this corresponds to a node in $A_t \setminus R_t$ in algorithm *Sync_Part*). Algorithm *Fast_Part* runs like algorithm *Dist_Part* with the following *Exploration Rule*:

- *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 single-node cluster:*
  - *if all its neighbors are sparse,*
  - *or if none of its dense neighbors has decided to explore 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 may happen 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.

It is obvious that the new exploration rule can be implemented using messages of size at most $\mathcal{O}(\log(n))$ using the same techniques than in algorithm *Dist_Part* [17]. In fact, using, for example, a couple *(ID,Dense)*, where *Dense* is a boolean variable indicating whether a cluster is dense or sparse, allows us to take into account the new modifications. Nevertheless, because a sparse node (having dense neighbors) decides to be finished if it was not explored, it must inform its neighbors by sending them a message. This enables neighboring nodes to be aware of their degrees.

Theorem 4 (Correctness) *Algorithm Fast_Part emulates algorithm Basic_Part.*

Proof. (Sketch) The correctness of the algorithm is ensured by the following three facts : *(i)* two clusters at distance at most 2 can not explore the same node *(ii)* dense clusters grow in a layered fashion with respect to the sparsity condition *(iii)* if a cluster wins in exploring a new layer then none of its sparse neighbors will be allowed to declare itself finished. □

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

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

Proof. The new exploration rule guarantees that a dense cluster is never stopped in its growth by a sparse one. In the worst case, no two dense clusters are constructed in parallel. Hence, we consider finished dense clusters sorted in a decreasing order of their time construction. The construction of one cluster has cost at most $\mathcal{O}(k^2)$. Thus, after at most $\mathcal{O}(k^2\,\Lambda)$ 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, 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 2 *Note that we can apply Remark* 1 *for the asynchronous algorithm Fast_Part in order to obtain a* $\mathcal{O}(n^{1-\frac{1}{k}})$ *time complexity.*

Remark 3 *The bound* $\mathcal{O}(k^2\,\Lambda)$ *becomes of special interest in the case of graphs for which* Λ *is known to be small comparing with* $n^{1-\frac{1}{k}}$*, e.g.,* Circulant *graphs.*

Remark 4 *Algorithm Fast_Part has the advantage to focus on clustering dense regions. In fact, 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 on dense regions. Moreover, the construction will be faster if dense areas are far from each other. This two facts do not hold for existing constructions.

4. Randomized Distributed Partition

4.1. Randomized local elections

In [26], a randomized algorithm called LE_2 (Local Election) is introduced to implement algorithms based on local computations and relabeling systems [25, 24, 2]. The LE_2-algorithm allows to simulate local computations on a closed star, that is, the labels (states) attached to the center and to the leaves of the star can be modified according to some rules. In algorithm LE_2 of [26], nodes are fighting to be centers of a ball of radius 1 so that the elected nodes can execute a computation step. Non elected nodes are part of at most one star which allows to execute local computations concurrently on closed balls of radius 1. For more details about the LE_2-algorithm the reader is referred to [26]. In particular, the authors studied the average number of nodes locally elected and they interpreted that as the degree of parallelism authorized by an algorithm. In the special case of our algorithm, that study gives an idea about the number of clusters constructed in parallel in one round and for $k = 2$. In fact, when taking $k = 2$ in the *Basic_Part* algorithm and using Theorem 1, the radius of the clusters produced by the decomposition can be either 0 (i.e., single-node clusters) or 1 (i.e., clusters containing a node v and its neighborhood $N_1(v)$). Therefore, we can use the LE_2-algorithm to produce the sparse partition we need. It suffices to run the LE_2-algorithm until there are no more active nodes in the graph. Every time a node v is center of a star, it computes its neighborhood $|N_1(v)|$ and decides to be either a radius 1 finished cluster or just a finished single-node cluster.

In the next paragraphs, we use a generalization (LE_k) of the algorithm of [26] in order to elect nodes which are centers of disjoint balls of radius $k \geqslant 2$. The LE_k algorithm is used as a subprocedure in algorithm *Elect_Part* in order to construct the graph partition.

