Distributed Computing manuscript No.
(will be inserted by the editor)

Efficient Algorithms for Constructing (1+ϵ,β)-Spanners in the Distributed and Streaming Models[*]

Michael Elkin[**], **Jian Zhang**[***]

Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520-8285, USA

e-mail: {elkin, zhang-jian}@cs.yale.edu

Summary. For an unweighted undirected graph $G = (V, E)$, and a pair of positive integers $\alpha \geq 1$, $\beta \geq 0$, a subgraph $G' = (V, H)$, $H \subseteq E$, is called an (α, β)-*spanner of* G if for every pair of vertices $u, v \in V$, $dist_{G'}(u, v) \leq \alpha \cdot dist_G(u, v) + \beta$.

It was shown in [21] that for any $\epsilon > 0$, $\kappa = 1, 2, \ldots$, there exists an integer $\beta = \beta(\epsilon, \kappa)$ such that for every n-vertex graph G there exists a $(1 + \epsilon, \beta)$-spanner G' with $O(n^{1+1/\kappa})$ edges. An efficient distributed protocol for constructing $(1 + \epsilon, \beta)$-spanners was devised in [19]. The running time and the communication complexity of that protocol are $O(n^{1+\rho})$ and $O(|E|n^{\rho})$, respectively, where ρ is an additional control parameter of the protocol that affects only the additive term β.

In this paper we devise a protocol with a drastically improved running time ($O(n^{\rho})$ as opposed to $O(n^{1+\rho})$) for constructing $(1 + \epsilon, \beta)$-spanners. Our protocol has the same communication complexity as the protocol of [19], and it constructs spanners with essentially the same properties as the spanners that are constructed by the protocol of [19]. The protocol can be easily extended to a parallel implementation which runs in $O(\log n + (|E| \cdot n^{\rho} \log n)/p)$ time using p processors in the EREW PRAM model. In particular, when the number of processors, p, is at least $|E| \cdot n^{\rho}$, the running time of the algorithm is $O(\log n)$. We also show that our protocol for constructing $(1 + \epsilon, \beta)$-spanners can be adapted to the streaming model, and devise a streaming algorithm that uses a constant number of passes and $O(n^{1+1/\kappa} \cdot \log n)$ bits of space for computing all-pairs-

[*] Preliminary version of this paper was published in PODC'04, see [22].

[**] The author's work on this paper was Supported by the DoD University Research Initiative (URI) administered by the Office of Naval Research under Grant N00014-01-1-0795..

[***] Supported by ONR grant N00014-01-1-0795 and NSF grants CCR-0105337 and ITR-0331548.

Correspondence to: Jian Zhang (zhang-jian@cs.yale.edu)

almost-shortest-paths of length at most by a multiplicative factor $(1 + \epsilon)$ and an additive term of β greater than the shortest paths. Our algorithm processes each edge in time $O(n^\rho)$, for an arbitrarily small $\rho > 0$. The only previously known algorithm for the problem [23] constructs paths of length κ times greater than the shortest paths, has the same space requirements as our algorithm, but requires $O(n^{1+1/\kappa})$ time for processing each edge of the input graph. However, the algorithm of [23] uses just one pass over the input, as opposed to the constant number of passes in our algorithm[1].

We also show that any streaming algorithm for $o(n)$-approximate distance computation requires $\Omega(n)$ bits of space.

Keywords: Spanner – Almost Shortest Paths – Streaming Model

1 Introduction

In this paper we study the problem of constructing *spanners* and computing *almost shortest paths* in the *distributed* and *streaming* models of computation.

1.1 Distributed Model

Consider an unweighted undirected graph $G = (V, E)$. Observe that the graph G induces a *metric space* $\mathcal{U}$ in which the vertex set V serves as the set of points, and the lengths of the shortest paths serve as *distances*. Intuitively, a graph *spanner* $G' = (V, H), H \subseteq E$ is a *sparse skeleton* of the graph G whose induced metric space $\mathcal{U}'$ is a close approximation of the metric space $\mathcal{U}$ of the graph G. Graph spanners have multiple applications in the areas of Graph Algorithms and Distributed Computing, and were subject of extensive research [33,4,13,11,18,21,19,36,9,10] in the course of the last fifteen years. In particular, spanners were used for routing [35,7], for constructing synchronizers [34,6], and for computing almost shortest paths [4,19].

In the area of Distributed Computing spanners were found particularly useful in the following (henceforth called *distributed*) model of computation. In this model every vertex of an n-vertex graph $G = (V, E)$ hosts a processor with an *unbounded computational power* only *limited knowledge*. Specifically, it is assumed that in the beginning of the computation every processor v knows only the identities of its *neighbors*. The communication is *synchronous* and proceeds in *discrete pulses*, called *rounds*. On each round each processor is allowed to send short (possibly different) messages to all its neighbors. The (worst-case) *running time* of a distributed algorithm is the (worst-case) number of rounds that are required for the algorithm to complete its execution. The design of distributed algorithms is a vivid area of research

[1] After the preliminary version of our paper [22] appeared on PODC'04, Feigenbaum et al. [24] came up with a new streaming algorithm for the problem that is far more efficient than [23] in terms of time-per-edge processing. However, our algorithm is still the only existing streaming algorithm that provides an *almost additive* approximation of distances.

(see, *e.g.*, [32], and the references therein). Spanners serve as an important tool in this area, and particularly, they were used for routing [34, 13], for constructing synchronizers [34, 6], and for computing almost shortest paths [19].

For all these applications it is crucially important that spanner is a part of the original network, and consequently, the processors can communicate over its links. In particular, the processors can execute over the links of a spanner any protocol that was designed for arbitrary networks. Also, since a spanner approximates the distances of the original network, the execution of a protocol on a spanner is almost as time-efficient as its execution on the original network (spanned by the spanner). However, since spanners are typically *much sparser* than the networks they span, an execution of a protocol on a spanner is typically *much more communication-efficient* than the corresponding execution on the original network. These properties make spanners extremely valuable in the design of distributed protocols, and raise the problem of designing efficient distributed protocols for constructing spanners with good parameters. In this paper we address this problem.

In the 90's most of the study of spanners and their applications focused on spanners whose metric space distorts the original metric space by at most a constant *multiplicative* factor. More formally, for a positive integer $\kappa = 1, 2, \ldots$, a subgraph $G' = (V, H)$ is a *κ-spanner* of the graph $G = (V, E)$, $H \subseteq E$, if for every pair of vertices $u, v \in V$, $dist_{G'}(u, v) \leq \kappa \cdot dist_G(u, v)$ (where $dist_G(u, v)$ stands for the *distance* between the vertices u and v in the graph G). A fundamental theorem concerning κ-spanners, that was proven by Peleg and Schäffer [33], says that for every n-vertex graph $G = (V, E)$ and a positive integer $\kappa = 1, 2, \ldots$, there exists a κ-spanner with $n^{1+O(\frac{1}{\kappa})}$ edges, and that this is the best possible up to the constant hidden by the O-notation.

More recently, Elkin and Peleg studied a more general notion of (α, β)-*spanner*: a subgraph $G' = (V, H)$ of the graph $G = (V, E)$ is an (α, β)-*spanner* of the graph G if for every pair of vertices $u, v \in V$, $dist_{G'}(u, v) \leq \alpha \cdot dist_G(u, v) + \beta$. They have shown that for every n-vertex graph $G = (V, E)$, there exists a $(1 + \epsilon, \beta)$-spanner $G' = (V, H)$ of G with $O(n^{1+1/\kappa})$ edges. This result shows that the tradeoff of Peleg and Schäffer [33] can be drastically improved if one is concerned only with approximating the distances that are larger than a certain constant.

While the proof of Elkin and Peleg [21] is not known to translate to an efficient distributed algorithm, soon afterwards Elkin [19] came up with an alternative proof of this theorem, which though providing somewhat inferior constants, translates directly into efficient distributed and sequential algorithms. The latter algorithms enabled [19] to use $(1 + \epsilon, \beta)$-spanners for efficient algorithms for computing almost shortest paths from s sources.

