Computing Almost Shortest Paths *

Michael Elkin *

Abstract

We study the *s-sources almost shortest paths* (shortly, *s-ASP*) problem. Given an unweighted graph $G = (V, E)$, and a subset $S \subseteq V$ of s nodes, the goal is to compute almost shortest paths between all the pairs of nodes $S \times V$. We devise an algorithm with running time $O(|E|n^{\rho} + s \cdot n^{1+\zeta})$ for this problem that computes the paths $P_{u,w}$ for all pairs $(u, w) \in S \times V$ such that the length of $P_{u,w}$ is at most $(1+\epsilon)d_G(u, w) + \beta(\zeta, \rho, \epsilon)$, and $\beta(\zeta, \rho, \epsilon)$ is constant when ζ, ρ and ϵ are (one can choose arbitrarily small constants ζ, ρ and ϵ).

We also devise a distributed protocol for the s-ASP problem that computes the paths $P_{u,w}$ as above, and has time and communication complexities of $O(s \cdot Diam(G) + n^{1+\zeta/2})$ (resp., $O(s \cdot Diam(G) \log^3 n + n^{1+\zeta/2} \log n)$) and $O(|E|n^{\rho} + s \cdot n^{1+\zeta})$ (resp., $O(|E|n^{\rho} + s \cdot n^{1+\zeta} + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}})$) in the synchronous (resp., asynchronous) setting.

Our sequential algorithm, as well as the distributed protocol, is based on a novel algorithm for constructing $(1+\epsilon, \beta(\zeta, \rho, \epsilon))$-spanners of size $O(n^{1+\delta})$, developed in this paper. This algorithm has running time of $O(|E|n^{\rho})$, which is significantly faster than the previously known algorithm of [20], whose running time is $\tilde{O}(n^{2+\rho})$. We also develop the first distributed protocol for constructing $(1+\epsilon, \beta)$-spanners. The time and communication complexities of this protocol are near-optimal.

*Preliminary version of this paper appeared in PODC'01 [19].

*Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Rehovot, 76100 Israel. elkin@wisdom.weizmann.ac.il.

1 Introduction

Consider the *s-sources almost shortest paths* (henceforth referred to as *s-ASP*) problem. Given an unweighted (undirected) graph $G = (V, E)$, and a subset $S \subseteq V$ of s sources, the goal is to compute the paths $P_{u,w}$ between all the pairs of nodes $(u, w) \in S \times V$. It is required that the length of the computed paths $P_{u,w}$ will be close to the distance $d_G(u, w)$ between the nodes u and w in G in the following sense: we say that an algorithm A is an s-ASP algorithm *with error* (α, β) if for any pair of nodes $(u, w) \in S \times V$, the length of $P_{u,w}$ (denoted by $L(P_{u,w})$) returned by the algorithm is at most $\alpha \cdot d_G(u, w) + \beta$. Here α is called the *multiplicative term of the error* of the algorithm A, and β is called the *additive term of the error*. The particular case of the problem when $S = V$ is referred to as the *APASP* (standing for *all pairs almost shortest paths*) problem. The s-ASP (resp., APASP) problem with no error (i.e., with error terms $(1, 0)$) is called the s-SP (resp., APSP) problem.

Shortest paths computation is one of the most fundamental algorithmic graph problems. We distinguish three main approaches for constructing efficient algorithms for this problem. The first one is through matrix multiplication. The main advantage of this approach is that it enables to compute the shortest paths, rather than the almost shortest ones. However, its main drawback is that the currently known non-trivial matrix multiplication algorithms are impractical, since their time complexities involve some huge constants. Another disadvantage is that this approach does not enable to provide for the s-SP problem faster algorithms than for the APSP problem. In other words, when one is interested in computing the paths from s nodes to all other nodes, the algorithm will compute the paths between all the pairs of nodes in the graph and discard all the pairs that are not required. Finally, the currently best known time complexity of algorithms of this type is $\tilde{O}(n^{2.376})$ [6, 7, 23, 24, 28], which is significantly higher than the time complexity of algorithms that follow other approaches.

The second approach is to construct a sparse spanner for the given graph, and to perform a straightforward algorithm for computing the shortest paths on the spanner. For an unweighted graph $G = (V, E)$, $H \subseteq E$ is an (α, β)*-spanner of* G if for any pair of nodes $u, w \in V$, $d_H(u, w) \leq \alpha \cdot d_G(u, w) + \beta$. Spanners have various different applications in distributed computing [27, 8, 1] and computational geometry [4, 14, 18], apart of serving as a subroutine for algorithms for computing almost shortest paths. The straightforward algorithm for computing the shortest paths between all the pairs of nodes $(u, w) \in S \times V$ is the algorithm that constructs s BFS spanning trees for the graph G rooted in the s sources. This algorithm requires $O(n|E|)$ time. The existence of an algorithm for constructing an (α, β)-spanner with $O(n^{1+\zeta})$ edges at time $O(T)$ immediately implies the existence of an algorithm for the s-ASP problem with error (α, β), whose running time is $O(T + s \cdot n^{1+\zeta})$. Indeed, the algorithm for the s-ASP problem would just construct the spanner, and invoke the straightforward shortest-paths algorithm on the spanner. Such a scheme was employed in [1, 15], and yielded algorithms for the s-ASP problem that run in $O(|E|n^{\zeta} + s \cdot n^{1+\zeta})$ time and have error $(1/\zeta, 0)$, where ζ can be fixed as a constant that is arbitrarily close to zero.

This approach has two inherent advantages. The first one is that it is very convenient to build a distributed implementation for algorithms that follow this approach, because the spanner is a part of the existing graph (network), and so any distributed protocol can be executed on the spanner. The second advantage is in scenarios when using some edge for a path of $P_{u,w}$ has a specific cost, and one is interested in ensuring that altogether the paths $P_{u,w}$ would use only a restricted number of edges. In this approach, the number of edges used by the constructed paths is bounded by the size of the spanner, i.e., by $O(n^{1+\zeta})$. However, the disadvantage of the constructions of [1, 15] is that the resulting paths may be longer than the shortest paths by a *multiplicative factor of* $1/\zeta$. This drawback is inherited from the algorithms for constructing spanners that are used as the first stage of the algorithms of [1, 15] for the almost shortest paths problem.

This bottleneck led to the development of the third approach, that is, to compute the almost shortest

	Error Terms	Running Time	Applicability to distributed setting	Improvement for $s < n$	Bounded number of edges
Algorithms based on matrix multiplication [6, 7, 23, 24, 28]	$(1, 0)$	$\tilde{O}(n^{2.376})$	NO	NO	NO
Algorithms based on spanners [1, 15]	$(\lceil 1/\zeta \rceil, 0)$	$O(\|E\|n^{\zeta} + s \cdot n^{1+\zeta})$	YES	YES	YES
Direct approach [17, 3]	$(1, \lceil 1/\zeta \rceil)$	$\tilde{O}(n^{2+\zeta})$	unknown	NO	NO
Direct approach [16]	$(1+\epsilon, \beta(\zeta, \rho, \epsilon, n))$ $\beta(n) = \Omega(\text{polylog}(n))$	$O(\|E\|n^{\rho} + s \cdot n^{1+\zeta})$	unknown	YES	NO
Our algorithm	$(1+\epsilon, \beta(\zeta, \rho, \epsilon))$ $\beta = O(1)$	$O(\|E\|n^{\rho} + s \cdot n^{1+\zeta})$	YES	YES	YES

Table 1: Summary of main results concerning s-ASP problem.

paths *directly*, without constructing a spanner. This approach was found very fruitful in reducing the multiplicative error α from $O(1/\zeta)$ to 1 (note that $\alpha = 1$ means that there is no multiplicative error whatsoever). Specifically, in [16], an algorithm for the s-ASP problem with running time $O(|E|n^{\rho} + s \cdot n^{1+\zeta})$ and error $(1+\epsilon, \beta(\zeta, \rho, \epsilon, n))$ was presented. In this algorithm $\beta(\zeta, \rho, \epsilon, n) = \Omega(\text{polylog}(n))$, whenever $0 < \zeta, \rho < 1$ are arbitrarily small constants and $\epsilon = \Omega(\frac{1}{\text{polylog }(n)})$. An algorithm with error $(1, \lceil 1/\zeta \rceil)$ for the APASP problem was presented in [17], based on [3], and its running time is $\tilde{O}(n^{2+\zeta})$.

Although these results are quite close to the optimal, they suffer from several disadvantages. First, the algorithm of [17] does not handle the s-ASP problem faster, even when $s = |S| << n$. Secondly, the result of [16] involves a polylogarithmic additive error term even when $0 < \zeta, \rho, \epsilon < 1$ are all constant. Additionally, both these results do not guarantee any bound on the number of edges used by the union of all the computed paths. Finally, the global nature of these algorithms seems to prevent any distributed implementation of them from being efficient.

In this paper we present an algorithm that has running time of $O(|E|n^{\rho} + s \cdot n^{1+\zeta})$, like the one of [16], but its error is $(1+\epsilon, \beta(\zeta, \rho, \epsilon))$, where $\beta(\zeta, \rho, \epsilon)$ is *constant* whenever $0 < \zeta, \rho, \epsilon < 1$ are, for arbitrarily small constants ζ, ρ and ϵ. Our algorithm follows the second approach, i.e., it constructs a $(1+\epsilon, \beta)$-spanner and invokes the straightforward algorithm on the spanner. Hence, our algorithm enjoys the advantages of the second approach described above, i.e., it guarantees an upper bound of $O(n^{1+\zeta})$ on the overall number of edges that are used by all the computed paths; its running time decreases with s; and even more importantly, it has an efficient distributed implementation. Table 1 summarizes the previous results concerning the s-ASP problem, and compares them to our result.

