Yale University
Department of Computer Science

Efficient Algorithms for Constructing $(1+\epsilon, \beta)$-Spanners in the Distributed and Streaming Models

Michael Elkin[1]
Computer Science Department, Yale University
elkin@cs.yale.edu

Jian Zhang[2]
Computer Science Department, Yale University
zhang-jian@cs.yale.edu

YALEU/DCS/TR-1275
Feb. 2004

This work was supported by the DoD University Research Initiative (URI) administered by the Office of Naval Research under Grant N00014-01-1-0795.

[1]Supported by ONR.
[2]Supported by ONR and NSF.

Efficient Algorithms for Constructing $(1+\epsilon,\beta)$-Spanners in the Distributed and Streaming Models

Michael Elkin*
Computer Science Department, Yale University
elkin@cs.yale.edu

Jian Zhang†
Computer Science Department, Yale University
zhang-jian@cs.yale.edu

Abstract

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 [19] 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 [17]. 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 [17], and it constructs spanners with essentially the same properties as the spanners that are constructed by the protocol of [17].

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-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 [20] 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 [20] uses just one pass over the input, as opposed to the constant number of passes in our algorithm. 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

This work 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.

†Supported by ONR and NSF.

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 [29, 4, 12, 10, 16, 19, 17, 32, 8, 9] in the course of the last fifteen years.

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* but 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 (or *protocol*) is the (worst-case) number of rounds that are required for the protocol to complete its execution, and the *message* (or *communication*) *complexity* of a protocol is the (worst-case) number of messages that are sent by the processors during the execution of the protocol. The design of distributed algorithms is a vivid area of research (see, e.g., [28], and the references therein). Spanners serve as an important tool in this area, and particularly, they were used for routing [31, 7], for constructing synchronizers [30, 6], and for computing almost shortest paths [4, 17].

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 Schaffer [29], 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 Schaffer [29] 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 [19] is not known to translate to an efficient distributed algorithm, soon afterwards Elkin [17] 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 [17] to use $(1 + \epsilon, \beta)$-spanners for efficient algorithms for computing almost shortest paths from s sources.

Specifically, it was shown in [17] 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 [17], and the parameters of the constructed spanners are also essentially the same as in the result of [17]. 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 [17] 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.

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, 24, 21], and is important for processing massive data sets (see, e.g., [27], and the references therein). While originally the research in this area focused on numerical problems such as estimating the frequency moments [3, 15], and L^1- and L^2-distances [21, 25], more recently the problems involving massive graphs have become a subject of study [27, 20]. In this paper we show that our distributed algorithm can be converted into a streaming algorithm for constructing $(1+\epsilon, \beta)$-spanners using only a *constant number of passes* through the input, $O(n^{1+1/\kappa} \cdot \log n)$ bits of space, and $O(n^{\rho})$ processing time-per-edge, for an arbitrarily small control parameter $\rho > 0$ (this parameter 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 $u, v \in V$, the algorithm outputs a path of length at most $(1+\epsilon) \cdot dist_G(u, v) + \beta$. The only previously known streaming algorithm for this problem due to Feigenbaum *et al.* [20] 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.* [20], it provides *far shorter* paths and distance estimates, and has a *drastically smaller processing time-per-edge*. Note that both these algorithms do not have enough storage to keep the paths or estimates that they compute in memory, and instead they output each such path after computing it, and write a new path on top of it. We remark that in the traditional (not streaming) model of computation, the problem of computing almost shortest paths was the focus of extensive recent research ([16, 14, 17]; see also the excellent survey [34] 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, 22, 33], or on multiple BFS traversals of the *entire* graph [1, 16, 14]. 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.

Finally, we remark that our adaptation of the algorithm of [18] 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 [23, 11] for clustering metric space. 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 these of [23, 11].