Specifically, it was shown in [19] that for every n-vertex graph $G = (V, E)$, positive integer $\kappa = 1, 2, \ldots$, and positive numbers $\epsilon, \rho > 0$, there exists a distributed algorithm that constructs $(1+\epsilon, \beta)$-spanners with $O(n^{1+1/\kappa})$ edges in time $O(n^{1+\rho})$ with message complexity $O(|E| \cdot n^{\rho})$. Note that while the message complexity of this result is near-optimal, its running time is prohibitively large. In this paper we drastically improve this running time and devise a *randomized* distributed algorithm that constructs $(1+\epsilon, \beta)$-spanners with $O(n^{1+1/\kappa})$ edges in time $O(n^{\rho})$, and with message complexity $O(|E| \cdot n^{\rho})$. Note that the message complexity of our algorithm is no worse than the message complexity of the algorithm of [19], and the parameters of the constructed spanners are also essentially the same as in the result of [19]. This result directly translates to an improved distributed algorithm for computing almost shortest paths from s sources.

We remark that both our algorithm and the algorithm of [19] can be adapted to the *asynchronous* model of distributed computation in a rather straightforward way by using synchronizers of [5]. The parameters of the obtained asynchronous algorithms are essentially the same as the parameters of the synchronous algorithms. Furthermore, our algorithm can also be easily extended to a parallel implementation which runs in $O(\log n + (|E| \cdot n^{\rho} \log n)/p)$ time using p processors in the EREW PRAM model. Particularly, when the number of processors, p, is at least $|E| \cdot n^{\rho}$, the running time of the algorithm is $O(\log n)$. This is the first known parallel algorithm for constructing sparse $(1+\epsilon, \beta)$-spanners.

1.2 Streaming Model

We also adapt our distributed algorithm for constructing spanners to the *streaming* model. In this model, the computation is *centralized*, and is performed by a single processor. However, unlike the traditional computational models, in the streaming model the processor is not allowed to store the entire input in its random-access-memory, but rather instead has in its disposal a very limited space (which is much smaller than the size of the input). Furthermore, the algorithm is only allowed to read the input *sequentially* using a *small number of passes* through the input.

This computational model was developed in the works of [3,28,25], and is important for processing massive data sets (see, e.g., [31], and the references therein). While originally the research in this area focused on numerical problems such as estimating the frequency moments [3,16], and L^1- and L^2-distances [25,29], more recently the problems involving massive graphs have become a subject of study [31, 23]. In this paper we show that our distributed algorithm can be converted into a streaming algorithm for constructing $(1+\epsilon, \beta)$-spanners. The streaming algorithm go through the input a *constant number of passes*, uses $O(n^{1+1/\kappa} \cdot \log n)$ bits of space, and $O(n^{\rho})$ processing time-per-edge. ($\rho > 0$ is an arbitrarily small control parameter. It affects the number of passes of the algorithm, and the additive term of the constructed spanner). This result, in turn, directly gives rise to a streaming algorithm with the same complexity parameters (number of passes and space) that given an input graph computes *almost shortest paths between all pairs* of vertices of the graph.

Specifically, for each pair of vertices $u, w \in V$, the algorithm outputs a path of length at most $(1+\epsilon) \cdot dist_G(u, w) + \beta$. The only previously known streaming algorithm for this problem due to Feigenbaum *et al.* [23] produces paths of length at most $\kappa \cdot dist_G(u, v)$, using $O(n^{1+1/\kappa} \cdot \log n)$ bits of space, uses only one pass through the input, but requires $O(n^{1+1/\kappa})$ processing time-per-edge. In other words, while our algorithm uses more passes than the algorithm of Feigenbaum *et al.* [23], it provides far shorter paths and distance estimates, and has a drastically smaller processing time-per-edge. We remark that after the preliminary version of this paper was published in PODC'04 [22], Feigenbaum *et al.* [24] came up with another streaming algorithm for the same problem that uses both one pass through the input *and* small processing time-per-edge ($O(\kappa^2 \cdot \log n \cdot n^{1/\kappa})$). However, that algorithm still produces paths of length at most $\kappa \cdot dist_G(u, w)$, exactly as the algorithm of [23]. Consequently, our algorithm provides better distance estimates than algorithms of both [23] and [24]. Note also that our processing time-per-edge is better than the time-per-edge of the recent algorithm of [24], though not by much.

Note that, for all streaming algorithms for the problem, to store the results of the computation, *i.e.*, the all-pairs shortest paths and the all-pairs shortest distances, it may require a memory space much larger than the one for storing the spanner. On the other hand, an algorithm can output and erase from its memory right after outputting it each such a path or a distance as far as it keeps the entire spanner in its memory. In other words, an algorithm needs not to store the entire table of paths or distances, but rather to store the spanner and, in any given moment of time, one additional path or distance. (This is the path or distance that is being outputted at this moment.) Hence the total memory space required for implementing such an algorithm is no larger than the size of the spanner.

In the traditional (not streaming) model of computation, the problem of computing almost shortest paths was the focus of extensive recent research ([18, 15, 19]; see also the excellent survey [38] and the references therein). We remark that many of the algorithms for all-pairs-(almost)-shortest-paths in the traditional model of computation rely heavily either on matrix-multiplication [2, 26, 37], or on multiple BFS traversals of the *entire* graph [1, 18, 15]. Both these approaches appear to be completely unsuitable for the streaming model of computation.

Note also that the space complexity of our streaming algorithm $O(n^{1+1/\kappa} \cdot \log n)$ is not far from optimal one, since we show that *any streaming algorithm that provides $o(n)$-approximate estimate of the distance between a fixed pair of vertices in constant number of passes must use $\Omega(n)$ bits of space.*

We remark that most known streaming algorithms use only a small constant number of passes (such as 1 or 2). (A notable exception is the algorithm of Feigenbaum *et al.* [23]

for $(\frac{2}{3} - \epsilon)$-approximation of the maximum matching problem. Their algorithm uses $f(\epsilon) = O(1)$ passes.) It can be argued that our streaming algorithm (that uses $f(\epsilon, \rho, \kappa) = O(1)$ passes) is impractical, because it uses a significantly larger (though still constant) number of passes over the input. However it is argued in [23] that streaming algorithms that use a constant number of passes can be applied in the context of large datasets, even when the number of passes is significantly greater than 1. In addition, we believe that our result is of theoretical value, as it sheds some light both on the problem of distance estimation, and on the streaming model of computation. It is also conceivable that the number of passes that is used by our algorithm will be reduced in the future.

Finally, we remark that our adaptation of the algorithm of [20] for constructing neighborhood covers for the streaming model is one of the first existing *streaming algorithms for clustering in graphs*. Despite the fact that the clustering problem is extremely well-studied in different disciplines, we are only aware of very few previous streaming algorithms [27,12] for clustering metric spaces. However, to the best of our knowledge, these algorithms are not applicable to the problem of clustering in graphs, and hence our result is incomparable with those of [27,12].

1.3 Our Techniques

Our distributed algorithm for constructing $(1+\epsilon, \beta)$-spanners builds upon the previous work of [19]. The algorithm of [19], like our algorithm, uses extensively a subroutine for constructing *neighborhood covers* [4,13,19] (see Sec. 2 for its definition). In fact, both algorithms invoke this subroutine on many subgraphs of the original graph. The best distributed algorithm for constructing neighborhood covers that was available when the work of [19] was done is the algorithm of Awerbuch *et al.* [4], and the latter algorithm, by itself, requires a super-linear time. Recently a significantly more efficient subroutine for computing neighborhood covers was devised in [20]. However, plugging the subroutine of [20] in the algorithm of [19] does not result in a sub-linear algorithm for constructing $(1 + \epsilon, \beta)$-spanners, because the recursive invocations of the subroutine for constructing covers are implemented almost *sequentially* in the algorithm of [19]. The main technical difficulty that we had to overcome in this paper is the *parallelization* of these recursive invocations. The latter task requires far more elaborate analysis of the algorithm, as we have to show that no edge is simultaneously employed by more than a certain number of different subroutines.