The recent development that made our result possible was the construction of a $(1+\epsilon, \beta)$-spanner with $O(n^{1+\zeta})$ edges, where β is constant, i.e., independent of n [20]. This result breaks the bottleneck described above and constructs spanners whose multiplicative error term is arbitrarily close to 1, and whose size is arbitrarily close to $O(n)$. However, its high running time ($\tilde{O}(n^{2+\rho})$) makes the construction of [20] unsuitable for the s-ASP problem. It is also not clear whether that construction can be efficiently implemented in the distributed setting. Our second contribution is that we present a *faster sequential* *algorithm* for constructing $(1+\epsilon, \beta)$-spanners with $O(n^{1+\zeta})$ edges, which is also easily convertible into a distributed protocol. Specifically, we present an algorithm that constructs a $(1+\epsilon, \beta(\zeta, \rho, \epsilon))$-spanner with $O(n^{1+\zeta})$ edges in time $O(|E|n^{\rho})$, for arbitrarily small $0 < \zeta, \rho, \epsilon < 1$, and $\beta(\zeta, \rho, \epsilon)$ is constant whenever ζ, ρ and ϵ are. We note, however, that the constant $\beta(\zeta, \rho, \epsilon)$ is slightly higher in our construction than in that of [20]. Specifically, β depends on ζ as $(1/\zeta)^{\log 1/\zeta}$, depends inverse-exponentially on ρ and inverse-polynomially on ϵ. In the construction of [20], β depends on ζ as $(1/\zeta)^{\log\log 1/\zeta}$, and depends inverse-polynomially on both ρ and ϵ.

We also study the s-ASP problem in the distributed setting. Consider a network of processors that share no common memory (Internet may serve as an example). This network is modeled by a graph in which a pair of vertices share an edge if and only if the corresponding processors have a direct link between them. The model assumes that the network may be either *synchronous* or *asynchronous*. In either case each processor has its own clock. Depending on its clock and on external messages that the processor gets, it decides when to start the execution, when to send messages to its neighbors, and when to stop participating in the execution. The difference between the cases is that in the *synchronous* case the clocks of two different processors *always agree*, and it is not necessarily the case in the asynchronous network.

Another aspect of modeling a distributed network concerns with the amount of communication that can be sent through a link during a single unit of time. In the more abstract $LOCAL$ model, no restriction on the rate of communication is imposed, while in the more realistic $CONGEST$ model it is assumed that $O(\log n)$ bits (an upper bound on the size of a succinct identifier of a processor in the network) can be delivered through a link during a single unit of time. See [25] for an extensive discussion about these concepts. Throughout the paper we consider the $CONGEST$ model, and devise protocols for both synchronous and asynchronous settings.

In the distributed setting, the particular case of the s-SP problem when $s = 1$ (i.e., the single-source shortest paths problem) was intensively studied [22, 29, 21, 10, 11, 5, 12, 8, 9]. The more general s-SP and APSP problems in the distributed setting studied in [22, 29, 9]. We are not aware of any previous studies of the *approximate* versions of the s-SP and APSP problems (i.e, the s-ASP and the APASP problems) in the distributed setting. The 1-SP problem in the distributed setting is well-understood. In the synchronous setting its time and communication complexities are $\Theta(Diam(G))$ and $\Theta(E)$, respectively (cf. [25]). In the asynchronous setting its time and communication complexities are $O(Diam(G)\log^3 n)$ and $O(|E| + n\log^3 n)$ [8], and both are optimal up to polylogarithmic factors.

Note that any protocol for the 1-SP problem with time complexity $O(T)$ and communication complexity $O(\mathcal{M})$ can be easily converted into a protocol for the s-SP problem with time complexity $O(s \cdot T)$ and communication complexity $O(s \cdot \mathcal{M})$. This can be done by invoking the protocol for the 1-SP problem s times, separately for each source. This implies that the s-SP problem can be solved in the synchronous (resp., asynchronous) setting with time and communication complexities of $O(s \cdot Diam(G))$ (resp., $O(s \cdot Diam(G)\log^3 n)$) and $O(s \cdot |E|)$ (resp., $O(s \cdot (|E| + n\log^3 n))$). Analogously, in the asynchronous setting the result of [8] implies that the s-SP (resp., APSP) problem can be solved with time and communication complexity of $O(s \cdot Diam(G)\log^3 n)$ (resp., $O(n \cdot Diam(G)\log^3 n)$) and $O(s \cdot (|E| + n\log^3 n))$ (resp., $O(n \cdot (|E| + n\log^3 n))$).

A distributed protocol for the APSP problem with *expected* communication complexity of $O(n^2 \log n)$ in both the synchronous and asynchronous settings was presented in [22]. The expected communication complexity in this context is the average of the communication complexities of the protocol when invoked on some specific ensemble of n-vertex graphs. Although not stated explicitly, the worst case time complexity of the protocol is $O(n \cdot Diam(G))$ and $O(n \cdot Diam(G)\log^3 n)$ in the synchronous and asynchronous settings, respectively. A protocol with *worst case* communication complexity of $O(n^2 \log n)$ for the APSP problem was presented in [9], but, unfortunately, the time complexity of that protocol is as high as $O(n^2 Diam(G)\log n)$, even in the synchronous setting.

In this paper we present a distributed protocol for the s-ASP and APASP problems that achieves the same time complexity as in [22], and a communication complexity close to that in [9], at the cost of computing the *almost* shortest paths instead the absolutely shortest ones. Specifically, in the synchronous (resp., asynchronous) setting the time and communication complexity of our protocol for the s-ASP problem with error terms $(1+\epsilon, \beta(\zeta,\rho,\epsilon))$ are $O(s \cdot Diam(G) + n^{1+\zeta/2})$ (resp., $O(s \cdot Diam(G)\log^3 n + n^{1+\zeta/2} \cdot \log n)$) and $O(|E|n^\rho + s \cdot n^{1+\zeta})$ (resp., $O(|E|n^\rho + s \cdot n^{1+\zeta} + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}})$). In the asynchronous setting, the time and communication complexities of our protocol are $O(s \cdot Diam(G)\log^3 n + n^{1+\zeta/2}\log n)$ (resp., $O(n \cdot Diam(G)\log^3 n + n^{1+\zeta/2}\log n)$) and $O(|E|n^\rho + s \cdot n^{1+\zeta} + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}})$ (resp., $O(|E|n^\rho + n^{2+\zeta})$). Our protocol has error terms $(1+\epsilon, \beta(\zeta,\rho,\epsilon))$, i.e., computes the paths $P_{u,w}$ for any pair $\{u,w\} \in S \times V$ such that $L(P_{u,w}) \le (1+\epsilon)d_G(u,w) + \beta(\zeta,\rho,\epsilon)$.

Like in the sequential setting, our approach to solving the distributed variant of the s-ASP problem is by presenting a distributed protocol that constructs a $(1+\epsilon, \beta(\zeta,\rho,\epsilon))$-spanner for the input graph, and then invoking the straightforward protocol for the s-SP problem on the constructed spanner. We believe that distributed protocol presented in this paper for constructing sparse $(1+\epsilon,\beta)$-spanner is of independent interest, since previously no distributed protocols for constructing spanners of size $O(n^{1+\zeta})$ with multiplicative error term $o(1/\zeta)$ were known.

In the synchronous (resp., asynchronous) setting the complexities of our protocol for constructing a $(1+\epsilon, \beta(\zeta,\rho,\epsilon))$-spanner with $O(n^{1+\zeta})$ edges are $O(n^{1+\zeta/2})$ (resp., $O(n^{1+\zeta/2} \cdot \log n)$) time and $O(|E|n^\rho)$ (resp., $O(|E|n^\rho + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}})$) communication.

Finally, we adapt our algorithms to work on *weighted* graphs, both in the sequential and distributed settings. For a weighted graph $G = (V, E, \omega)$, denote by ω_{max} the maximum weight ratio $\max\{\frac{\omega(e_1)}{\omega(e_2)} \mid e_1, e_2 \in E\}$. We present two sequential algorithms for the s-ASP problem on weighted graphs. The first one computes the distances for all pairs of nodes $S \times V$ with error $(1+\epsilon, \omega_{max} \cdot \beta(\zeta,\rho,\epsilon))$ in time $O(|E|n^\rho + s \cdot n^{1+\zeta})$, but does not compute the actual paths. The second one computes the paths with the same error, but does it in time $O(|E|n^\rho + s \cdot \omega_{max} \cdot n^{1+\zeta})$. In the distributed setting we present a protocol for the s-ASP problem with this error that has $O(\omega_{max} \cdot n^{1+\zeta/2} + s \cdot Diam(G))$ (resp., $O(\omega_{max} n^{1+\zeta/2}\log^3 n + s \cdot Diam(G))$) time and $O(|E|n^\rho + s \cdot n^{1+\zeta})$ (resp., $O(|E|n^\rho + s \cdot n^{1+\zeta} + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}})$) communication complexities in the synchronous (resp., asynchronous) setting.

Our sequential algorithm for weighted graphs should be compared with the result of [16], that states that computing distances with error $(1+\epsilon, \omega_{max} \cdot \beta(\zeta,\rho,\epsilon,n))$ can be done in $O(|E|n^\rho + s \cdot n^{1+\zeta})$ time. However, in [16], $\beta(\zeta,\rho,\epsilon,n) = \Omega(\text{polylog}(n))$ even when ζ, ρ and ϵ are all constant. So, our result reduces the additive error term by a polylogarithmic factor, while keeping the same running time.

Concerning our techniques, the main idea of our fast algorithm for constructing $(1+\epsilon,\beta)$-spanners with $O(n^{1+\zeta})$ edges is adapted from the algorithm for constructing hopsets of [16]. However, although hopsets are related to spanners, there are some essential differences. The main difference is that hopsets need not be subgraphs of the original graph, but rather may contain edges that were not present in the original graph. Furthermore, these edges may have arbitrary weights, even if the original graph is unweighted. These differences make it hard to convert the hopset construction algorithm of [16] directly into an algorithm for constructing spanners. Some of the additional ingredients that made this conversion possible were the techniques developed in [20].

For the distributed implementation, the main tools that are used by our protocol are the efficient protocol for constructing (κ, W)-covers due to [1], and the efficient protocol for constructing BFS spanning tree for a graph in the asynchronous setting due to [8].