1.3 Our Techniques

Our distributed algorithm for constructing $(1+\epsilon,\beta)$-spanners builds upon the previous work of [17]. The algorithm of [17], like our algorithm, uses extensively a subroutine for constructing *neighborhood covers* [4, 12, 17] (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 [17] 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 [18]. However, plugging the subroutine of [18] in the algorithm of [17] 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 [17]. 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 $dist_G(u,v)$ the *distance* between two vertices u and v in the graph G, that is, the length of the shortest path between them. Given a subset $V' \subseteq V$, denote by $E_G(V')$ the set of edges in G *induced* by V', i.e., $E_G(V') = \{(u,v) \mid (u,v) \in E \text{ and } u,v \in V'\}$. Let $G(V') = (V', E_G(V'))$. Denote by $\Gamma_k(v,V')$ the *k-neighborhood* of vertex v in the graph $(V', E_G(V'))$, i.e., $\Gamma_k(v,V') = \{u \mid u \in V', \text{ and } dist_{(V',E_G(V'))}(u,v) \le k\}$. Let $diam(G)$ denote the *diameter* of the graph G, i.e., $diam(G) = max_{u,v\in V} dist_G(u,v)$. 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 $diam(\mathcal{F}) = \max_{V'\in\mathcal{F}}\{diam(V')\}$. Unless specified explicitly, we say that an event happens with high probability if the probability is at least $1 - \frac{1}{n^{\Omega(1)}}$.

For a graph $G=(V,E)$ and two integers $\kappa, W > 0$, a (κ,W)*-cover* [4, 12, 17] $\mathcal{C}$ is a collection of not necessarily disjoint subsets $C \subseteq V$ that satisfy the following conditions. (1) $\bigcup_{C\in\mathcal{C}} C = V$. (2) $diam(\mathcal{C}) \le c_0 \cdot \kappa W$, where c_0 is a small constant. (3) The *size* of the cover $s(\mathcal{C}) = \sum_{C\in\mathcal{C}} |C|$ is at most $O(n^{1+1/\kappa})$, and furthermore, every vertex belongs to $O(polylog(n) \cdot n^{1/\kappa})$ 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 $C \in \mathcal{C}$ that contains both vertices, along with the shortest path between them.

Our algorithm is an improved version of the algorithm of [17]. The distributed algorithm of [17] is based on the following sequential construction (which dates back to the parallel construction of hopsets due to [13]): given a graph, construct a (κ, W)-cover for the graph. Intuitively, there are only a small number of large clusters in the cover. Hence, we can afford to connect, by the shortest paths between them, the large clusters that are close to each other. Observe that for any two vertices that belong to two different large clusters, the shortest path between them can be approximated by the shortest path between the two clusters plus the sum of the diameters of the two clusters. For the small clusters of the cover, we recursively build covers for all of them. Given any two vertices in the graph, the shortest path between them consists of segments, and each segment can be approximated in the above way. We next describe the algorithm in a more formal way.

Algorithm 2.1 (RECUR_SPANNER). *The input to the algorithm is a graph $G = (V, E)$ and four parameters κ, ν, D, and K, where κ, D and K are positive integers and $0 < \nu < 1$ is a small positive number.*

1. $\ell \leftarrow \lceil \log_{1/(1-\nu)} \log_K |V| \rceil$.
2. *Construct a (κ, D^ℓ)-cover $\mathcal{C}$ for G. Set $\mathcal{C}^H \leftarrow \{C \in \mathcal{C} \mid |C| \geq |V|^{1-\nu}\}$, $\mathcal{C}^L \leftarrow \mathcal{C} \setminus \mathcal{C}^H$. (We call the clusters from $\mathcal{C}^H$ "large clusters," and the clusters from $\mathcal{C}^L$ "small clusters.")*
3. **Interconnecting subroutine:** *For all pairs of clusters $C_1, C_2 \in \mathcal{C}^H$ 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 $C \in \mathcal{C}^L$, invoke Recur_Spanner$(G' = (C, E_G(C)), \kappa, \nu, D, K)$.*

Algorithm RECUR_SPANNER constructs the spanner that has the following properties:

Theorem 2.1. [17] *Let G=(V,E) be an n-vertex graph, $\kappa = 1, 2, \ldots, 0 < \nu \leq 1/2 - 1/\kappa, D = 2, 3, \ldots, K = 1, 2, \ldots$. let $\ell = \lceil \log_{1/(1-\nu)} \log_K n \rceil$. Let H be the spanner constructed by Algorithm RECUR_SPANNER invoked on the five-tuple (G, κ, ν, D, K). Then H is a $(1 + \frac{4c_0\kappa\ell}{D}, 4c_0\kappa D^\ell)$-spanner, where c_0 is a small universal constant, and its size is at most $O((K + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}})$.*