2 Fast Distributed Algorithm for Constructing a Spanner

First, we introduce some notation. Let $G = (V, E)$ be a unweighted undirected graph. Denote by $dist_G(u, w)$ the *distance* between two vertices u and w in the graph G, that is, the length of the shortest path between them. For two subsets of vertices $V', V'' \in V$, the distance in G between

V' and V'', $dist_G(V', V'')$, is the shortest distance between a vertex in V' and a vertex in V'', i.e., $dist_G(V', V'') = min\{dist_G(u, w) \mid u \in V', w \in V''\}$. Let $diam(G)$ denote the *diameter* of the graph G, i.e., $diam(G) = max_{u,v \in V} dist_G(u, v)$.

Given a subset $V' \subseteq V$, denote by $E_G(V')$ the set of edges in G *induced* by V', i.e., $E_G(V') = \{(u, w) \mid (u, w) \in E \text{ and } u, w \in V'\}$. Let $G(V') = (V', E_G(V'))$. Denote by $\Gamma_k(v, V')$ the *k-neighborhood* of vertex v in the graph $G(V')$, i.e., $\Gamma_k(v, V') = \{u \mid u \in V' \text{ and } dist_{(V', E_G(V'))}(u, v) \leq k\}$. The diameter of a subset $V' \subseteq V$, denoted by $diam(V')$, is the maximum pairwise distance in G between a pair of vertices from V'. For a collection $\mathcal{F}$ of subsets $V' \subseteq V$, let $diam(\mathcal{F}) = \max_{V' \in \mathcal{F}}\{diam(V')\}$. Finally, unless specified explicitly, we say that an event happens with high probability if the probability is at least $1 - \frac{1}{n^{\Omega(1)}}$.

Our spanner construction utilizes graph covers. For a graph $G = (V, E)$ and two integers $\kappa, W > 0$, a (κ, W)-*cover* [4,13,19] $\mathscr{C}$ is a collection of not necessarily disjoint subsets (or clusters) $C \subseteq V$ that satisfy the following conditions. (1) $\bigcup_{C \in \mathscr{C}} C = V$. (2) $diam(\mathscr{C}) \leq O(\kappa W)$. (3) The *size* of the cover $s(\mathscr{C}) = \sum_{C \in \mathscr{C}} |C|$ is at most $O(n^{1+1/\kappa})$, and furthermore, every vertex belongs to $O(\text{polylog}(n) \cdot n^{1/\kappa})$ clusters. (4) For every pair of vertices $u, v \in V$ that are at distance at most W one from another, there exists a cluster $C \in \mathscr{C}$ that contains both vertices, along with the shortest path between them.

We remark that each cluster C of the cover contains a special vertex v that is dedicated to serve as the *root* of the cluster C. We will assume that the description of each cluster C contains a description of a BFS spanning tree T of C rooted at v.

There are quite a number of spanner-constructing algorithms that use covers (e.g., [4,13,19]). In fact, for an unweighted graph, the union of the BFS spanning trees of a $(\kappa, 1)$-cover is a κ-spanner. Moreover, it was demonstrated in [19] that covers can serve as a basis for $(1+\epsilon, \beta)$-spanners as well, but in this case they are combined with other tools. (Currently it is not known whether $(1 + \epsilon, \beta)$-spanners can be built using a hierarchy of covers alone.)

The algorithm of [19] for constructing $(1+\epsilon, \beta)$-spanners has a recursive structure. It first constructs a cover on the whole graph. The cover contains clusters of different sizes. The algorithm divides the clusters into two groups according to their sizes (the size of a cluster is the number of vertices in that cluster.): a group $\mathscr{C}^H$ of large clusters and a group $\mathscr{C}^L$ of small clusters. The algorithm will be recursively invoked on the clusters that belong to the small-cluster group $\mathscr{C}^L$. We can assume that, after the recursion returns, a spanner (with smaller estimation error) for each subgraph induced by a cluster in this group will be constructed. What remains then is to deal with the large clusters in the group $\mathscr{C}^H$. For these clusters, The algorithm finds the shortest path between pairs of clusters that are not far away from each other. These paths, together with the BSF trees of the clusters and the spanners for the subgraphs, form the spanner of the whole graph.

Algorithm 1 (RECUR_SPANNER) *The input to the algorithm is a graph $G = (V, E)$ and four parameters κ, ν, D, and Δ, where κ, D and Δ are positive integers and $0 < \nu < 1$.*

1. $\ell \leftarrow \lceil \log_{1/(1-\nu)} \log_\Delta |V| \rceil$.
2. *Construct a (κ, D^ℓ)-cover $\mathscr{C}$ for G. Include the edges of the BFS spanning trees of all the clusters in the spanner. Set $\mathscr{C}^H \leftarrow \{C \in \mathscr{C} \mid |C| \geq |V|^{1-\nu}\}$, $\mathscr{C}^L \leftarrow \mathscr{C} \setminus \mathscr{C}^H$. (We call the clusters from $\mathscr{C}^H$ "large clusters," and the clusters from $\mathscr{C}^L$ "small clusters.")*
3. **Interconnecting subroutine:** *For all pairs of clusters $C_1, C_2 \in \mathscr{C}^H$ s.t. $dist_G(C_1, C_2) \leq D^{\ell+1}$, compute one of the shortest paths between C_1 and C_2 in G. Include this path in the spanner.*
4. *For all clusters $C \in \mathscr{C}^L$, invoke Recur_Spanner($G' = (C, E_G(C)), \kappa, \nu, D, \Delta$).*

We cite from [19] the following theorem on the the size and stretch factor of the constructed spanner:

Theorem 1. [19] *Let G=(V,E) be an n-vertex graph, $\kappa = 1, 2, \ldots, 0 < \nu \leq 1/2 - 1/\kappa, D = 2, 3, \ldots, \Delta = 1, 2, \ldots$. let $\ell = \lceil \log_{1/(1-\nu)} \log_\Delta n \rceil$. Let H be the spanner constructed by Algorithm RECUR_SPANNER invoked on the five-tuple $(G, \kappa, \nu, D, \Delta)$. Then H is a $(1+O(\frac{\kappa\ell}{D}), O(\kappa D^\ell))$-spanner, and the size of H is at most $O((\Delta + D^{\ell+1}) n^{1+\frac{1}{\kappa\nu}})$.*

A detailed proof of the theorem is presented in [19]. To help the readers understand the connection between the covers and the constructed spanner, we give a short proof sketch here.

Proof. (Sketch)

We consider the stretch factor of the spanner H first. Consider a pair of vertices and a shortest path between them in the original graph. We divide the path into segments of length at most $D^{\ell+1}$. We argue that if the spanner can give a $(1 + \epsilon', \beta)$-estimation on the distances of the segments, it can also provide a $(1 + \epsilon, \beta)$-estimation on the whole path, where $\epsilon = \epsilon' + \frac{\beta}{D^{\ell+1}}$. To see this, let Dst_G be the distance between the two vertices in the orginal graph and $Dst_G = a \cdot D^{\ell+1} + b$ for some non-negative integer a and b. (The distance is divided into a segments of length exactly $D^{\ell+1}$ plus one possible segment of length b.) Let Dst_H be the distance between the same pair of vertices in the spanner H. For each segment, the spanner provide a $(1 + \epsilon', \beta)$-approximation. Hence,

$$
\begin{aligned}
Dst_H &\leq a(D^{\ell+1}(1 + \epsilon') + \beta) + b(1 + \epsilon') + \beta \\
&= a \cdot D^{\ell+1}(1 + \epsilon' + \frac{\beta}{D^{\ell+1}}) + b(1 + \epsilon') + \beta \\
&\leq (a \cdot D^{\ell+1} + b)(1 + \epsilon' + \frac{\beta}{D^{\ell+1}}) + \beta \\
&= Dst_G(1 + \epsilon) + \beta \; .
\end{aligned}
$$

Thus, we only need to focus on how the spanner provides estimation for distances up to $D^{\ell+1}$. (This is also the reason that in step 3 of Algorithm 1, only pairs of large clusters that are close to each other are connected.) For two vertices u, w that are at most distance $D^{\ell+1}$ from each other, we consider again some shortest path P_{uw} between them. We can view this path as going from left to right. Let u' (w') be the left-most (right-most) vertex on P_{uw} that is contained in a large cluster. Note that, the part of path from u to u' and from w' to w are contained in subgraphs induced by some small clusters. In step 4 of Algorithm 1, after the recursion returns, we have a spanner (with smaller estimation error) for these subgraphs and we can use it to estimate the length of these two parts. What is left now is the part between u' and w'. Note that both u' and w' are contained in large cluster(s). Their distance can then be approximated by the shortest distance between the two clusters plus the diameters of the two clusters. The possible overestimation caused by the diameters of the large cluster contributes to the additive part of the estimation error.