The paper has the following structure. Section 2.1 presents the sequential Algorithm *Recur_Spanner* for constructing $(1+\epsilon, \beta(\zeta,\rho,\epsilon))$-spanners with $O(n^{1+\zeta})$ edges in $O(|E|n^\rho)$ time. Sections 2.2, 2.3 and 2.4 contain the analysis of the size of the spanner constructed by the algorithm, the stretch properties of the constructed spanner and the running time of the algorithm, respectively. The results concerning

Algorithm *Recur_Spanner* are summarized in Theorem 2.15. Section 2.5 contains several modifications of Algorithm *Recur_Spanner* and their analysis. One of these modifications enables to generalize our results to weighted graphs, and another is needed to simplify the the distributed implementation. Section 2.6 presents our results for the s-ASP and APASP problems in the sequential setting. These results are summarized in Theorem 2.18. Section 3 is devoted to the overview of the distributed implementation of Algorithm *Recur_Spanner*. In Section 3.1 we give an overview of the distributed protocol, which we formally describe in Section 3.2. We analyze the protocol in Section 3.3.

2 Sequential Construction

In this section we present a sequential construction of $(1+\epsilon, \beta(\zeta,\rho,\epsilon))$-spanner of size $O(n^{1+\zeta})$, with β roughly $O((1/\zeta)^{\log\frac{1}{\zeta\epsilon}}(1/\rho)^{1/\rho})$, for arbitrarily small $0 < \zeta, \rho, \epsilon < 1$. We show in Section 2.4 that this construction can be implemented faster than the construction of [20], which has similar parameters.

2.1 Algorithm

One of the major building blocks that are used by the construction is the (κ, W)-*cover*, that was presented and analyzed in [1].

We need some notation. For a subset of nodes $C \subseteq V$, let the *diameter* of C be $Diam(C) = \max\{Diam(C) \mid C \in \mathcal{C}\}$. For a collection of subsets of vertices $\mathcal{C} = \{C_1, C_2, \ldots, C_t\}$ let the *vertex* (resp., *edge*) *size* of $\mathcal{C}$ be $s(\mathcal{C}) = \sum_{C\in\mathcal{C}} |C|$ (resp., $e(\mathcal{C}) = \sum_{C\in\mathcal{C}} |E(C)|$).

Definition 2.1 *For any weighted graph $G = (V, E, \omega)$ and positive integer numbers κ, W, a collection $\mathcal{C}$ of connected subsets $C \subseteq V$ (later called* clusters*) is a (κ, W)*-cover *of G if the following six conditions hold.*

1. $\bigcup_{C\in\mathcal{C}} C = V$.
2. $Diam(\mathcal{C}) = c_0 \cdot \kappa W$, *where c_0 is some small constant.*
3. $s(\mathcal{C}) = O(n^{1+1/\kappa})$.
4. $e(\mathcal{C}) = O(|E| n^{1/\kappa})$.
5. *For any pair of vertices $u, w \in V$ such that $d_G(u,w) \le W$ there exists a cluster $C \in \mathcal{C}$ which contains both nodes, and also all the nodes of some shortest path $P_{u,w}$ between u and w.*
6. *Every cluster $C \in \mathcal{C}$ contains a subset $ker(C) \subseteq C$, called* the kernel of C *such that V is a disjoint union of these kernels. I.e., $V = \bigcup_{C\in\mathcal{C}} C$ and for any pair of clusters $C, C' \in \mathcal{C}$, $C \neq C'$, $ker(C)$ and $ker(C')$ are disjoint.*

Any collection $\mathcal{C}$ that satisfies conditions 2-6 is called a partial (κ, W)-cover *of G.*

It was shown in [1, 15] that for any weighted graph G and integers $\kappa, W \ge 1$ a (κ, W)-cover for the graph can be sequentially constructed in $O(|E| n^{1/\kappa})$ time. The (κ, W)-covers were also studied in [15], where a different and somewhat better construction of (κ, W)-covers is presented.

Next, we describe the algorithm for constructing the spanner. The algorithm is given an unweighted graph $G' = (V', E')$, positive integers κ, D and K, and a real number $0 < \nu \le 1/2 - 1/\kappa$. The algorithm returns an edge set $H_C \subseteq E'$, later referred to as a *spanner*.

Algorithm *Recur_Spanner*(G', κ, ν, D, K) for $G' = (V', E')$
starts with creating a (κ, D^ℓ)-cover $\mathcal{C}$ for the graph G', where $\ell = \lceil \log_{1/(1-\nu)} \log |V'| \rceil$. Next, it partitions

$\mathcal{C}$ to two sub-covers, $\mathcal{C}^H$ and $\mathcal{C}^L$, where $\mathcal{C}^H$ contains "relatively big" clusters, i.e., clusters of size at least $|V'|^{1-\nu}$, and $\mathcal{C}^L$ contains the "relatively small" ones, i.e., of size at most $|V'|^{1-\nu}$. For any pair of big clusters that are "relatively close" one to another, i.e., at distance at most $D^{\ell+1}$, the algorithm inserts one of the shortest paths between them into the spanner. For any small cluster it recursively applies Algorithm *Recur_Spanner* on the cluster with the same κ, ν, D and K, that do not change through the recursion. Consider the tree of recursive invocations of Algorithm *Recur_Spanner*, where the root of the tree is the invocation $Recur\_Spanner(G,\kappa,\nu,D,K)$, and there is an edge between $Recur\_Spanner(G',\kappa,\nu,D,K)$ and $Recur\_Spanner(G'')$ if the latter was recursively invoked from the former. Let the *recursion level* of an invocation $Recur\_Spanner(G',\kappa,\nu,D,K)$ be the depth of the invocation $Recur\_Spanner(G',\kappa,\nu,D,K)$ in this tree. (In particular, the depth of $Recur\_Spanner(G,\kappa,\nu,D,K)$ is 1.)

The upper bound on the size of clusters, on which the algorithm is recursively invoked, decreases double-exponentially fast when the recursion level i grows. In particular, when i becomes equal to $\ell = \lceil \log_{1/(1-\nu)} \log_K n \rceil$, where n is the size of the vertex set of the input graph $G = (V, E)$ of the algorithm, this upper bound becomes at most K. When Algorithm *Recur_Spanner* is invoked on a graph with K nodes, then no matter what are the values of other parameters, the algorithm returns as its output the entire edge set of its input graph. This is the stopping condition of the recursion.

Intuitively, ν controls the running time of the algorithm and κ and D affect the ϵ and β parameters of $(1+\epsilon,\beta)$-spanner. More specifically, D determines the value of ϵ, and ϵ, κ, ν and K determine the additive term β.

The formal description of Algorithm *Recur_Spanner* follows.

Algorithm *Recur_Spanner*
Input: Graph $G' = (C, E_C)$, $\kappa = 1,2,\ldots$, $0 < \nu \le 1/2 - 1/\kappa$, $D = 1,2,\ldots$, $K = 1,2,\ldots$
Output: Edge set $H_C \subseteq E_C$

1. **If** $|C| \le K$ **then**
   $H_C \leftarrow E_C$;
   Else
2. (a) $H_C \leftarrow \emptyset$; $\ell \leftarrow \lceil \log_{1/(1-\nu)} \log_K |C| \rceil$;
   (b) Form a (κ, D^ℓ)-cover $\mathcal{C}$ for G'; Insert into the spanner H_C the BFS trees of all clusters of $\mathcal{C}$;
   (c) $\mathcal{C}^H \leftarrow \{C' \in \mathcal{C} \mid |C'| \ge |C|^{1-\nu}\}$;
   (d) **For** all pairs $C_1, C_2 \in \mathcal{C}^H$ s.t. $d_{G'}(C_1, C_2) \le D^{\ell+1}$ **do**
      i. Compute one of the shortest paths P between C_1 and C_2 in G';
      ii. $H_C \leftarrow H_C \cup E(P)$;
   (e) $\mathcal{C}^L \leftarrow \mathcal{C} \setminus \mathcal{C}^H$;
   (f) **For** all clusters $C' \in \mathcal{C}^L$ **do**
      i. $H' \leftarrow Recur\_Spanner((C', E_C(C')), \kappa, \nu, D, K)$;
      ii. $H_C \leftarrow H_C \cup H'$;
3. Return(H_C);

2.2 Size Analysis

In this section we analyze the size of the spanner output by Algorithm *Recur_Spanner*, that was presented in the previous section.

Let us introduce some additional notation. For any $i = 1, \ldots, \ell$, let $\mathcal{C}_i$ (resp., $\mathcal{C}_i^L$) denote the union of all the covers $\mathcal{C}$ (resp., $\mathcal{C}^L$) that were formed by the invocations of Algorithm *Recur_Spanner* at ith level of the recursion. For any cluster $C \in \mathcal{C}_i^L$ for some $i = 1, \ldots, \ell - 1$, denote by $\mathcal{C}(C)$ the cover that was formed by the invocation of Algorithm *Recur_Spanner* on this cluster. Also denote $\mathcal{C}^H(C) = \{C' \in \mathcal{C}(C) \mid |C'| \geq |C|^{1-\nu}\}$ and $\mathcal{C}^L(C) = \mathcal{C}(C) \setminus \mathcal{C}^H(C)$. For any cover $\mathcal{C}$, denote $ms(\mathcal{C}) = \min_{C \in \mathcal{C}} |C|$ and $MS(\mathcal{C}) = \max_{C \in \mathcal{C}} |C|$.

We start by proving the following simple lemmas.

Lemma 2.2 *For any $i = 1, 2, \ldots, \ell$, $MS(\mathcal{C}_i^L) \leq n^{(1-\nu)^i}$.*

Proof: The proof is by induction on i. Note that $\mathcal{C}_1^H$ is formed by step 2c of the invocation *Recur_Spanner*(G, ν, μ, D, K), and so $\mathcal{C}_1^H = \{C' \in \mathcal{C}_1 \mid |C'| \geq |V|^{1-\nu}\}$. Hence $MS(\mathcal{C}_1^L) \leq n^{1-\nu}$.