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 $\mathcal{C}^H$ is independent of the cover constructions within clusters of $\mathcal{C}^L$. 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 [17] in two ways. First, the algorithm of [17] 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 [17] 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 [18] 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 [18] 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 briefly describe the construction of (κ, W)-covers [12, 18]. This construction builds a (κ, W)-cover in κ phases. A vertex v in graph G is called *covered* if there is a cluster $C \in \mathcal{C}$ 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 2.2 (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 $p_i = min\{1, \log n \cdot \frac{n^{i/\kappa}}{n}\}$ 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 D be the union of the core sets of the clusters constructed on step (b). Set $U_{i+1} \leftarrow U_i \setminus D$.*

Theorem 2.2. [18] *Given an unweighted undirected n-vertex graph, Algorithm 2.2 constructs a (κ, W)-cover such that, with probability at least $1 - \frac{1}{n^{\Omega(1)}}$, every vertex is included in at most $O(\kappa \log n \cdot n^{1/\kappa})$ clusters of the cover. The construction requires $O(\kappa^2 \log n \cdot n^{1/\kappa} \cdot W)$ rounds of distributed computation.*

We now present our distributed implementation of Algorithm RECUR_SPANNER, that uses the above algorithm for constructing covers. Given a cluster C, let $\mathcal{C}(C)$ be the cover constructed for the graph $(C, E_G(C))$. For a cluster $C' \in \mathcal{C}(C)$, we define $Parent(C') = C$. Denote by $\mathcal{C}_i$ the collection of all the clusters on level i. Let $\mathcal{C}_i^H$ be the set of large clusters of $\mathcal{C}_i$, and $\mathcal{C}_i^L$ the set of the small clusters of $\mathcal{C}_i$.

Algorithm 2.3 (SPANNER). *The input to the algorithm is a graph $G = (V, E)$ on n vertices, and four parameters κ, ν, D, and K, where κ, D and K are positive integers and $0 < \nu < 1$ is a small positive number.*

1. $\mathcal{C}_0^L \leftarrow V$, $\mathcal{C}_0^H = \phi$.
2. *For level* $i = 1, 2, \ldots, \ell = \lceil \log_{1/(1-\nu)} \log_K n \rceil$ *do*
   (a) *For all clusters $C \in \mathcal{C}_{i-1}^L$, in parallel, construct (κ, W)-covers using Algorithm 2.2. Set $\mathcal{C}_i \leftarrow \bigcup_{C \in \mathcal{C}_{i-1}^L} \mathcal{C}(C)$, $\mathcal{C}_i^H \leftarrow \{C \in \mathcal{C}(C') \mid C' \in \mathcal{C}_{i-1}^L,$ and $|C| \geq |C'|^{(1-\nu)}\}$, $\mathcal{C}_i^L \leftarrow \mathcal{C}_i \setminus \mathcal{C}_i^H$.*
   (b) *For all clusters $C' \in \mathcal{C}_i^H$, in parallel, construct BFS trees in $G(C)$, where $C = Parent(C')$. For each cluster, the BFS tree is rooted at the center of the cluster C'.*
   Let $h = \lceil \log_{1/(1-\nu)} \log_K |Parent(C')| \rceil$. The depth of the tree is $2D^h + D^{h+1}$.
   (c) *For every BFS tree that was constructed on step (b), find all the vertices that serve as centers of clusters of C_i^H. For each such center and its cluster C'', if $Parent(C'') = Parent(C')$, where C' is the cluster centered at the root of the BFS tree, add to the spanner the shortest path between the root and this center.*
3. *Add to the spanner all the edges of the set* $\bigcup_{C \in \mathcal{C}_{\ell+1}} E_G(C)$.

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 2.3. 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 2.3) 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 steps (2b) and (2c).