Now we consider the size of the spanner. The spanner consists of the BFS trees of the clusters and the paths connecting the large clusters. Because the size of the cover is small, so is the number of edges in the union of the BFS trees of the clusters. Furthermore, the algorithm only connects large clusters that are close one to another. As there are not many large clusters (otherwise, the size of the cover will be too large), the number of edges added by the interconnecting paths is not too big. The size of the spanner is controlled by the parameters that determine the size of the cover, the number of the large clusters and the maximum length of the inter-cluster paths. $\square$

We observe that within each cluster, the cover construction and the interconnecting subroutine are local to the cluster. That is, these processes in one cluster are independent of the analogous processes in other clusters of the same level. Furthermore, the process of interconnecting the clusters of $\mathscr{C}^H$ is independent of the cover constructions within clusters of $\mathscr{C}^L$. Thus, they can, in principle, be carried out in parallel.

Our distributed implementation of the algorithm for constructing $(1+\epsilon,\beta)$-spanners is different from that of [19] in two ways. First, the algorithm of [19] traverses a spanning tree of the entire graph, and performs the local subroutines for constructing neighborhood covers and interconnecting large clusters *sequentially*. Our algorithm avoids the traversal of the entire graph and performs these local subroutines *in parallel*. Additionally, the algorithm of [19] uses the algorithm of [4] for constructing neighborhood covers, and this algorithm by itself requires a super-linear time. Our algorithm, instead, uses a far more efficient algorithm due to [20] for constructing neighborhood covers. The latter algorithm has running time of $O(n^\rho)$ for an arbitrarily small $\rho > 0$. These two modifications enable us to come up with a drastically improved distributed algorithm for constructing $(1+\epsilon,\beta)$-spanners. We note that while replacing the subroutine of [4] for constructing neighborhood covers by an analogous subroutine from [20] is done in the straightforward way, the parallel execution of different local subroutines is technically much more involved, and constitutes, essentially, the heart of the current paper.

We next briefly describe the construction [13,20] of (κ, W)-covers which we use as a subroutine in our algorithm. This construction builds a (κ, W)-cover in κ phases. A vertex v in graph G is called *covered* if there is a cluster $C \in \mathscr{C}$ such that $\Gamma_W(v, V) \subseteq C$. Let U_i be the set of uncovered vertices at phase i. At the beginning, $U_1 = V$. At each phase i, a subset of vertices is covered and removed from U_i.

Algorithm 2 (COVER) *The input to the algorithm is a graph $G = (V, E)$, and two positive integer parameters κ and W.*

1. $U_1 \leftarrow V$.
2. *On each phase $i = 1, 2, \ldots, \kappa$:*
   (a) *Include each vertex $v \in U_i$ independently at random with probability $p_i = min\{1, \frac{n^{i/\kappa}}{n} \cdot \log n\}$ into the set S_i of phase i.*
   (b) *Each vertex $s \in S_i$ constructs a cluster by growing a BFS tree of depth $d_{i-1} = 2((\kappa - i) + 1)W$ in the graph $(U, E(U))$. We call s the* center *of the cluster and the set $\Gamma_{2(\kappa-i)W}(s, U)$ the* core set *of the cluster $\Gamma_{2((\kappa-i)+1)W}(s, U)$.*
   (c) *Let R_i be the union of the core sets of the clusters constructed on step (b).*
   Set $U_{i+1} \leftarrow U_i \setminus R_i$.

Theorem 2. [20] *Given an unweighted undirected n-vertex graph, Algorithm 2 constructs a (κ, W)-cover such that, with probability at least $1 - \frac{1}{n^{\Omega(1)}}$, every vertex is included in at most $O(\kappa n^{1/\kappa} \cdot \log n)$ clusters of the cover. The construction requires $O(\kappa^2 n^{1/\kappa} W \cdot \log n)$ rounds of distributed computation.*

We now present our distributed implementation of Algorithm RECUR_SPANNER, that uses the above cover construction as a subroutine. Given a cluster C, let $\mathscr{C}(C)$ be the cover constructed for the graph $(C, E_G(C))$. For a cluster $C' \in \mathscr{C}(C)$, we define $Parent(C') = C$.

An execution of the algorithm can be divided into ℓ stages (levels). The original graph is viewed as a cluster of level 0. The algorithm starts level 1 by constructing a cover for this cluster. Recall that a cover is also a collection of clusters. The clusters of $\cup\mathscr{C}(C)$, where the union is over all the clusters C of level 0, are called *clusters of level 1*, and we denote the set of those clusters by $\mathscr{C}_1$. If a cluster $C \in \mathscr{C}_1$ satisfies $|C| \geq |Parent(C)|^{1-\nu}$, we say that C is a *large cluster* on level 1. Otherwise, we say that C is a *small cluster* on level 1. We denote by $\mathscr{C}_1^H$ the set of large clusters on level 1 and $\mathscr{C}_1^L$ the set of small clusters on level 1. Note that the cover-construction subroutine (Algorithm 2) builds a BFS spanning tree for each clusters in the cover. Our algorithm includes all the BFS spanning trees into the spanner and then goes on to make interconnections between all pairs of clusters in $\mathscr{C}_1^H$ that are close to each other. After these interconnections are completed, the algorithm enters level 2. For each cluster in $\mathscr{C}_1^L$, it constructs a cover. We call the clusters in each of these covers the *clusters of level 2*. The union of all the level 2 clusters is denoted by $\mathscr{C}_2$. If a cluster $C \in \mathscr{C}_2$ satisfies $|C| \geq |Parent(C)|^{1-\nu}$, we say that C is a *large cluster* on level 2. Otherwise, we say that C is a *small cluster* on level 2. Again, we denote by $\mathscr{C}_2^H$ the set of large clusters on level 2 and $\mathscr{C}_2^L$ the set of small clusters on level 2. The BFS spanning trees of all the clusters in $\mathscr{C}_2$ are included into the spanner and all the close pairs of clusters in $\mathscr{C}_2^H$ get interconnected by the algorithm. The algorithm proceeds in a similar fashion on levels 3 and above. That is, on level i, the algorithm constructs covers for each small cluster in $\mathscr{C}_{i-1}^L$, and interconnects all the close pairs of large clusters in $\mathscr{C}_i^H$. Similarly, we denote by $\mathscr{C}_i$ the collection of all the clusters in the covers constructed on level i, by $\mathscr{C}_i^H$ the set of large clusters of $\mathscr{C}_i$, and $\mathscr{C}_i^L$ the set of small clusters of $\mathscr{C}_i$. After level ℓ, each of the small clusters of level ℓ contains very few vertices and the algorithm can include in the spanner all the edges induced by these clusters. A formal description of the detailed algorithm is given below (Algorithm 3).

See Fig. 1 for an example of covers and clusters constructed by the algorithm. The circles in the figure represent the clusters. $\mathscr{C}_1 = \{C_1, C_2, C_3\}$, $\mathscr{C}_1^H = \{C_1\}$ and $\mathscr{C}_1^L = \{C_2, C_3\}$. Note that for each cluster in $\mathscr{C}_1^L$, a cover is constructed. The union of the clusters in these covers forms $\mathscr{C}_2$, i.e., $\mathscr{C}_2 = \{C_4, C_5, C_6, C_7, C_8, C_9, C_{10}\}$. The large clusters in $\mathscr{C}_2$ form $\mathscr{C}_2^H = \{C_4, C_5, C_8, C_9\}$ and the small clusters in $\mathscr{C}_2$ form $\mathscr{C}_2^L = \{C_6, C_7, C_{10}\}$. Also note that a pair of close large clusters C_8 and C_9 are interconnected by a shortest path between them.