4.2. Algorithm *Elect_Part*

Algorithm *Elect_Part* is depicted in Figure 3. It runs in many phases until each node of the graph belongs to a finished cluster. A phase of the algorithm is executed in two stages. The first stage consists in running algorithm LE_k (see Figure 4) which is a variant of algorithm *Dist_Part* in which the sparsity condition does not matter: only the radius of elected clusters is important. After the first stage, some balls of radius k and centered on some elected nodes are constructed. Note that some other balls of radius less then k are also constructed. The second stage is devoted to compute finished clusters and to re-initialize the computations. 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 of each ball and its 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 can not grow any more and it also declares itself finished (Step 2.d). Finally, the remaining clusters are just broken into single-node clusters in order to run another phase (Step 2.e).

Note that, 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 thus avoiding the preferred edge election step needed for some applications. This is discussed in more details in Section 5.

while There exist nodes not in a finished cluster **do**
  (0.) each node selects randomly an identity from a big set of integers.
  Stage 1: **local election in balls of radius** k
  (1.a) Each node v not in a finished cluster runs algorithm LE_k
  Stage 2: **reinitialization**
  (2.a) Each formed cluster S computes independently the sparsity condition for each layer $j \leqslant k$,
  if S contains a layer j violating the sparsity condition **then**
    (2.b) S releases all layers $l \geqslant j$ and becomes a finished cluster,
    (2.c) nodes in released layers become single-node clusters.
  else
    if all neighbors are finished **then**
      (2.d) S becomes finished.
    end if
  end if
  (2.e) Break all non finished clusters and form new single--node clusters.
end while

Figure 3. Algorithm *Elect_Part*

```
Round ← 0;
while Round < k do
  execute the Exploration Rule;
  Round ← Round + 1;
  if Non Success of the Exploration Rule then
    execute the Battle Rule;
  end if
end while
```

Figure 4. Algorithm LE_k: code for a cluster

4.3. Analysis of the algorithm

In this section, we compute a bound on 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 has succeeded the LE_k phase without losing against any other cluster. We also use a parameter K such that: $\forall v \in V, N_{2k}(v) \leqslant K$.

Inspired by proofs in [26], the following proposition holds:

Proposition 1 *The expected number of nodes locally k-elected in a phase satisfies:*

$$\overline{M}_k(G_{|V-V_f}) = \sum_{v \in V-V_f} \frac{1}{N_{2k}(v)} > \frac{|V-V_f|}{K}$$

Theorem 6 *Let T be the time complexity of the algorithm Elect_Part. The expected value of T satisfies:*

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

Proof. (Sketch) Let G_i denotes the graph induced by not finished nodes at step i. Let the random variable X_i denotes the number of nodes of G_i. Let the random variables Y_i denotes respectively the number of nodes locally k-elected in the i^{th} step, i.e., in G_i. Using Proposition 1, one can show that the expected number of Y_i satisfies $E(Y_i \mid G_i) \geqslant X_i/K$. Then, using some further probabilistic arguments to bound $E(Y_{i+1})$ using $E(Y_i)$, one can show that $E(X_i) \leqslant n\left(1-\frac{1}{K}\right)^i$. The theorem is proved by noting that the algorithm terminates when $X_i = 1$. □

Remark 5 *The bound given by Theorem 6 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.*

4.4. Discussion

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 single node cluster. Thus, we should apply some new rules similar to those of algorithm *Fast_Part* in order to *(i)* prohibit that a sparse node keeps a dense one from growing *(ii)* and to allow a sparse node to declare itself finished if it is not explored by any neighbor. In consequence, we obtain a modified version of algorithm *Elect_Part* in which only dense nodes can be k-elected. In this modified version, only dense nodes are allowed to compete in order to grow a ball of radius k. Just like in algorithm *Fast_Part*, we let a dense node always wins against a sparse one using the couple $(ID, Dense)$. Furthermore, a sparse node is allowed to declare itself finished cluster if it is not invaded by any neighboring cluster i.e., if dense neighbors loose 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 6, 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 6. It is also easy to prove using the same arguments than in Theorem 5 that the complexity of the modified algorithm is bounded by $\mathcal{O}(k^2 \cdot \Lambda)$. The new version of algorithm *Elect_Part* is of special interest because it has a sublinear time complexity for general graphs and by the same time it allows to express the high degree of parallelism of our method when analyzing the time complexity for some particular graphs. For example, let us consider a graph G such that $K = \mathcal{O}(n^\epsilon)$ with $\epsilon < 1$. This defines a large class of graphs for which we can achieve an improved time complexity, namely $\mathcal{O}(k^2 \log(n) n^\epsilon)$.