For the induction step, consider $\mathcal{C}_{i+1}$ for some $i = 1, 2, \ldots, \ell - 1$. By definition of $\mathcal{C}_i$ and by step 2(f)i of Algorithm *Recur_Spanner*, $\mathcal{C}_{i+1} = \bigcup_{C \in \mathcal{C}_i^L} \mathcal{C}(C)$.

Consider some $C' \in \mathcal{C}_{i+1}^H$. It follows that $C' \in \mathcal{C}(C)$ for some $C \in \mathcal{C}_i^L$ and $|C'| \leq |C|^{1-\nu}$. By the induction hypothesis, $MS(\mathcal{C}_i^L) \leq n^{(1-\nu)^i}$. Hence $|C'| \leq |C|^{1-\nu} \leq (MS(\mathcal{C}_i^L))^{1-\nu} \leq n^{(1-\nu)^{i+1}}$. ∎

Note that by step 2f of Algorithm *Recur_Spanner* for any $i = 1, \ldots, \ell - 1$,

$$\sum_{C \in \mathcal{C}_i^L} s(\mathcal{C}(C)) = \sum_{C \in \mathcal{C}_{i+1}} |C| = s(\mathcal{C}_{i+1}) \geq s(\mathcal{C}_{i+1}^L) \ . \tag{1}$$

Let us next prove the following lemma.

Lemma 2.3 *For $\kappa = 1, 2, \ldots$, $i = 1, 2, \cdots, \ell$, and real constant $0 < \nu \leq 1/2 - 1/\kappa$, $s(\mathcal{C}_i) = O(n^{1 + \frac{1}{\kappa} \frac{1-(1-\nu)^i}{\nu}})$.*

Proof: The proof is by induction on i. First, note that

$$s(\mathcal{C}_1) = \sum_{C \in \mathcal{C}_1} |C| = O(n^{1+1/\kappa}) \ , \tag{2}$$

by property 3 of (κ, W)-cover, for any $W = 1, 2, \ldots$ (see Definition 2.1).

Next, again by property 3 of (κ, W)-covers for any W, and by Lemma 2.2, observe that for $i \geq 2$

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

Hence, by the induction hypothesis,

$$s(\mathcal{C}_i) \leq n^{1 + \frac{1}{\kappa}\left(\frac{1-(1-\nu)^{i-1}}{\nu} + (1-\nu)^{i-1}\right)} = n^{1 + \frac{1}{\kappa} \frac{1-(1-\nu)^i}{\nu}} \ . \qquad \blacksquare$$

It follows that $s(\mathcal{C}_i) \leq n^{1 + \frac{1}{\kappa\nu}}$, for any $i = 1, 2, \ldots, \ell$.

Note that $s(\mathcal{C}_i)$ is an upper bound on the number of edges inserted into the spanner on the ith level of the recursion by step 2b of Algorithm *Recur_Spanner*.

Hence we get the following corollary.

Corollary 2.4 *For any $\kappa = 1, 2, \ldots$, $0 < \nu < 1/2 - 1/\kappa$, the total number of edges inserted into the spanner due to the insertion of the BFS trees of different clusters (steps 2b of Algorithm Recur_Spanner on different levels of the recursion) during the algorithm is $O(n^{1 + \frac{1}{\kappa\nu}})$.*

Proof: As argued above the size of the edge set is bounded by

$$\sum_{i=1}^{\ell} s(\mathcal{C}_i) \le \sum_{i=1}^{\ell} n^{1+\frac{1-(1-\nu)^i}{\nu\kappa}} = O(n^{1+\frac{1}{\kappa\nu}}) \ . \quad \blacksquare$$

Let us next bound the number of edges inserted to the spanner as result of the execution of steps 2d of Algorithm *Recur_Spanner* at different levels of the recursion.

Lemma 2.5 *For any $\kappa, D = 1, 2, \ldots$, $0 < \nu \le 1/2 - 1/\kappa$, the total number of edges added to the spanner as result of the insertion of shortest paths between big clusters at different levels of the recursion is bounded by $O(D^{\ell+1} n^{1+\frac{1}{\kappa\nu}})$.*

Proof: Consider some cluster C that is formed during the algorithm, and consider the cover $\mathcal{C}^H(C)$. By step 2c of Algorithm *Recur_Spanner*, $ms(\mathcal{C}^H(C)) \ge |C|^{1-\nu}$. Also, by property 3 of the covers constructed by the algorithm (see Def. 2.1), $s(\mathcal{C}^H(C)) \le |C|^{1+1/\kappa}$. Hence,

$$|\mathcal{C}^H(C)| \le \frac{|C|^{1+1/\kappa}}{ms(\mathcal{C}^H(C))} \le |C|^{\nu+1/\kappa} \ .$$

The number of paths added to the spanner on the first level of recursion step 2d of Algorithm *Recur_Spanner* is at most

$$|\mathcal{C}_1^H|^2 = |\mathcal{C}^H(V)|^2 \le n^{2\nu+2/\kappa} \ . \tag{3}$$

For any $i = 2, \ldots, \ell$, the number of paths added to the spanner at step 2d of Algorithm *Recur_Spanner* at the ith level of the recursion is at most

$$\sum_{C \in \mathcal{C}_{i-1}^L} |\mathcal{C}^H(C)|^2 \le \sum_{C \in \mathcal{C}_{i-1}^L} |C|^{2\nu+2/\kappa} \ .$$

However, since $2\nu + 2/\kappa \le 1$, it follows that

$$\sum_{C \in \mathcal{C}_{i-1}^L} |\mathcal{C}^H(C)|^2 \le \sum_{C \in \mathcal{C}_{i-1}^L} |C| \le s(\mathcal{C}_{i-1}^L) \ . \tag{4}$$

Hence, the number of paths added to the spanner at steps 2d of Algorithm *Recur_Spanner* on the different levels of the recursion is at most $\sum_{i=1}^{\ell} s(\mathcal{C}_i) = O(n^{1+\frac{1}{\kappa\nu}})$. Since every such a path is of length at most $D^{\ell+1}$, it follows that the total number of edges added to the spanner at these steps is at most $O(D^{\ell+1} n^{1+\frac{1}{\kappa\nu}})$. $\blacksquare$

Next, let us bound the number of edges inserted into the spanner in steps 1 of Algorithm *Recur_Spanner*.

Lemma 2.6 *The number of edges added to the spanner on the very last level of the recursion (i.e., the $(\ell+1)$st one) is $O(Kn^{1+\frac{1}{\kappa\nu}})$.*

Proof: On this level of the recursion all the edges of small clusters of the previous level, i.e., clusters of $\mathcal{C}_\ell^L$, are inserted into the spanner. Indeed, the size of these clusters is

$$MS(\mathcal{C}_\ell^L) \le n^{(1-\nu)^{\log_{1/(1-\nu)} \log_K n}} = n^{1/\log_K n} = K \ .$$

Hence, number of edges taken in each such cluster C into the spanner is $K \cdot |C|$. Hence, on the $(\ell+1)$st level of the recursion, the total number of edges that are inserted into the spanner is at most

$$\sum_{C \in \mathcal{C}_\ell^L} K \cdot |C| = K \cdot s(\mathcal{C}_\ell^L) \le K \cdot n^{1+\frac{1}{\kappa\nu}} \ . \quad \blacksquare \tag{5}$$

Since all the edges that are added to the output set of the algorithm (the spanner) are inserted there either at step 2b or 2d of some invocation of Algorithm *Recur_Spanner*, we conclude the following.

Corollary 2.7 *The edge set H returned by the algorithm has size at most $O((K + D^{\ell+1}) n^{1+\frac{1}{\kappa\nu}})$.*

2.3 Stretch Analysis

We next analyze the stretch properties of the edge set H returned by Algorithm *Recur_Spanner*, presented at Section 2.1.

Let us introduce some definitions. For a partial cover $\mathcal{C}$, the node $u \in V$ is called *$\mathcal{C}$-clustered* if there exists a cluster $C \in \mathcal{C}$ such that $u \in C$. Let $\mathcal{C}' \subseteq \mathcal{C}$ be some subset of $\mathcal{C}$. Then the path P is *$\mathcal{C}'$-clustered with respect to $\mathcal{C}$* if all its nodes are $\mathcal{C}'$-clustered, and furthermore, all its nodes except maybe its endpoints are not $(\mathcal{C} \setminus \mathcal{C}')$-clustered.

Next, we present an intuitive description of the stretch analysis. Consider a pair of nodes $u, w \in V$ and one of the shortest paths $P_{u,w} = (u = u_0, u_1, \ldots, u_{d_G(u,w)} = w)$ between them. It is convenient to visualize the path $P_{u,w}$ as going from left to right, where u is the leftmost and w is the rightmost nodes. We partition the path $P_{u,w}$ into segments $(u_0, \ldots, u_{D^{\ell+1}})$, $(u_{D^{\ell+1}}, \ldots, u_{2D^{\ell+1}}), \ldots$, $(u_{aD^{\ell+1}}, \ldots, u_{d_G(u,w)})$, where $a = \lfloor d_G(u,w)/(D^{\ell+1}) \rfloor$. All these segments are of length precisely $D^{\ell+1}$, except the last, whose length is at most $D^{\ell+1}$. Consider an arbitrary segment $P' = (u'_0, \ldots, u'_t)$, $t \leq D^{\ell+1}$. Let u'_i (respectively, u'_j) be the leftmost (resp., rightmost) $\mathcal{C}_1^H$-clustered node in P'. Consider the set $\mathcal{S}_i$ of clusters of $C \in \mathcal{C}_1^H$ such that $u'_i \in C$. Since u'_i is $\mathcal{C}_1^H$-clustered node, the set $\mathcal{S}_i$ is not empty. Pick an arbitrary cluster $C'_i \in \mathcal{S}_i$. Analogously, pick an arbitrary cluster $C'_j \in \mathcal{C}_1^H$ that contains the node u'_j. Note that the distance between C'_i and C'_j is at most $D^{\ell+1}$. Hence, by step 2d of Algorithm *Recur_Spanner*, there exists a path of length $d_G(C'_i, C'_j)$ between them in the spanner H. Also by step 2b of Algorithm *Recur_Spanner*, H contains BFS spanning trees for C'_i and C'_j. It follows that the distance between u'_i, u'_j in H is greater than the distance between them in G by at most a relatively small additive term (roughly, the sum of diameters of C'_i and C'_j).