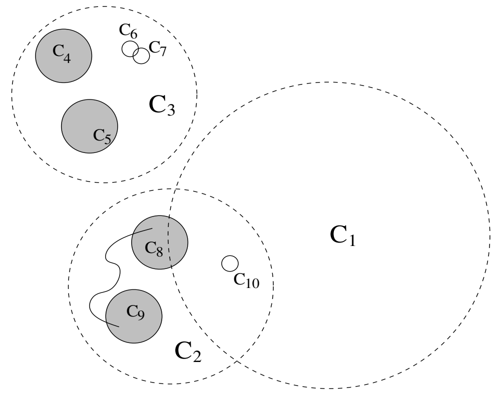

Fig. 1. Example of Clusters in Covers.

Algorithm 3 (SPANNER) *The input to the algorithm is a graph $G = (V, E)$ on n vertices, and four parameters κ, ν, D, and Δ, where κ, D and Δ are positive integers and $0 < \nu < 1$.*

1. $\mathscr{C}_0^L \leftarrow \{V\}$, $\mathscr{C}_0^H = \phi$.
2. *For level* $i = 1, 2, \ldots, \ell = \lceil \log_{1/(1-\nu)} \log_\Delta n \rceil$ *do*
   (a) ***Cover Construction****:*
   For all clusters $C \in \mathscr{C}_{i-1}^L$, in parallel, construct (κ, D^ℓ)-covers using Algorithm 2. (Invoking Algorithm 2 with parameters κ and $W = D^\ell$.) Include the edges of the BFS spanning trees of all the clusters in the spanner. Set $\mathscr{C}_i \leftarrow \bigcup_{C \in \mathscr{C}_{i-1}^L} \mathscr{C}(C)$, $\mathscr{C}_i^H \leftarrow \{C \in \mathscr{C}_i \mid |C| \geq |Parent(C)|^{(1-\nu)}\}$, $\mathscr{C}_i^L \leftarrow \mathscr{C}_i \setminus \mathscr{C}_i^H$.
   (b) ***Interconnection****:*
   For all clusters $C' \in \mathscr{C}_i^H$, in parallel, construct BFS trees in $G(C)$, where $C = Parent(C')$. For each cluster C', the BFS tree is rooted at the center of the cluster and the depth of the BFS tree is $2D^h + D^{h+1}$ where $h = \lceil \log_{1/(1-\nu)} \log_\Delta |Parent(C')| \rceil$. For all

the clusters C'' whose center vertex is on the BFS tree, if $C'' \in \mathscr{C}_i^H$ and $Parent(C'') = Parent(C')$, add to the spanner the shortest path between the center of C' and the center of C''.

3. *Add to the spanner all the edges of the set $\bigcup_{C \in \mathscr{C}_{\ell+1}} E_G(C)$.*

Note that the above algorithm is a synchronous protocol. By constructing and using a synchronizer [5], this protocol can be converted into an asynchronous protocol.

3 Analysis of the Time and Message Complexity

In this section we analyze the time and message complexity of Algorithm 3. In the distributed model, carrying out the cover constructions in parallel for all clusters of a certain family (as it is done in step (2a) of Algorithm 3) does not necessarily mean that the running time is equal to the running time of constructing one single cover. If all the cover constructions utilize the same edges, the running time of step (2a) may be no better than the time required for constructing these covers sequentially because of congestions. We show that this is not the case in our algorithm. Specifically, we show that each edge is utilized only by a small number of subroutines that construct covers. This is also true regarding the interconnecting subroutines on step (2b).

Given a cover $\mathscr{C}$, let $MS(\mathscr{C}) = max_{C \in \mathscr{C}}\{|C|\}$. The number of levels in the algorithm is $\ell = \lceil \log_{1/(1-\nu)} \log_\Delta n \rceil$. Throughout the analysis assume that κ, ν, ℓ are constant, i.e., independent of n.

Lemma 1. *For a vertex v, and an index $i = 1, 2, \ldots, \ell$, with high probability, the number of clusters $C \in \mathscr{C}_i$ that contain v is at most $O(n^{\frac{1}{\kappa\nu}})$.*

Proof. By Theorem 2, with high probability, in a cover constructed for a cluster C, a vertex is contained in at most $O(\log |C| \cdot |C|^{1/\kappa})$ clusters. Also note that $MS(\mathscr{C}_i^L) \leq n^{(1-\nu)^i}$.

Let $\mathcal{M}_i(v) = \{C \in \mathscr{C}_i \mid v \in C\}$. We have:

$$\begin{aligned}
|\mathcal{M}_{i+1}(v)| &\leq \sum_{C \in \mathcal{M}_i(v)} O(\log |C| \cdot |C|^{1/\kappa}) \\
&\leq |\mathcal{M}_i(v)| \cdot O(\log(MS(\mathscr{C}_i^L)) \cdot MS(\mathscr{C}_i^L)^{1/\kappa}) \\
&\leq |\mathcal{M}_i(v)| \cdot O(\log n^{(1-\nu)^i} \cdot n^{\frac{(1-\nu)^i}{\kappa}}).
\end{aligned}$$

Given that the number of levels is $\ell = O(1)$, $|\mathcal{M}_i(v)| \leq O(\log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}) \leq O(n^{\frac{1}{\kappa\nu}})$. $\square$

Lemma 2. *With high probability, all the subroutines for constructing covers require altogether at most $O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1})$ time and $O(|E| \cdot n^{\frac{1}{\kappa\nu}})$ communication.*

Proof. By Lemma 1, for a fixed vertex v, and index $i = 1, 2, \ldots, \ell$, the vertex is contained in at most $O(n^{\frac{1}{\kappa\nu}})$ clusters of level i. Hence, on level i the vertex is explored at most $O(n^{\frac{1}{\kappa\nu}})$ times. Hence, for a fixed edge e, the edge is explored at most $O(n^{\frac{1}{\kappa\nu}})$ times on level i. On level i, each BFS exploration for constructing a cover has depth at most

$$\begin{aligned}
&2\kappa \cdot D^{\log_{1/(1-\nu)} \log_\Delta MS(\mathscr{C}_{i-1}^L)} \\
&\leq 2\kappa \cdot D^{\log_{1/(1-\nu)} \log_\Delta n^{(1-\nu)^{i-1}}} = 2\kappa \cdot D^{\ell+1-i}.
\end{aligned}$$

Because each cover construction consists of $\kappa = O(1)$ phases, the overall time required for constructing all the covers on level i is at most $O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1-i})$. Hence, the overall time is: $\sum_{i=1}^{\ell} O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1-i}) \leq O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1})$.

Because each edge is explored at most $O(n^{\frac{1}{\kappa\nu}})$ times, and because there are $\kappa = O(1)$ phases and $\ell = O(1)$ levels, the overall number of messages that are sent for constructing all the covers is at most $O(|E| \cdot n^{\frac{1}{\kappa\nu}})$. □

Lemma 3. *With high probability, for each level $i = 1, 2, \ldots, \ell$, and a vertex $v \in V$, the vertex is explored by $O(n^{\frac{1}{\kappa\nu}+\nu})$ BFS explorations that are initiated by the interconnecting subroutines.*

Proof. Consider a cluster $C \in \mathcal{M}_{i-1}(v) \cap \mathscr{C}_{i-1}^L$. The size of the cover constructed for the cluster C is at most $O(\log|C| \cdot |C|^{1+1/\kappa})$. The large clusters in this cover have size at least $|C|^{(1-\nu)}$. Hence, the number of such clusters in the cover for C is at most $O(\frac{\log|C| \cdot |C|^{1+1/\kappa}}{|C|^{(1-\nu)}}) = O(\log|C| \cdot |C|^{\frac{1}{\kappa}+\nu})$. In each cluster $C \in \mathcal{M}_{i-1}(v) \cap \mathscr{C}_{i-1}^L$ of level $(i-1)$, each of the $O(\log|C| \cdot |C|^{\frac{1}{\kappa}+\nu})$ large clusters of level i may explore the vertex v. Hence, the overall number of BFS explorations that may visit the vertex v is at most

$$\begin{aligned}
&\sum_{C \in \mathcal{M}_{i-1}(v)} O(\log|C| \cdot |C|^{\frac{1}{\kappa}+\nu}) \\
&\leq |\mathcal{M}_{i-1}(v)| \cdot O(\log MS(\mathscr{C}_{i-1}^L) \cdot MS(\mathscr{C}_{i-1}^L)^{\frac{1}{\kappa}+\nu}) \\
&\leq O(\log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}} \cdot n^{(\frac{1}{\kappa}+\nu)\cdot(1-\nu)^{i-1}}) \\
&\leq O(n^{\frac{1}{\kappa\nu}+\nu}).
\end{aligned}$$