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

Lemma 3.1. *For a vertex v, and an index $i = 1, 2, \ldots, \ell$, with high probability, the number of clusters $C \in \mathcal{C}_i$ that contain v is at most $O(n^{\frac{1}{\kappa\nu}})$.*

Proof: By Theorem 2.2, with high probability, in a cover constructed for a cluster $(C, E_G(C))$, a vertex is contained in at most $O(\log |C| \cdot |C|^{1/\kappa})$ clusters. Also note that $MS(\mathcal{C}_i^L) \leq n^{(1-\nu)^i}$.

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

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

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 3.2. *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 3.1, 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 $2\kappa \cdot D^{\log_{1/(1-\nu)} \log_K MS(\mathcal{C}_{i-1}^L)} \leq 2\kappa \cdot D^{\log_{1/(1-\nu)} \log_K n^{(1-\nu)^{i-1}}} = 2\kappa \cdot D^{\ell+1-i}$. 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 $O(|E| \cdot n^{\frac{1}{\kappa\nu}})$. $\square$

Lemma 3.3. *With high probability, for an index $i = 1, 2, \ldots, \ell$, 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 $C \in \mathcal{M}_{i-1}(v) \cap \mathcal{C}^L_{i-1}$. 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 $C \in \mathcal{M}_{i-1}(v) \cap \mathcal{C}^L_{i-1}$ 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

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

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 $\nu < 1$.

$\square$

Lemma 3.4. *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 $3D^{\log_{1/(1-\nu)} \log_K MS(\mathcal{C}^L_{i-1})+1} \leq 3D^{\log_{1/(1-\nu)} \log_K n^{(1-\nu)^{i-1}}+1} = 3D^{\ell+2-i}$. By Lemma 3.3, 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 $\mathcal{C}^H_i$ initiates a BFS exploration in the cluster $Parent(C) \in \mathcal{C}^L_{i-1}$. Following the analysis in Lemma 3.3, the number of large clusters in the cover $\mathcal{C}(Parent(C))$ 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, $e(\mathcal{C}_i) = \sum_{C \in \mathcal{C}_i} |E_G(C)|$ 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,

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

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

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

Theorem 3.1. *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 $K = n^{\delta/2}$, $\frac{1}{\kappa\nu} = \delta/2$, $\frac{1}{\kappa\nu} + \nu = \rho$. This gives $\nu = \rho - \frac{\delta}{2} > 0$, $\nu = O(1)$, $\kappa = \frac{2}{(\rho-\delta/2)\delta} = O(1)$, and $\ell = \log_{1/(1-\nu)} \log_K n = \log_{1/(1-\nu)} \frac{2}{\delta} = O(1)$, satisfying the requirement that κ, ν and ℓ are all constants.

Also set $\frac{4c_0\kappa\ell}{D} = \epsilon$. By Theorem 2.1, $(1+\epsilon)$ is the multiplicative stretch factor of the constructed spanner. This gives $D = \frac{4c_0\kappa\ell}{\epsilon} = O(1)$. Hence, $\beta = 4c_0\kappa D^{\ell} = O(1)$ is the additive term of the spanner. By Theorems 2.2 and 2.1, the size of our spanner is $O\left((K + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}}\right)$. For a sufficiently large n, $K = n^{\delta/2} > D^{\ell+1}$, hence the spanner size is $O(n^{1+\delta})$. By Lemma 3.2 and 3.4, 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 $O(|E| \cdot n^{\rho})$. □

4 Adaptation to the Streaming Model

In this section we adapt Algorithm 2.3 to the streaming model and devise an algorithm for computing all-pairs-almost-shortest-paths in this model.

First, 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. Consider a graph and two fixed vertices x and y of the graph. The graph also includes a path $(x, p_1), (p_1, p_2), \ldots, (p_t, y)$ 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. Given this graph, 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. Since the set-disjointness problem has a linear communication complexity [26], it has a linear streaming space complexity. It follows that a streaming algorithm for $o(n)$-approximating the distance between x and y in constant number of passes needs $\Omega(n)$ bits of space.