Therefore, it remains to consider only the "tails" of the segment P', i.e., the sub-paths $(u'_0, \ldots, u'_i)$ and $(u'_j, \ldots, u'_t)$. We observe that both these sub-paths are $\mathcal{C}_1^L$-clustered. (Note that although any cluster of $\mathcal{C}_1$ is either in $\mathcal{C}_1^L$ or in $\mathcal{C}_1^H$, this is not the case for the nodes. For example, the nodes u'_i and u'_j are both $\mathcal{C}_1^L$ and $\mathcal{C}_1^H$-clustered.) Partition each of these sub-paths into subsegments of length at most D^ℓ. Consider arbitrary subsegment $P'' = (u''_0, \ldots, u''_p)$, $p \leq D^\ell$. Observe that at least one among the two nodes u''_0 and u''_p is not $\mathcal{C}_1^H$-clustered, since otherwise it would contradict either the assumption that u'_i is the leftmost $\mathcal{C}_1^H$-clustered node or the assumption that u'_j is the rightmost one. Recall that by step 2b of Algorithm *Recur_Spanner*, $\mathcal{C}_1$ is a (κ, D^ℓ)-cover. Hence by property 5 of (κ, D^ℓ)-cover (see Def. 2.1) and since $d_G(u''_0, u''_p) = p \leq D^\ell$, it follows that there exists a cluster $C'' \in \mathcal{C}_1^L$ that contains u''_0, u''_p as well as some shortest path (of length at most p) between these two nodes. Note that this path is not necessarily P''.

Recall that step 2f of Algorithm *Recur_Spanner* applies the algorithm recursively to all clusters of $\mathcal{C}_1^L$ and, in particular, to C''; intuitively, this ensures that the spanner H contains a path between u''_0 and u''_p of length greater than p by only some small constant term.

The formal argument that bounds $d_H(u,w)$ in terms of $d_G(u,w)$ for any pair of nodes $u, w \in V$ is presented below.

Lemma 2.8 *Let $G = (V, E)$ be a graph and $\kappa = 1, 2, \ldots$, ν be a real constant $0 < \nu \leq 1/2 - 1/\kappa$, $D, K = 1, 2, \ldots$. Let $\ell = \ell(G) = \lceil \log_{1/(1-\nu)} \log_K |V| \rceil$. Let $u, w \in V$ be a pair of nodes at distance $d_G(u,w) \leq D^{\ell+1}$. Let H be the spanner constructed by Algorithm Recur_Spanner applied to (G, κ, ν, D, K). Then*

$$d_H(u,w) \leq d_G(u,w)\Big(1 + \sum_{i=0}^{\ell-2} \frac{2c_0\kappa}{D^{i+2}} \sum_{j=0}^{i} 2^j D^{i+1-j}\Big) + 2c_0\kappa \sum_{j=0}^{\ell-1} 2^j D^{\ell-j} \ , \tag{6}$$

for c_0 as in Def. 2.1.

Proof: The proof is by induction on ℓ (or, implicitly on $|V|$). The base of induction is obvious, since $\ell = 0$ means that $|V| \le K$ and thus all the edges of G are taken into the spanner. Thus, $d_H(u,w) = d_G(u,w)$ for any pair of nodes $u, w \in V$.

For the induction step, suppose that the assertion of the lemma holds for some $\ell \ge 0$. Consider a graph $G = (V, E)$ with $\ell + 1 = \lceil \log_{1/(1-\nu)} \log_K |V| \rceil$. Algorithm *Recur_Spanner* forms a $(\kappa, D^{\ell+1})$-cover $\mathcal{C}$ for the graph G at step 2b.

Consider a pair of nodes $u, w \in V$ such that $d_G(u,w) \le D^{\ell+2}$ and let P be one of the shortest paths between them. Let u_i (resp., u_j) be the leftmost (resp., rightmost) $\mathcal{C}^H$-clustered node in P. It follows that the paths $(u, \ldots, u_i)$ and $(u_j, \ldots, w)$ are $\mathcal{C}^L$-clustered with respect to $\mathcal{C}$. Let C_i (resp., C_j) be some cluster of $\mathcal{C}^H$ that contains u_i (resp., u_j). As argued above, $d_H(C_i, C_j) = d_G(C_i, C_j)$. Consider the path of length $d_H(C_i, C_j)$ in H and let $z_i \in C_i$ and $z_j \in C_j$ be its two endpoints. By step 2b of Algorithm *Recur_Spanner*, H contains the BFS spanning trees of C_i and C_j. Hence, $d_H(u_i, z_i) \le Diam(C_i)$ and $d_H(u_j, z_j) \le Diam(C_j)$. Also, $Diam(C_i), Diam(C_j) \le Diam(\mathcal{C}) = 2\kappa D^{\ell+1}$. Thus,

$$d_H(u,w) \le d_H(u,u_i) + d_H(u_j,w) + d_G(u_i,u_j) + 2c_0\kappa D^{\ell+1} \ . \tag{7}$$

Consider the path $(u, \ldots, u_i)$. Partition it into segments of length at most $D^{\ell+1}$ each. Consider one such segment $P' = (u', \ldots, w')$. The distance between u' and w' is at most $D^{\ell+1}$, and so there exists a cluster $C \in \mathcal{C}$ that contains both nodes. Since the path $(u, \ldots, u_i)$ is $\mathcal{C}^L$-clustered with respect to $\mathcal{C}$, it follows that $C \in \mathcal{C}^L$. Furthermore, $d_{(C,E(C))}(u', w') = d_G(u', w') \le D^{\ell+1}$. Since the cluster C belongs to $\mathcal{C}^L$, it follows that $|C| \le |V|^{1-\nu}$, and so $\lceil \log_{1/(1-\nu)} \log_K |C| \rceil \le \lceil \log_{1/(1-\nu)} \log_K |V| \rceil - 1 = \ell$.

Hence, the induction hypothesis is applicable to C. Let $d_G(u, u_i) = a \cdot D^{\ell+1} + b$, with $b \le D^{\ell+1}$. Then

$$\begin{aligned}
d_H(u,u_i) &\le a(D^{\ell+1}(1 + \sum_{i=0}^{\ell-2} \frac{2c_0\kappa}{D^{i+2}} \sum_{j=0}^{i} 2^j D^{i+1-j}) + 2c_0\kappa \sum_{j=0}^{\ell-1} 2^j D^{\ell-j}) \\
&+ b(1 + \sum_{i=0}^{\ell-2} \frac{2c_0\kappa}{D^{i+2}} \sum_{j=0}^{i} 2^j D^{i+1-j}) + 2c_0\kappa \sum_{j=0}^{\ell-1} 2^j D^{\ell-j} \\
&= d_G(u,u_i)(1 + \sum_{i=0}^{\ell-2} \frac{2c_0\kappa}{D^{i+2}} \sum_{j=0}^{i} 2^j D^{i+1-j} + \frac{2c_0\kappa}{D^{\ell+1}} \sum_{j=0}^{\ell-1} 2^j D^{\ell-j}) + 2c_0\kappa \sum_{j=0}^{\ell-1} 2^j D^{\ell-j} \\
&= d_G(u,u_i)(1 + \sum_{i=0}^{\ell-1} \frac{2c_0\kappa}{D^{i+2}} \sum_{j=0}^{i} 2^j D^{i+1-j}) + 2c_0\kappa \sum_{j=0}^{\ell-1} 2^j D^{\ell-j} \ .
\end{aligned}$$

An analogous formula follows for the pair of nodes u_j, w from the same considerations. Hence, by (7),

$$\begin{aligned}
d_H(u,w) &\le d_G(u,w)(1 + \sum_{i=0}^{\ell-1} \frac{2c_0\kappa}{D^{i+2}} \sum_{j=0}^{i} 2^j D^{i+1-j}) + 2 \cdot 2c_0\kappa \sum_{j=0}^{\ell-1} 2^j D^{\ell-j} + 2c_0\kappa D^{\ell+1} \\
&= d_G(u,w)(1 + \sum_{i=0}^{\ell-1} \frac{2c_0\kappa}{D^{i+2}} \sum_{j=0}^{i} 2^j D^{i+1-j}) + 2c_0\kappa \sum_{j=0}^{\ell} 2^j D^{\ell+1-j} \ . \qquad \blacksquare
\end{aligned}$$

Next, consider a pair of nodes $u, w \in V$ located at an arbitrary distance from one another. Then, by considering one of the shortest paths between them in G, dividing it into segments of length at most $D^{\ell+1}$ and applying the statement of Lemma 2.8 to each segment separately, it is easy to obtain the following bound.

Corollary 2.9 *Let $G=(V,E)$ be a graph and let $\kappa=1,2,\ldots$, ν be a real constant $0<\nu\le 1/2-1/\kappa$, $D,K=1,2,\ldots$, and $\ell=\ell(G)=\lceil \log_{1/(1-\nu)}\log_K|V|\rceil$. Let H be the spanner constructed by Algorithm Recur_Spanner invoked on (G,κ,ν,D,K). Then for any pair of nodes $u,w\in V$,*

$$d_H(u,w)\le d_G(u,w)(1+\sum_{i=0}^{\ell-1}\frac{2c_0\kappa}{D^{i+2}}\sum_{j=0}^{i}2^jD^{i+1-j})+2c_0\kappa\sum_{j=0}^{\ell-1}2^jD^{\ell-j}\ , \tag{8}$$

where c_0 is the constant defined in Definition 2.1.

Next, by computing the sums involved in inequality (8) we get the following result.