For the last inequality, note that $n^{\frac{(1-\nu)^{i-1}}{\kappa\nu} - \frac{(1-\nu)^{i-1}}{\kappa}} = n^{\frac{(1-\nu)^i}{\kappa\nu}} \geq \log^{O(1)} n$ for a sufficiently large n, and constant numbers $\kappa = 1, 2, \ldots$ and $\nu < 1$. □

Lemma 4. *With high probability, for an index $i = 1, 2, \ldots, \ell$, the overall time and communication complexities of all the interconnecting subroutines is $O(D^{\ell+2} \cdot n^{\frac{1}{\kappa\nu}+\nu})$ and $O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu})$, respectively.*

Proof. On level i, the depth of the BFS explorations that are required for interconnecting the large clusters is at most

$$\begin{aligned}
&3D^{\log_{1/(1-\nu)} \log_\Delta MS(\mathscr{C}_{i-1}^L)+1} \\
&\leq 3D^{\log_{1/(1-\nu)} \log_\Delta n^{(1-\nu)^{i-1}}+1} = 3D^{\ell+2-i}.
\end{aligned}$$

By Lemma 3, a vertex participates in at most $O(n^{\frac{1}{\kappa\nu}+\nu})$ interconnecting processes. Hence, the overall running time of all the interconnecting subroutines on level i is $O(n^{\frac{1}{\kappa\nu}+\nu} \cdot D^{\ell+2-i})$. Adding up the ℓ levels, we have $\sum_{i=1}^{\ell} O(n^{\frac{1}{\kappa\nu}+\nu} \cdot D^{\ell+2-i}) \leq O(D^{\ell+2} \cdot n^{\frac{1}{\kappa\nu}+\nu})$.

For an upper bound on communication complexity on level i, note that each cluster C in $\mathscr{C}_i^H$ initiates a BFS exploration in the cluster $Parent(C) \in \mathscr{C}_{i-1}^L$. Following the analysis in Lemma 3, the number of large clusters in the cover $\mathscr{C}(Parent(C))$ is at most $O(\log|Parent(C)| \cdot |Parent(C)|^{\frac{1}{\kappa}+\nu})$. Because for every index $i = 1, 2, \ldots, \ell$, $|\mathcal{M}_i(v)| \leq O\left(\log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}\right)$, it follows that, $e(\mathscr{C}_i) = \sum_{C \in \mathscr{C}_i} |E_G(C)|$ is at most $O\left(|E| \cdot \log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}\right)$.

To summarize,

$$\begin{aligned}
&\sum_{C \in \mathscr{C}_{i-1}^L} O(\log|C| \cdot |C|^{\frac{1}{\kappa}+\nu}) \cdot e(C) \\
&\leq O\left(\log(MS(\mathscr{C}_{i-1}^L)) \cdot MS(\mathscr{C}_{i-1}^L)^{\frac{1}{\kappa}+\nu}\right) \cdot \sum_{C \in \mathscr{C}_{i-1}^L} e(C) \\
&\leq O\left(\log^{O(1)} n \cdot n^{(\frac{1}{\kappa}+\nu)(1-\nu)^{i-1}}\right) \cdot e(\mathscr{C}_{i-1})
\end{aligned}$$

$\leq O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu})$.

Because there are ℓ levels, the overall communication complexity is at most $O(\ell \cdot |E| \cdot n^{\frac{1}{\kappa\nu}+\nu}) = O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu})$. □

Now we are ready to prove the main result of this section.

Theorem 3. *Given an unweighted undirected graph on n vertices and constants $0 < \rho, \delta, \epsilon < 1$, such that $\delta/2 + 1/3 > \rho > \delta/2$, there is a distributed algorithm that, with probability at least $1 - \frac{1}{n^{\Omega(1)}}$, constructs a $(1+\epsilon, \beta)$-spanner of size $O(n^{1+\delta})$ for the graph, where $\beta = \beta(\rho, \delta, \epsilon) = O(1)$. The running time and the communication of the algorithm are $O(n^\rho)$ and $O(|E|n^\rho)$, respectively.*

Proof. Set $\Delta = n^{\delta/2}$, $\frac{1}{\kappa\nu} = \delta/2$, $\frac{1}{\kappa\nu} + \nu = \rho$. This gives $\nu = \rho - \frac{\delta}{2} > 0, \nu = O(1)$, $\kappa = \frac{2}{(\rho-\delta/2)\delta} = O(1)$, and $\ell = \log_{1/(1-\nu)} \log_\Delta n = \log_{1/(1-\nu)} \frac{2}{\delta} = O(1)$, satisfying the requirement that κ, ν and ℓ are all constants.

Also set $O(\frac{\kappa\ell}{D}) = \epsilon$. By Theorem 1, $(1+\epsilon)$ is the multiplicative stretch factor of the constructed spanner. This gives $D = O(\frac{\kappa\ell}{\epsilon}) = O(1)$. Hence, $\beta = O(\kappa D^\ell) = O(1)$ is the additive term of the spanner. By Theorems 2 and 1, the size of our spanner is $O\left((\Delta + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}}\right)$. For a sufficiently large n, $\Delta = n^{\delta/2} > D^{\ell+1}$, hence the spanner size is $O(n^{1+\delta})$. By Lemma 2 and 4, the running time of the algorithm is $O(D^{\ell+2} \cdot n^{\frac{1}{\kappa\nu}+\nu}) = O(n^\rho)$, and its communication complexity is $O(|E| \cdot n^\rho)$. □

3.1 Parallel implementation

Observe that the basic operation in both Algorithm 2 and Algorithm 3 is the construction of constant-depth BFS trees on the unweighted graph. Hence, we can use the straightforward parallelization of the BFS search for this operation [17, 13] and implement Algorithm 3 in the parallel computing models. We now briefly analyze the complexity of such an implementation in a EREW PRAM model.

The overall number of edges needed to be explored determines the amount of work performed by the parallel implementation. Note that in both the cover construction and the interconnection processes, an edge may be explored by multiple BFS tree constructions. The algorithm runs in $\ell = O(1)$ iterations (levels). It invokes cover construction and interconnection at each level. For cover construction, by Lemma 1 and the fact that there are $\kappa = O(1)$ phases in the construction, an edge may be explored at most $O(n^{\frac{1}{\kappa\nu}})$ times. For interconnection, by Lemma 3, an edge may be explored at most $O(n^{\frac{1}{\kappa\nu}+\nu})$ times. Hence, the overall number of edges being explored is at most $O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu}) = O(|E| \cdot n^\rho)$.

For a graph $G(V, E)$, recall that $\Gamma_k(u, V)$ is the k-neighborhood of the vertex u, i.e., the set of vertices of distance at most k from u. Also recall that $E_G(\Gamma_k(u, V))$ is the set of edges induced by the set of vertices in the neighborhood $\Gamma_k(u, V)$. The following proposition is from [13].

Proposition 1. [13] *For a graph $G(V, E)$, a set of integers $k_1, \ldots, k_r$ and a set of vertices $s_1, \ldots, s_r$, the computation* *of $\Gamma_{k_i}(s_i, V)$ for $i = 1, \ldots, r$ can be performed using p processors in time*

$$O\left((\max_i k_i \log n) + \sum_{i=1}^{r} |E_G(\Gamma_{k_i}(s_i, V))|(\log n)/p\right)$$

Note that our parallel implementation runs in constant number of iterations. The number of phases in each cover construction is also constant and so is the depth of each BFS exploration. Therefore, using the BFS search algorithm of [13], the parallel implementation of Algorithm 3 requires $O(\log n + (|E| \cdot n^{\rho} \log n)/p)$ running time using p processors in the EREW PRAM model. Particularly, when the number of processors, p, is at least $|E| \cdot n^{\rho}$, the running time of the algorithm is $O(\log n)$.