Remark 6 *Both our deterministic and randomized algorithms are directly implementable on real data. Some experiments using some well chosen critical graphs*

have been done with ViSiDiA *[16], i.e., a platform which enables to visualize and to simulate on the fly the execution of a distributed algorithm. Our experiments confirm that, in practice, our techniques allow a high degree of parallelism. Furthermore, most of the graphs we have used are quickly portioned in many independent connected components which enhance the time complexity, i.e., this was not taken into account in our theoretical analysis.*

5. Application to Graph Spanners

An (α,β)-spanner of a graph is a subgraph H such that for any two nodes u,v: $d_H(u,v) \leqslant \alpha.d_G(u,v) + \beta$ where $d_H(u,v)$ (resp. $d_G(u,v)$) is the distance in H (resp. in G) between u and v. There are three features to take into account when evaluating spanner constructions: the size of the spanner (space), the approximation quality (stretch) and the construction time. Many efforts have been done in order to give algorithms that improve one (or all) of these features. In particular, it is well known that there exist $(2k-1,0)$-spanners with size $\mathcal{O}(n^{1+\frac{1}{k}})$ which is optimal [1, 13, 31].

One immediate application of algorithm *Basic_Part* is the construction of a $(4k-3,0)$-spanner with $\mathcal{O}(n^{1+\frac{1}{k}})$ edges. The spanner is obtained by considering the set of edges spanning each cluster and by selecting an intercluster edge for each pair of two neighboring clusters. The bounds on the stretch and the size of the spanner follow from Theorem 1. Therefore, in order to distributively construct such a spanner, we have to select an edge between every two neighboring clusters. Nevertheless, we can avoid this additional step of selecting preferred edges and by the same time the stretch can be improved without changing the bound on the spanner size.

In fact, let us consider a cluster S under construction. Before completing the construction of S (i.e., just after the sparsity condition is no longer satisfied), for every neighboring vertex u of S (i.e., u is on the last rejected layer of S), we select an edge from u to some v in the last layer of S. Thus, we obtain a $(2k-1,0)$-spanner with the same size bound by considering the selected edges and the trees spanning the clusters. This idea was attributed to [19] in [29] (Exercise 3, page 188) and was used in [18] as a first step to construct $(1+\epsilon,\beta)$-spanners. The same idea was also used in [27] to improve the complexity of synchronizers γ_1 and γ_2. The time complexity of the algorithms used in [29, 27] was $\mathcal{O}(n)$ and it has not been improved since.

Let us remark that in all our algorithms the last rejected layer is always explored. Hence, the edges connecting a cluster with nodes in the last rejected layer can be computed without any extra time. Hence, the following holds:

Theorem 7 *There is a deterministic algorithm that given a graph with n nodes and a fixed integer $k \geqslant 1$, constructs a $(2k-1,0)$-spanner with $\mathcal{O}(n^{1+1/k})$ edges in $\mathcal{O}(n^{1-1/k})$ time in the worst case.*

Corollary 8 *There is a deterministic algorithm that given a graph with n nodes constructs a $(3,0)$-spanner with $\mathcal{O}(n^{3/2})$ edges in $\mathcal{O}(\sqrt{n})$ time in the worst case.*

To our knowledge, this is the fastest deterministic algorithm that constructs such spanners in the $\mathcal{CONGEST}$ model.

Remark 7 *Although it is very hard to construct spanners deterministically, there already exist randomized efficient algorithms. In particular, a randomized algorithm for constructing a $(2k-1,0)$-spanner with expected size $\mathcal{O}(k\, n^{1+\frac{1}{k}})$ was introduced in [11]. A randomized distributed algorithm for that construction was given in [10] with a running time of $\mathcal{O}(k)$. Note that the randomization in the algorithm of [11] is on the size of the spanner which can be unsatisfying in applications where the correctness of the spanner is crucial.*

6. Conclusion

The algorithms given in this paper improve the time complexity of past constructions and are implementable in the practical $\mathcal{CONGEST}$ distributed model. Since the decomposition described in this paper is used in many other settings, we are hopeful that our construction will help improving other applications that use the *Basic_Part* algorithm as a starting point. For instance, one can show that with some few modifications, our technique improves the time complexity of the preprocessing phase of synchronizers γ_1 and γ_2 from $\mathcal{O}(n)$ to $\mathcal{O}(n^{1-\frac{1}{k}})$.