Corollary 2.10 *Let $G=(V,E)$ be an n-vertex graph, $\kappa=1,2,\ldots$, $0<\nu\le 1/2-1/\kappa$, $D=2,3,\ldots$, $K=1,2,\ldots$. Let $\ell=\ell(G)=\lceil\log_{1/(1-\nu)}\log_K n\rceil$. Let H be the spanner constructed by Algorithm Recur_Spanner invoked on (G,κ,ν,D,K). Then H is an $(1+\frac{4c_0\kappa\ell}{D},4c_0\kappa D^{\ell})$-spanner, for c_0 as in Def. 2.1.*

Proof: Note that

$$\sum_{j=0}^{i}2^jD^{i+1-j}=D^{i+1}\sum_{j=0}^{i}(2/D)^j=D^{i+1}\frac{1-(2/D)^{i+1}}{1-2/D}\le 2D^{i+1}\ ,$$

for $D\ge 2$. Hence,

$$\sum_{i=0}^{\ell-1}\frac{2c_0\kappa}{D^{i+2}}\sum_{j=0}^{i}2^jD^{i+1-j}\le 4c_0\kappa\ell/D\ .$$

It follows that the multiplicative term is at most $1+4c_0\kappa\ell/D$, and the additive term is at most $4c_0\kappa D^{\ell}$. ∎

2.4 Running Time

In this section we analyze the running time of Algorithm *Recur_Spanner*.

First, let us evaluate the running time required for constructing (κ,D^{ℓ})-covers on step 2b of Algorithm *Recur_Spanner* on different levels of the recursion. Recall that the running time of constructing a single (κ,W)-cover for any W on an n-vertex graph with edge set E is $O(|E|n^{1/\kappa})$ (cf. [1]). Note that the running time does not depend on W. Recall that for a cover $\mathcal{C}'$ of a graph $G=(V,E)$, $e(\mathcal{C}')=\sum_{C\in\mathcal{C}'}|E(C)|$ (see Definition 2.1). Hence, the overall running time of constructing all the covers through the algorithm is at most

$$|E|n^{1/\kappa}+\sum_{i=1}^{\ell}\sum_{C\in\mathcal{C}_i^L}|E(C)||C|^{1/\kappa}=|E|n^{1/\kappa}+\sum_{i=1}^{\ell}MS(\mathcal{C}_i^L)^{1/\kappa}e(\mathcal{C}_i^L)\ . \tag{9}$$

For evaluating this expression we prove the following lemma.

Lemma 2.11 *For any $i=1,2,\ldots,\ell$, $e(\mathcal{C}_i^L)\le|E|n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}$.*

Proof: The proof is by induction on i.

The induction base follows from property 4 of Definition 2.1. Specifically,

$$e(\mathcal{C}_1^L)\le e(\mathcal{C}_1)=|E|n^{1/\kappa}\ .$$

The induction step follows from the same property of the covers that are built through the algorithm, from Lemma 2.2, and from the induction hypothesis. Specifically,

$$\begin{aligned} e(\mathcal{C}_{i+1}^L) &= \sum_{C\in\mathcal{C}_{i+1}^L} |E(C)| \le \sum_{C\in\mathcal{C}_i^L}\sum_{C'\in\mathcal{C}(C)} |E(C')| = \sum_{C\in\mathcal{C}_i^L} e(\mathcal{C}(C)) \le \sum_{C\in\mathcal{C}_i^L} |E(C)||C|^{1/\kappa} \\ &\le MS(\mathcal{C}_i^L)^{1/\kappa} e(\mathcal{C}_i^L) \le n^{(1-\nu)^i/\kappa} n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}|E| = n^{\frac{1}{\kappa}\frac{1-(1-\nu)^{i+1}}{\nu}}|E| \ . \quad \blacksquare \end{aligned}$$

Lemma 2.11 enables to get an easy upper bound on expression (9), yielding the following corollary.

Corollary 2.12 *For any sufficiently large integer n, any n-vertex graph G, $\kappa = 1,2,\ldots$, $0 < \nu \le 1/2 - 1/\kappa$, $D, K = 1,2,\ldots$, when invoking Algorithm Recur_Spanner on $(G = (V,E),$ $\kappa,\nu,D,K)$, the overall running time of constructing all the covers is $O(|E|n^{\frac{1}{\kappa\nu}})$.*

Proof: By inequality (9) and Lemma 2.11, the running time of constructing all the covers is

$$|E|n^{1/\kappa} + \sum_{i=1}^{\ell} e(\mathcal{C}_i^L)\cdot MS(\mathcal{C}_i^L)^{1/\kappa} = |E|n^{1/\kappa} + \sum_{i=1}^{\ell} |E|n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}\cdot n^{(1-\nu)^i/\kappa} = O(n^{\frac{1}{\kappa\nu}}|E|) \ . \quad \blacksquare$$

Let us next analyze the time required for interconnecting big clusters (i.e., step 2d of Algorithm *Recur_Spanner*) at different levels of the recursion.

Lemma 2.13 *For any graph $G = (V,E)$, $\kappa = 1,2,\ldots$, $0 < \nu \le 1/2 - 1/\kappa$, $D, K = 1,2,\ldots$, when invoking Algorithm Recur_Spanner on $(G = (V,E),\kappa,\nu,D,K)$, the overall time of interconnecting big clusters by short paths (steps 2d of Algorithm Recur_Spanner on different levels of the recursion) is $O(n^{\frac{1}{\kappa\nu}+\nu}|E|)$.*

Proof: First, note that for any cluster C, the running time of step 2d of Algorithm *Recur_Spanner* invoked on the five-tuple $((C,E(C)),\kappa,\nu,D,K)$ is $\tau(C) = |E(C)||C|^{\frac{1}{\kappa}+\nu}$. Thus, at the first level of the recursion the running time of this step is $O(n^{\frac{1}{\kappa}+\nu}|E|)$. For $i = 1,2,\ldots,\ell$, the running time of step 2d in ith level of the recursion (denote $\mathcal{C}_0^L = V$) is at most

$$\sum_{C\in\mathcal{C}_{i-1}^L} \tau(C) = \sum_{C\in\mathcal{C}_{i-1}^L} |C|^{\frac{1}{\kappa}+\nu}|E(C)| \le (MS(\mathcal{C}_{i-1}^L))^{\frac{1}{\kappa}+\nu} e(\mathcal{C}_{i-1}^L) \le n^{(1-\nu)^{i-1}(\frac{1}{\kappa}+\nu)}|E|n^{\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}} \ .$$

Note that the exponent of n in the last expression is at most $\frac{1}{\kappa\nu} + (1-\nu)^{i-1}\nu$. Hence, the overall running time of steps 2d Algorithm *Recur_Spanner* on the different levels of the recursion is $O(|E|n^{\frac{1}{\kappa\nu}+\nu})$. Hence, the lemma follows. $\blacksquare$

Furthermore, note that the bound on the running time in Lemma 2.13 dominates the one in Corollary 2.12. Thus,

Corollary 2.14 *The overall running time of Algorithm Recur_Spanner invoked on (G,κ,ν,D,K) for an n-vertex graph G, $\kappa = 1,2,\ldots$, $0 < \nu \le 1/2 - 1/\kappa$, $D, K = 1,2,\ldots$ is $O(n^{\frac{1}{\kappa\nu}+\nu}|E|)$.*

We are now ready to summarize the properties of the sequential construction by the following theorem, which follows by substituting appropriate values to the parameters κ, ν, D and K, and using Corollaries 2.7, 2.10 and 2.14.

Theorem 2.15 *For any sufficiently large integer n, any unweighted n-vertex graph $G=(V,E)$, and any three real numbers $0<\epsilon,\rho,\zeta<1$ such that $\zeta/2+1/3>\rho>\zeta/2$, ρ,ζ are constant, and $\epsilon=\Omega(\frac{1}{\mathrm{polylog}\ n})$, invoking Algorithm Recur_Spanner on $(G,\kappa=\frac{2}{\zeta(\rho-\zeta/2)},\nu=\rho-\zeta/2,$ $D=\frac{8c_0}{\epsilon\zeta(\rho-\zeta/2)}\lceil\log_{1-(\rho-\zeta/2)}\zeta/2\rceil, K=n^{\zeta/2})$ produces a $(1+\epsilon,\beta(\zeta,\rho,\epsilon))$-spanner of size $O(n^{1+\zeta})$. The running time of Algorithm Recur_Spanner on this input is $O(|E|n^{\rho})$. $\beta(\zeta,\rho,\epsilon)$ is constant whenever ζ, ρ and ϵ are, and is defined by*

$$\beta(\zeta,\rho,\epsilon)=\left(\frac{8c_0}{\epsilon\zeta(\rho-\zeta/2)}\right)^{\lceil\log_{\frac{1}{1-(\rho-\zeta/2)}}\zeta/2\rceil+1}\lceil\log_{1-(\rho-\zeta/2)}\zeta/2\rceil^{\lceil\log_{1-(\rho-\zeta/2)}\zeta/2\rceil}, \qquad (10)$$

for c_0 as in Definition 2.1.

Proof: First, note that

$$\ell=\lceil\log_{\frac{1}{1-\nu}}\log_{n^{\zeta/2}}n\rceil=\lceil\log_{1-(\rho-\zeta/2)}\zeta/2\rceil .$$

We observe that for constant ζ and ρ, and for $\epsilon=\Omega(\frac{1}{\mathrm{polylog}\ n})$, $K=n^{\zeta/2}$ is greater than $D^{\ell+1}$ for sufficiently large n. Thus, by Corollary 2.7,

$$|H|=O((K+D^{\ell+1})n^{1+\frac{1}{\kappa\nu}})=O(n^{1+\zeta}) .$$

By Corollary 2.10, H is an (α,β)-spanner, where $\alpha=1+\frac{4c_0\kappa\ell}{D}=1+\epsilon$, and $\beta=c_0\kappa D^{\ell}\le\beta(\zeta,\rho,\epsilon)$.