4 Adaptation to the Streaming Model

In this section we adapt Algorithm 3 to the streaming model and devise an algorithm for computing all-pairs-almost-shortest-paths in this model.

In [8,23], it was shown that in the streaming model, finding the common neighborhood (distance 1 neighborhood) of two vertices requires $\Omega(n)$ bits of space in constant number of passes. We extend this result to show the following:

Theorem 4. *Any streaming algorithm that provides $o(n)$-approximate estimate of the distance between a fixed pair of vertices in constant number of passes must use $\Omega(n)$ bits of space.*

Proof. Consider an n-vertex graph and two fixed vertices x and y of the graph. The graph also includes a path

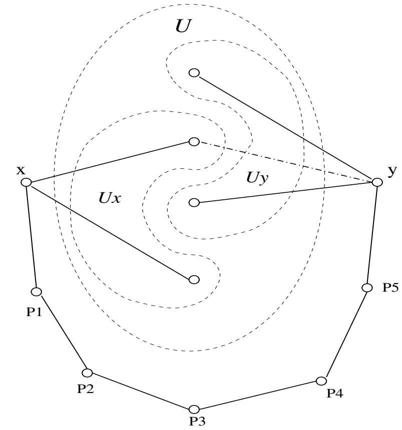

Fig. 2. The distance between x and y is 6 if there is no common neighborhood of x and y. Otherwise (if the dash-dotted edge exists), the distance is 2.

$(x, p_1), (p_1, p_2), \ldots, (p_t, y)$, and another set of vertices U, where $t = \Theta(n)$ and $|U| = \Theta(n)$. There are edges between x and some of the vertices in U. Let U_x denote the set of neighbors of x in U. There are also edges between y and some of the vertices in U. Let U_y denote the set of neighbors of y in U. (See Fig. 2 for an example.) Note that any streaming algorithm that can $o(n)$-approximate the distance between x and y is also able to decide whether the two sets U_x and U_y are disjoint, i.e., whether there is a common neighborhood of x and y. Since the set-disjointness problem has communication complexity of $\Omega(n)$ [28], the problem of deciding whether a pair of vertices in a graph has a common neighbor, in the streaming model using a constant number of passes, has space complexity $\Omega(n)$. (The lower bound on the space complexity of the problem of deciding whether two vertices have a common neighbor is due to Buchsbaum *et al.* [8].) □

Leaving space limitations aside, it is easy to see that many distributed algorithms with time complexity T translate directly into streaming algorithms that use T passes. For example, a straightforward streaming adaptation of a synchronous distributed algorithm for constructing a BFS tree would be the following: on each pass over the input stream, the BFS tree grows one more level. An exploration of d levels would result in d passes over the input stream. On the other hand, there are cases when the running time of a synchronous algorithm may not translate directly to the number of passes of the streaming adaptation. In the example of BFS tree, if two BFS trees are being constructed in parallel, some edges may be explored by both constructions, resulting in congestions that may increase the running time of the distributed algorithm. On the other hand, for a streaming algorithm, both explorations of the same edge can be done using only one pass over the stream.

We adapt Algorithm 2 for constructing covers to the streaming model. The streaming adaptation proceeds in κ phases. On each phase i, the algorithm passes through the input stream d_{i-1} times to build the BFS trees $\tau(v)$ of depth d_{i-1} for each selected vertex $v \in S_i$. The cluster and its core set can be computed during the construction of these BFS trees. Note that for any i, $d_{i-1} \leq 2\kappa W$. Hence,

Lemma 5. *With high probability, the streaming adaptation of Algorithm 2 constructs a (κ, W)-cover using $2\kappa^2 W$ passes over the input stream.*

Now, we briefly describe the adaptation of Algorithm 3 to the streaming model. The adapted algorithm is recursive, and the recursion has ℓ levels. On level i, a cover is constructed for each of the small clusters in $\mathscr{C}^L_{i-1}$ using the streaming algorithm for constructing cover described above. Because the processes of building BFS trees for constructing cover are independent, they can be carried out in parallel. That is, when the algorithm encounters an edge in the input stream, it examines its two endpoints. For each of the clusters in $\mathscr{C}^L_{i-1}$ that contains both endpoints, for each of the BFS-tree construction in such cluster that has reached one of the endpoints, the algorithm checks whether the edge would help to extend the BFS tree. If so, the edge would be added to that BFS tree. After the construction of the covers is completed, the algorithm makes interconnections between close large clusters of each cover. Again, the constructions of the BFS trees that are invoked by different interconnection subroutines are independent and can be performed in parallel.

Lemma 6. *With high probability, the streaming adaptation of Algorithm 3 requires $2\kappa^2 D^{\ell+1} + 3D^{\ell+2}$ passes over the input stream.*

Proof. Note that on level i, for cover construction, the value of W is bounded by

$$D^{\log_{1/(1-\nu)} \log_\Delta MS(\mathscr{C}^L_{i-1})}$$

$$\leq D^{\log_{1/(1-\nu)} \log_\Delta n^{(1-\nu)^{i-1}}} \leq D^{\ell+1-i}.$$

By Lemma 5, the number of passes that are required for constructing a cover on level i is at most $2\kappa^2 D^{\ell+1-i}$.

On level i, the algorithm also invokes interconnecting subroutines. The depth of the BFS trees that are required for interconnection on recursion level i is at most $3D^{\log_{1/(1-\nu)}\log_\Delta MS(\mathscr{C}^L_{i-1})+1} \leq 3D^{\ell-i+2}$.

The overall number of passes is at most $\sum_{i=1}^{\ell}(2\kappa^2 D^{\ell+1-i} + 3D^{\ell-i+2}) \leq 2\kappa^2 D^{\ell+1} + 3D^{\ell+2}$. $\square$

Lemma 7. *The space complexity of the streaming adaptation of Algorithm 3 is* $O\left(\log n \cdot (\Delta + D^{\ell+1}) \cdot n^{1+\frac{1}{\kappa\nu}}\right)$.

Proof. The algorithm needs storage space for the following: (1) The algorithm stores the edges of the spanner. (2) The covers constructed by the algorithm are collections of clusters. The algorithm includes a vertex in a cluster by labeling the vertex with the ID of the cluster. Hence, for each vertex, the algorithm stores the IDs of the clusters to which the vertex belongs. (3) On each level i, the large clusters of $\mathscr{C}^H_i$ construct BFS trees for interconnection. Each BFS tree is constructed by labeling the vertices layer by layer using the ID of the initiating cluster. Hence, for each vertex, the algorithm also stores the IDs of clusters $C \in \mathscr{C}^H_i$ whose BFS exploration has visited/explored the vertex.

By Theorem 1 and Lemma 1, items (1) and (2) require $O((\Delta + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}})$ and $O(n^{\frac{1}{\kappa\nu}})$ cells of memory, respectively. By Lemma 3, item (3) requires $O(n^{\frac{1}{\kappa\nu}+\nu})$ cells of memory for each level. We observe that once the interconnections are made, the algorithm will no longer need the BFS trees constructed for the interconnections. The space used to store these BFS trees on level i can be reused on level $i+1$. Hence, the overall amount of memory cells required by item (3) is at most $O(n^{\frac{1}{\kappa\nu}+\nu})$. Note that all the above quantities are given in terms of the number of edges and IDs. The space in terms of the number of bits is greater by at most a factor of $\log n$.

Hence, the overall space complexity is at most:

$$\begin{aligned}&O\left((\Delta + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}} \cdot \log n\right)\\ &+ O\left(n^{\frac{1}{\kappa\nu}} \cdot \log n\right) + O\left(n^{\frac{1}{\kappa\nu}+\nu} \cdot \log n\right)\\ &= O\left(\log n \cdot (\Delta + D^{\ell+1}) \cdot n^{1+\frac{1}{\kappa\nu}}\right). \quad \square\end{aligned}$$

Lemma 8. *With high probability, the streaming adaptation of Algorithm 3 processes each edge using at most* $O(n^{\frac{1}{\kappa\nu}+\nu})$ *time.*

Proof. For a fixed index $i = 1, 2, \ldots, \ell$, on level i, during the construction of the cover, the algorithm may need to examine the vertex v for each of the clusters of level i that contain the vertex when it encounters the edge (u, v) in the stream. Let $T_{cover}(v)$ be the processing time for this purpose. By Lemma 1, $T_{cover}(v) = O(n^{\frac{1}{\kappa\nu}})$.