We can note that, in the case $k = 2$, the time complexity bound obtained using our technique is the same as the bound one can obtain by both assuming a more powerful distributed model i.e., unlimited message size, and using techniques from [4]. In the case of sparse spanners, this remark is intriguing and one can be interested in a lower bound on the time complexity of distributively computing a $(3,0)$-spanner. We are optimistic that deterministic algorithms with better bounds exist.

It is also unknown if there exists poly-logarithmic algorithm for the basic sparse partition problem studied in this paper. We are optimistic that poly-logarithmic

algorithms exist even in the $\mathcal{CONGEST}$ distributed model.

Acknowledgment We would like to thank Ph. Duchon and O. Bernardi for helpful discussions.

References

[1] I. Althofer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9:81–100, 1993.

[2] D. Angluin. Local and global properties in networks of processors. 12^{th} *Symp. on Theory of Computing*, pages 82–93, 1980.

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

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

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

[6] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Poltkin. Network decomposition and locality in distributed computation. 30^{th} *IEEE Symp. on Found. of Computer Science*, pages 364–369, 1989.

[7] B. Awerbuch and D. Peleg. Sparse partitions. 31^{st} *IEEE Symp. on Found. of Comp. Science*, pages 514–522, 1990.

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

[9] B. Awerbuch and D. Peleg. Online tracking of mobile users. *J. of the ACM*, 42:1021–1058, 1995.

[10] S. Baswana, T. Kavitha, K. Mehlhorn, and S. Pettie. New constructions of (α,β)-spanners and purely additive spanners. 16^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, To appear, 2005.

[11] S. Baswana and S. Sen. A simple linear time algorithm for computing $(2k-1)$-spanner of $\mathcal{O}(n^{1+1/k})$ size for weighted graphs. *Int. Colloquium on Automata, Languages and Programming*, pages 384–296, 2003.

[12] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $\mathcal{O}(n^2 \log(n))$ time. 15^{th} *Annual ACM-SIAM Symposium On Discrete Algorithms*, New Orleans, LA, USA.:271–280, 2004.

[13] B. Chandra, G. Das, G. Narasimhan, and J. Sores. New sparseness results on graph spanners. *Proceedings of the* 8^{th} *annual symposium on Computational geometry*, pages 192–201, 1992.

[14] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.

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

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

[17] B. Derbel and M. Mosbah. A fully distributed linear time algorithm for cluster network decomposition. 16^{th} *IASTED Int. Conf. on Parallel and Distributed Computing and Systems, MIT*, pages 548–553, 2004.

[18] M. Elkin and D. Peleg. ($1+\epsilon$, β)-spanner constructions for general graphs. *Siam J. Comput*, 33:608–631, 2004.

[19] S. Halperin and U. Zwick. Unpublished result. 1996.

[20] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer. Fast deterministic distributed maximal independent set computation on growth-bounded graphs. In 19^{th} *International Symposium on Distributed Computing (DISC)*, Sept. 2005.

[21] F. Kuhn, T. Moscibroda, and R. Wattenhofer. On the locality of bounded growth. In 24^{th} *ACM Symp. on the Principles of Dist. Comp. (PODC)*, 2005.

[22] N. Linial. Distributive graph algorithms - global solutions from local data. 28^{th} *IEEE Symp. Found. of Computer Science*, pages 331–335, 1987.

[23] N. Linial. Locality in distributed graph algorithms. *SIAM J. Computing*, 21:1:193 – 201, 1992.

[24] I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.

[25] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

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

[27] S. Moran and S. Snir. Simple and efficient network decomposition and synchronization. *Theoretical Computer Science*, 243:217–241, 2000.

[28] A. Panconesi and A. Srinivasan. Improved distributed algorithms for coloring and network decomposition. 24^{th} *ACM Symp. on Theory of Computing*, pages 581–592, 1992.

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

[30] D. Peleg and V. Rubinovich. A near-tight bound on the time complexity of distributed minimum-weight spanning tree. *SIAM J. COMPT*, 32(5):1427–1442, 2000.

[31] D. Peleg and A. Schaffer. Graph spanners. *J. Graph Theory*, 13:99–116, 1989.

[32] L. Shabtay and A. Segall. Low complexity network synchronization. 8^{th} *Internat. Workshop on Distributed Algorithms*, pages 223–237, 1994.

[33] M. Thorup and U. Zwick. Approximate distance oracles. *33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 183–192, 2001.