Finally, the running time of Algorithm *Recur_Spanner* invoked on the five-tuple of inputs as above is $O(|E|n^{\frac{1}{\kappa\nu}+\nu})=O(|E|n^{\zeta/2+\rho-\zeta/2})=O(|E|n^{\rho})$, by Corollary 2.14. ∎

It is instructive to compare our result with the previously known construction of $(1+\epsilon,\beta')$-spanner due to [20]. It was shown there that for any unweighted graph G and for any $0<\zeta,\rho,\epsilon<1$, $(1+\epsilon,\beta'(\zeta,\rho,\epsilon))$-spanner of G with $O(n^{1+\zeta})$ edges can be constructed in $\tilde{O}(n^{2+\rho})$ time, where $\beta'(\zeta,\rho,\epsilon)=(1/\zeta)^{\log\log(1/\zeta)-\log\epsilon-\log\rho}$.

Our result is faster ($O(|E|n^{\rho})$ time instead of $\tilde{O}(n^{2+\rho})$), whenever $|E|<<n^2$. However, the additive term β that we achieve is slightly worse than the additive term β' achieved in [20]. Specifically, β depends on ζ as $(1/\zeta)^{\log 1/\zeta}$, on $(\rho-\zeta/2)$ as $(\frac{1}{\rho-\zeta/2})^{\frac{1}{\rho-\zeta/2}}$ and inverse-polynomially on ϵ. β' depends on ζ as $(1/\zeta)^{\log\log 1/\zeta}$, and depends inverse-polynomially both on ρ and ϵ. Nevertheless, the crucial point is that both β and β' are constant, whenever ζ, ρ and ϵ are. Recall that the additive term (β in our construction and β' in that of [20]) reflects the following threshold. The distance in the spanner between all the pairs of nodes that are located in the original graph at distance greater than the threshold, is at most by a factor of $(1+\epsilon)$ greater than the distance between them in the original graph. Hence, essentially, the tradeoff between our construction and that of [20] is that our construction is significantly faster, but the produced spanner approximates distances between pairs of nodes that are located at the original graph at distance greater than some constant, and in our construction this constant is greater than in that of [20].

Another advantage of our construction is that, unlike the construction of [20], our construction can be easily converted into an efficient distributed protocol, as we indeed do in Section 3.

2.5 Modified Variants of Algorithm *Recur_Spanner*

In this section we present several modifications of Algorithm *Recur_Spanner*. The first two modifications are used to generalize the algorithm to the weighted graphs. The third one is geared to make its conversion into a distributed protocol easier.

Consider a weighted graph $G = (V, E, \omega)$. Denote $\omega_{min} = \min\{\omega(e) \mid e \in E\}$ and $\omega_{max} = \max\{\omega(e) \mid e \in E\}$. Suppose, without loss of generality, that $\omega_{min} = 1$. Indeed, otherwise one can just normalize the weights of the graph by dividing the weight of all the edges by ω_{min}.

Consider the following modification. First, at step 2b the algorithm will form a $(\kappa, D^{\ell}\omega_{max})$-cover instead of a (κ, D^{ℓ})-cover. Secondly, at step 2d the algorithm will compare the distance between C_1 and C_2 with $D^{\ell+1}\omega_{max}$ instead of $D^{\ell+1}$. Let us refer the modified algorithm as *Algorithm Recur_Weighted*.

It is easy to see that these changes would not affect the running time analysis. However, it affects the size analysis, since when at step 2d the algorithm inserts a path into the spanner, this path might now consist of up to $D^{\ell+1}\omega_{max}$ edges, and not up to $D^{\ell+1}$ edges as previously. Hence, the size of the constructed spanner may now be up to $O(\omega_{max} n^{1+\zeta})$. Also, this modification increases by a factor of ω_{max} the diameter of the clusters constructed through the algorithm. Thus, the additive term grows now by a factor of ω_{max} too. To summarize,

Corollary 2.16 *For any sufficiently large integer n, any n-vertex weighted graph $G = (V, E, \omega)$, any $0 < \zeta, \rho, \epsilon < 1$ such that $\zeta/2 + 1/3 > \rho > \zeta/2$, ρ, ζ are constant and $\epsilon = \Omega(\frac{1}{\text{polylog }(n)})$, Algorithm Recur_Weighted invoked on the same five-tuple as in Theorem 2.15 produces a $(1+\epsilon, \beta(\zeta, \rho, \epsilon)\omega_{max})$-spanner with $O(\omega_{max} n^{1+\zeta})$ edges in time $O(n^{1+\rho})$, where $\beta(\zeta, \rho, \epsilon)$ is defined by (10).*

We remark that a similar adaptation to the weighted graphs is possible for the construction of [20], yielding a slower construction with slightly smaller value of constant $\beta(\zeta, \rho, \epsilon)$ (see the discussion after Theorem 2.15).

While the result of Corollary 2.16 is good for weighted graphs with $\omega_{max} = O(1)$, it is unsatisfactory for general weighted graphs. In order to cope with the case when ω_{max} is big, we introduce the following relaxed notion of spanner, called *Steiner spanner*. A weighted graph $G' = (V, H, \omega')$ is a Steiner (α, β)-spanner of a weighted graph $G = (V, E, \omega)$ with the same vertex set V, if for any pair of nodes u, w in V, $d_G(u, w) \le d_H(u, w) \le \alpha \cdot d_G(u, w) + \beta$. Note that unlike regular spanner, H is not necessarily a subset of E. Also, while for a weighted spanner H, for any edge $e \in H$, $\omega(e) = \omega'(e)$, it is not necessarily the case for a Steiner spanner.

Consider the following modification of Algorithm *Recur_Spanner*, that enables us to get rid of the factor ω_{max} in the size of the spanner, while compromising on getting Steiner spanner as an output. Apart of the modifications of steps 2b and 2d, we now also modify step 2(d)ii as follows.

"Let u, w be the endpoints of P. Set $e = (u, w)$, $\omega(e) = d_G(C_1, C_2) = d_G(u, w)$. Set $H_C \leftarrow H_C \cup \{e\}$."

Let us refer the modified algorithm as Algorithm *Recur_Steiner*. Consider the invocations of algorithms *Recur_Weighted* and *Recur_Steiner* on the same input. Note that for any path inserted by the former algorithm, the latter will insert only one edge. Thus, we conclude the following.

Corollary 2.17 *For any sufficiently large integer n, any n-vertex weighted graph $G = (V, E, \omega)$, any $0 < \zeta, \rho, \epsilon < 1$ such that $\zeta/2 + 1/3 > \rho > \zeta/2$, ρ, ζ are constant and $\epsilon = \Omega(\frac{1}{\text{polylog }(n)})$, Algorithm Recur_Steiner invoked on the same five-tuple as in Theorem 2.15 produces a Steiner $(1+\epsilon, \beta(\zeta, \rho, \epsilon)\omega_{max})$-spanner with $O(n^{1+\zeta})$ edges in time $O(n^{1+\rho})$, where $\beta(\zeta, \rho, \epsilon)$ is defined by (10).*

We remark that a result similar to this was implicitly (without using the notion of Steiner spanner or an equivalent notion) obtained in [16]. However, in construction of [16] β depends polylogarithmically on n even when ζ, ρ and ϵ are constant. Thus in [16] ϵ was set to $\theta(\frac{1}{\text{polylog }(n)})$, since it yields a multiplicative term of $(1 + \theta(\frac{1}{\text{polylog }(n)}))$, and the additive term is polylogarithmic in n anyway. Corollary 2.17, on the other hand, is an improvement over the result of [16], since setting $\epsilon = \theta(\frac{1}{\text{polylog }(n)})$ yields a slightly better result than that of [16]. Furthermore, in our construction, whenever ζ, ρ and ϵ are constant β is constant as well, instead of $\beta = \text{polylog}(n)$ in the construction of [16]. Finally, consider another modification of

Algorithm *Recur_Spanner*. This modification yields another algorithm for unweighted graphs, which we refer Algorithm *Recur_Spanner'*. This modified algorithm produces spanner with approximately the same (up to small constant values) ϵ, β and size parameters, as the ones of the spanner produced by Algorithm *Recur_Spanner*. However, this modification simplifies the analysis of the distributed implementation of Algorithm *Recur_Spanner*, that presented in Section 3 later on.

The modification is in step 2d of Algorithm *Recur_Spanner*, which now looks as follows.
For all pairs $C_1, C_2 \in \mathcal{C}^H$ **do**

1. Pick arbitrary $u_1 \in C_1$, $u_2 \in C_2$.
2. **If** $d_{G'}(u_1, u_2) \leq D^{\ell+1} + D^{\ell}\kappa$ **then**
   (a) Compute shortest path between u_1 and u_2 in G';
   (b) $H_C \leftarrow H_C \cup E(P)$;

Note that after this modification, exactly as in the original algorithm, the number of paths inserted to the spanner at step 2d is bounded by $|\mathcal{C}^H|^2$. However, our upper bound on the number of edges inserted to the spanner at this step is now slightly greater than in Algorithm *Recur_Spanner*, because the length of each path is now at most $D^{\ell+1} + D^{\ell}\kappa$ (and previously it was up to $D^{\ell+1}$). We observe that this change would not significantly affect the upper bound on the number of edges inserted to the spanner in this step if we set $D \geq \kappa$, as we anyway did in Theorem 2.15. Indeed, setting $D \geq \kappa$ guarantees $D^{\ell+1} + D^{\ell}\kappa \leq 2D^{\ell+1}$, and so the size the spanner output by Algorithm *Recur_Spanner'* in this case is greater by at most by a factor of two than the size of the spanner output by Algorithm *Recur_Spanner* invoked with the same parameters.

It is also easy to verify that that this modification of the algorithm does not affect significantly the stretch analysis. Specifically, there is only one minor change in the bound on $d_H(u, w)$ inequality (6). Instead of 4κ appearing twice in the bound inequality (6), there will appear 8κ. The proof of the modified lemma is along the same lines as the proof of Lemma 2.8.