During interconnection, the algorithm may also need to examine the vertex v for each cluster $C \in \mathscr{C}^H_i$ whose BFS exploration visits v. Let $T_{interconnect}(v)$ be this processing time. By Lemma 3, $T_{interconnect}(v) = O(n^{\frac{1}{\kappa\nu}+\nu})$.

The overall time that is required to process the edge (u, v) is $2 \cdot (T_{cover}(v) + T_{interconnect}(v)) = O(n^{\frac{1}{\kappa\nu}+\nu})$. $\square$

To summarize,

Theorem 5. *Given an unweighted undirected graph on n vertices, presented as a stream of edges, and constants $0 < \rho, \delta, \epsilon < 1$, such that $\delta/2 + 1/3 > \rho > \delta/2$, there is a streaming algorithm that, with probability $1 - \frac{1}{n^{\Omega(1)}}$, constructs a $(1+\epsilon, \beta)$-spanner of size $O(n^{1+\delta})$. The algorithm accesses the stream sequentially in $O(1)$ passes, uses $O(n^{1+\delta} \cdot \log n)$ bits of space and processes each edge of the stream in $O(n^{\rho})$ time.*

Note that once the spanner is computed, the algorithm is able of computing all-pairs-almost-shortest-paths and distances in the graph by computing the *exactly* shortest paths and distances on the spanner using the same space. Observe that for a pair of vertices, $u, v \in V$, the path $P_{u,v}$ that is computed by the algorithm satisfies the inequality $|P_{u,v}| \leq (1+\epsilon)d_G(u,v) + \beta$. Note also that this computation of the shortest paths on the spanner requires no additional passes through the input, and also, no additional space (for the latter we assume that once computed, the paths are immediately output by the algorithm and are not stored; obviously, any algorithm that stores estimates of distances for all pairs of vertices requires $\Omega(n^2)$ space). To summarize,

Corollary 1. *Given an unweighted undirected graph on n vertices, presented as a stream of edges, and constants $0 < \rho, \delta, \epsilon < 1$, such that $\delta/2 + 1/3 > \rho > \delta/2$, there is a streaming algorithm that, with probability $1 - \frac{1}{n^{\Omega(1)}}$, computes all-pairs-almost-shortest-paths in the graph with error terms of $(1+\epsilon, \beta)$. The algorithm accesses the stream sequentially in $O(1)$ passes, uses $O(n^{1+\delta} \cdot \log n)$ bits of space and processes each edge in the stream in $O(n^{\rho})$ time.*

5 Conclusion

We devised a distributed randomized algorithm that improves the distributed algorithm of [19]. Except for the need for randomization, our algorithm drastically reduces the running time at no other cost. Applying our algorithm leads to more efficient distance-approximation algorithms in the distributed setting. For example the running time of the distributed algorithm for the s-source almost-shortest-path problems in [19] can be improved using our algorithm. The adaptation of our algorithm to the streaming model provides a $(1+\epsilon, \beta)$-approximation for all-pair-shortest-paths problem in this model. The algorithm uses only a constant number of passes. We remark that the only previously existing streaming algorithm [23] for computing all-pairs-almost-shortest-paths in the streaming model has a much higher multiplicative stretch factor of κ and a prohibitively large edge-processing time. We also show that our algorithm can be adapted to the EREW PRAM model of parallel computation. This adaptation is the first known parallel algorithm for constructing sparse $(1+\epsilon, \beta)$-spanners.

Acknowledgements. The authors are very grateful to Joan Feigenbaum for inspiring and stimulating discussions, helpful comments, and remarks. The authors also wish to thank an anonymous referee of PODC'04 for helpful comments and remarks.

References

1. D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999.
2. N. Alon, Z. Galil, and O. Margalit. On the exponet of the all pairs shortest path problem. In *Proc. 32st IEEE Symposium on Foundations of Computer Science*, pages 569–575, 1991.
3. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.
4. B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998.
5. B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks. Adapting to asynchronous dynamic networks. In *Proc. 24th ACM Symposium on Theory of Computing*, pages 557–570, 1992.
6. B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 514–522, 1990.
7. B. Awerbuch and D. Peleg. Routing with polynomial communication-space tradeoff. *SIAM Journal on Discrete Mathematics*, 5:151–162, 1992.
8. A.L. Buchsbaum, R. Giancarlo and J.R. Westbrook. On finding common neighborhoods in massive graphs. *Theoretical Computer Science*, 299 (1-3):707–718, 2003.
9. S. Baswana and S. Sen. A simple linear time algorithm for computing a (2k-1)-spanner of $O(n^{1+1/k})$ size in weighted graphs. In *Proc. International Colloq. on Automata, Languages and Computing*, pages 284–296, 2003.
10. S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $o(n^2 \log n)$ time. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, 2004.
11. B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. In *Proc 8th ACM Symposium on Computational Geometry*, pages 192–201, 1992.
12. M. Charikar, L. O'Callaghan, and R. Panigrahy. Better streaming algorithms for clustering problems. In *Proc. 35th ACM Symposium on Theory of Computing*, pages 30–39, 2003.
13. E. Cohen. Fast algorithms for t-spanners and stretch-t paths. In *Proc. 34th IEEE Symposium on Foundation of Computer Science*, pages 648–658, 1993.
14. E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest-paths. In *Proc. 26th ACM Symposium on Theory of Computing*, pages 16–26, 1994.
15. E. Cohen and U. Zwick. All pairs small stretch paths. *Journal of Algorithms*, 38:335–353, 2001.
16. D. Coppersmith and R. Kumar. An improved data stream algorithm for frequency moments. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, 2004.
17. R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17:128–142, 1988.
18. D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.
19. M. Elkin. Computing almost shortest paths. In *Proc. 20th ACM Symposium on Principles of Distributed Computing*, pages 53–62, 2001.

20. M. Elkin. A fast distributed protocol for constructing the minimum spanning tree. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 352–361, 2004.

21. M. Elkin and D. Peleg. $(1+\epsilon, \beta)$-spanner constructions for general graphs. In *Proc. 33rd Annual ACM Symp. on Theory of Computing*, pages 173–182, 2001.

22. M. Elkin and J. Zhang. Efficient algorithms for constructing $(1+\epsilon, beta)$-spanners in the distributed and streaming models. In *Proc. of 23rd Annual SIGACT-SIGOPS Symp. on Principles of Distributed Computing, PODC*, pp. 160-168, St. John's, Newfoundland, Canada, 2004.

23. J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. in *Proc. 31st International Colloquium on Automata, Languages and Programming*, LNCS Vol. 3142, pages 531-543, 2004.

24. J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph Distances in the Streaming Model: The Value of Space. in *Proceedings of 16th ACM-SIAM Symposium on Discrete Algorithms* pages 745-754, 2005.

25. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L^1 difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, 2002.

26. Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243–254, 1997.

27. S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. 41th IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

28. M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *Technical Report 1998-001, DEC Systems Research Center*, 1998.

29. P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proc. 41th IEEE Symposium on Foundations of Computer Science*, pages 189–197, 2000.

30. B. Kalyanasundaram and G. Schnitger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Math.*, 5:545–557, 1990.

31. S. Muthukrishnan. Data streams: Algorithms and applications. 2003. Available at http://athos.rutgers.edu/$\sim$muthu/stream-1-1.ps.

32. D. Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, Philadelphia, PA, 2000.

33. D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

34. D. Peleg and J. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.

35. D. Peleg and E. Upfal. A tradeoff between space and efficiency for routing tables. *Journal of ACM*, 36(3):510–530, 1989.

36. L. Roditty, M. Thorup, and U. Zwick. Roundtrip spanners and roundtrip routing in directed graphs. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 844–851, 2002.

37. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graph. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

38. U. Zwick. Exact and approximate distances in graphs - a survey. In *Proc. 9th European Symposium on Algorithms*, pages 33–48, 2001.