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 2.2 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 4.1. *With high probability, the streaming adaptation of Algorithm 2.2 constructs a (κ, W)-cover using $2\kappa^2 W$ passes over the input stream.*

Now, we briefly describe the adaptation of Algorithm 2.3 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 $\mathcal{C}_{i-1}^{L}$ 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 $\mathcal{C}_{i-1}^L$ 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 4.2. *With high probability, the streaming adaptation of Algorithm 2.3 requires $2\kappa^2 D^\ell + 3D^{\ell+1}$ passes over the input stream.*

Proof: Note that on level i, for cover construction, the value of W is bounded by $D^{\log_{1/(1-\nu)} \log_K MS(\mathcal{C}_i^L)} \leq D^{\log_{1/(1-\nu)} \log_K n^{(1-\nu)^i}} \leq D^{\ell-i}$. By Lemma 4.1, the number of passes that are required for constructing a cover on level i is at most $2\kappa^2 D^{\ell-i}$.

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 $3D^{\log_{1/(1-\nu)} \log_K MS(\mathcal{C}_i^L)+1} \leq 3D^{\ell-i+1}$.

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

Lemma 4.3. *The space complexity of the streaming adaptation of Algorithm 2.3 is*

$$O\left(\log n \cdot (K + D^{\ell+1}) \cdot n^{1+\frac{1}{\kappa\nu}}\right).$$

Proof: The algorithm needs storage space for the following: (1) the edges of the spanner; (2) each vertex needs to store the IDs of the clusters to which it belongs; (3) on each level i, each vertex v needs to store the IDs of all the clusters $C \in \mathcal{C}_i^H$ that initiated a BFS exploration as part of the execution of their interconnection procedure, and this exploration has visited v.

By Theorem 2.1 and Lemma 3.1, items (1) and (2) require $O((K + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}})$ and $O(n^{\frac{1}{\kappa\nu}})$ cells of memory, respectively. Following Lemma 3.3, it is easy to see that item (3) requires $O(n^{\frac{1}{\kappa\nu}+\nu})$ cells of memory. Note that these 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:

$$O(\log n) \cdot \left(O\left((K + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}}\right) + O\left(n^{\frac{1}{\kappa\nu}}\right) + O\left(n^{\frac{1}{\kappa\nu}+\nu}\right)\right) = O\left(\log n \cdot (K + D^{\ell+1}) \cdot n^{1+\frac{1}{\kappa\nu}}\right).$$

□

Lemma 4.4. *With high probability, the streaming adaptation of Algorithm 2.3 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 3.1, $T_{cover}(v) = O(n^{\frac{1}{\kappa\nu}})$.

During interconnection, the algorithm may also need to examine the vertex v for each cluster $C \in \mathcal{C}_i^H$ whose BFS exploration visits v. Let $T_{interconnect}(v)$ be this processing time. By Lemma 3.3, $T_{interconnect}(v) = O(n^{\frac{1}{\kappa\nu}+\nu})$.

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

□

To summarize,

Theorem 4.1. *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 $1 - \frac{1}{n^{\Omega(1)}}$, 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 4.1. *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 [17]. 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 [17] 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 [20] 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.

Acknowledgment: The authors are very grateful to Joan Feigenbaum for inspiring and stimulating discussions, 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] Noga Alon, Yossi Matias, and Mario 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.E. 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] S. Baswana and S. Sen. A simple linear time algorithm for computing a (2k-1)-spanner of o(n1+1/k) size in weighted graphs. In *Proc. International Colloq. on Automata, Languages and Computing*, pages 284–296, 2003.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] E. Cohen and U. Zwick. All pairs small stretch paths. *Journal of Algorithms*, 38:335–353, 2001.

[15] D. Coppersmith and R. Kumar. An improved data stream algorithm for frequency moments. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[16] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.

[17] M.L. Elkin. Computing almost shortest paths. In *Proc. 20th ACM Symposium on Principles of Distributed Computing*, pages 53–62, 2001.

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

[19] M.L. 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.

[20] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Yale University Technical Report YALEU/DCS/TR-1262*, 2003.

[21] 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.

[22] 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.

[23] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *Proc. 41th IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

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

[25] 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.

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

[27] S. Muthukrishnan. Data streams: Algorithms and applications. 2003. Available at http://athos.rutgers.edu/~muthu/stream-1-1.ps.

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

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

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

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

[32] 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.

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

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