The modification of the induction step of Lemma 2.8 that is required for its proof with respect to Algorithm *Recur_Spanner'* is as follows.

Consider the $\mathcal{C}^H$ clustered nodes u_i and u_j such that $d_G(u_i, u_j) \leq D^{\ell+2}$ and the clusters $C_i, C_j \in \mathcal{C}^H$ such that $u_i \in C_i$, $u_j \in C_j$. For Algorithm *Recur_Spanner* it holds that for such C_i and C_j, the distance between them in H is equal to the one in G, i.e., $d_G(C_i, C_j) = d_H(C_i, C_j)$. It is not necessarily true for Algorithm *Recur_Spanner'*. However, we show that $d_H(C_i, C_j) \leq d_G(C_i, C_j) + 4\kappa D^{\ell+1}$. To see this, consider the modified step 2d of Algorithm *Recur_Spanner'* when it was applied to the pair of clusters C_i, C_j. There were picked arbitrary nodes $z_i \in C_i$, $z_j \in C_j$. Recall that $d_G(C_i, C_j) \leq D^{\ell+2}$. Hence, $d_G(z_i, z_j) \leq d_G(C_i, C_j) + 4\kappa D^{\ell+1}$. Hence, one of the shortest paths between z_i and z_j is inserted to the spanner, ensuring that $d_H(C_i, C_j) \leq d_G(C_i, C_j) + 4\kappa D^{\ell+1}$, as required. Finally, note that the latter implies that $d_H(u_i, u_j) \leq d_G(u_i, u_j) + 8\kappa D^{\ell+1}$, as required.

The change in Lemma 2.8 is propagated to Corollaries 2.9 and 2.10, increasing some of the constants that appear there by a factor of 2.

2.6 Computing Almost Shortest Paths

In this section we show how our fast construction of $(1+\epsilon, \beta)$-spanner directly implies better than currently known results concerning the problem of computing almost shortest paths and distance estimates between pairs of nodes in a graph.

For any weighted graph $G = (V, E, \omega)$, and any path $P \subseteq E$, denote $L(P) = \sum_{e \in P} \omega(e)$. In an unweighted graph $\omega(e) = 1$ for any edge $e \in E$, and so $L(P) = |P|$.

Theorem 2.18 *For any sufficiently large integer n, any vertex set V of size n, any real numbers $0 < \zeta, \rho, \epsilon < 1$ such that $\zeta/2 + 1/3 > \rho > \zeta/2$, ρ, ζ are constant and $\epsilon = \Omega(\frac{1}{\text{polylog }(n)})$, any subset $S \subseteq V$ of s vertices, the following two statements hold (for $\beta(\zeta, \rho, \epsilon)$ as defined by (10)).*

1. *For any unweighted graph $G = (V, E)$, the paths $P_{u,w} \subseteq E$ between all pairs of nodes $\{u, w\} \in S \times V$ can be computed in time $O(|E|n^{\rho} + sn^{1+\zeta})$. These paths satisfy*
   (a) $L(P_{u,w}) \le (1 + \epsilon)d_G(u, w) + \beta(\zeta, \rho, \epsilon)$.
   (b) $|\bigcup_{\{u,w\}\cap S\neq\emptyset} P_{u,w}| = O(n^{1+\zeta})$.
2. *For any weighted graph $G = (V, E, \omega)$, the paths $P_{u,w} \subseteq E$ (respectively, $P'_{u,w}$, that are not necessarily contained in E) between all pairs of nodes $\{u, w\} \in S \times V$ can be computed in time $O(|E|n^{\rho} + s \cdot \omega_{max}n^{1+\zeta})$ (resp., $O(|E|n^{\rho} + sn^{1+\zeta})$). These paths satisfy*
   (a) $L(P_{u,w}), L(P'_{u,w}) \le (1 + \epsilon)d_G(u, w) + \omega_{max}\beta(\zeta, \rho, \epsilon)$.
   (b) $|\bigcup_{\{u,w\}\cap S\neq\emptyset} P_{u,w}| = O(\omega_{max}n^{1+\zeta})$.
   (c) $|\bigcup_{\{u,w\}\cap S\neq\emptyset} P'_{u,w}| = O(n^{1+\zeta})$.

Proof: (of statement 1) The algorithm operates as follows. First, use Algorithm *Recur_Spanner* to construct a $(1 + \epsilon, \beta(\zeta, \rho, \epsilon))$-spanner H for the graph. Next, perform s BFS searches initiated from each node $v \in S$ over the constructed spanner.

Note that all the paths $P_{u,w}$ that were found by the search satisfy $L(P_{u,w}) = d_H(u, w)$. By definition of $(1 + \epsilon, \beta(\zeta, \rho, \epsilon))$-spanner, $d_H(u, w) \le (1 + \epsilon)d_G(u, w) + \beta(\zeta, \rho, \epsilon)$, implying 1a.

Note also that for any $u \in S$, $w \in V$, $P_{u,w} \subseteq H$. Hence, $\bigcup_{\{u,w\}\cap S\neq\emptyset\}} P_{u,w} \subseteq H$. By Theorem 2.15, $|H| = O(n^{1+\zeta})$, implying 1b.

Finally, the running time of the algorithm consists of two parts. The first is constructing the spanner, and the second is performing the s searches over it. By Theorem 2.15, the first part consumes $O(|E|n^{\rho})$ time. Regarding the second part, every BFS search consumes $O(|H|) = O(n^{1+\zeta})$ time. Hence, s BFS searches consume $O(sn^{1+\zeta})$ time, completing the proof of statement 1.

The only change is that Corollaries 2.16 and 2.17 are used instead of Theorem 2.15. ▮

The part of statement 2 of Theorem 2.18 that considers the paths $P'_{u,w}$ is, in fact, an improvement of the corresponding result from [16]. The difference is that in the result of [16], β depends at least polylogarithmically on n even when ζ, ρ and ϵ are constant. In our result, β is constant whenever ζ, ρ and ϵ are.

Statement 1 of Theorem 2.18 should be compared with the following result from [17]. For any $0 < \zeta < 1$, the APASP problem on unweighted n-vertex graphs with error $(1, \lceil 1/\zeta \rceil)$ can be solved in time $\tilde{O}(n^{2+\zeta})$. This result corresponds to the case $s = n$ of statement 1 of Theorem 2.18, and, to the best of our knowledge, no generalization of it to $s \le n$ is known. Our result is significantly faster whenever $s < n$ and $|E| < n^2$. Furthermore, it has the advantage that the paths $P_{u,w}$ altogether use only $O(n^{1+\zeta})$ edges of the original graph (see Theorem 2.18(1b)). Another advantage is that our algorithm can be easily converted into a distributed protocol, as we indeed do in Section 3, whereas no distributed counter apart of the algorithm of [17] is known. However, the result of [17] provides a better approximation of distances. Specifically, its multiplicative term is 1 instead of $(1+\epsilon)$ for any $\epsilon > 0$ in our construction, and its additive term is linear in $1/\zeta$, instead of super-polynomial in $1/\zeta$ in our construction ($(1/\zeta)^{\log(1/\zeta)}$).

3 Distributed Implementation

In this section we present a distributed implementation of Algorithm *Recur_Spanner*, that was presented and analyzed in the sequential model in Section 2. This algorithm constructs a $(1+\epsilon, \beta)$-spanner with ϵ, β

and size parameters similar to those listed in Theorem 2.15. Next, we will use this distributed construction of spanner to present an efficient (in terms of time and communication complexities) distributed protocol for computing almost shortest paths.

3.1 Overview of the Protocol

In this section we provide a high-level description of the distributed counter apart of Algorithm *Recur_Spanner* (or, more precisely, *Recur_Spanner'*, see Section 2.5).

The protocol, henceforth referred to as *Protocol Distrib_Spanner*, starts with computing a BFS spanning tree T of the entire graph. Note that after this computation, no node actually knows all the edges of the tree T, but instead, every node has a list of incident to it edges of T.

This tree is later used for traversals over the graph. Informally, the execution of Protocol *Distrib_Spanner* can be divided to periods through which different nodes become, in some sense, the "centers" of the execution. The traversals over the tree T are needed, therefore, for delivering the "relay baton" between the different nodes.

After constructing the tree T, Protocol *Distrib_Spanner* constructs a (κ, D^{ℓ})-cover $\mathcal{C}$ of the entire graph, where $\ell = \lceil \log_{1/(1-\nu)} \log_K n \rceil$, and κ, D, ν and K are parameters of the protocol (for more details about this parameters, see Section 2.1). As for the spanning tree construction, after the construction of the cover no node actually knows the description of the entire cover, but instead, every node knows the list of clusters to which it belongs. Also, for each such cluster the node knows the list of edges incident to it that have another endpoint in the same cluster, and also the list of edges incident to it that belong to the BFS spanning tree of the cluster. At the end of the construction, every node knows also for every cluster to which it belongs, whether this cluster is "big" or "small", i.e., whether its size is at least $n^{1-\nu}$ or not.

Through Protocol *Distrib_Spanner*, the spanner is constructed in an incremental manner. Every node keeps an initially empty set H of edges that are incident to the node and belong to the spanner. Through the protocol some edges are inserted into these lists in different nodes. In particular, when the cover $\mathcal{C}$ is constructed, every node v inserts to its set H the incident to v edges of the spanning trees of the clusters that contain v.

After constructing the BFS spanning tree T and the cover $\mathcal{C}$, Protocol *Distrib_Spanner* traverses the tree T in a depth-first order. Upon reaching any node v of the tree, if this node is the root of some cluster $C \in \mathcal{C}^H$, which still was not interconnected to the other clusters of $\mathcal{C}^H$ that are "relatively close" to it (i.e., those at distance at most $D^{\ell+1}$ from it), the protocol inserts to the spanner all the edges of the shortest paths that connect C to these clusters. Note that on each level of recursion, every cluster initiates at most one interconnection process.

Finally, when the node v finishes interconnecting clusters of $\mathcal{C}^H$ that contain it to other clusters, (or, if they were already interconnected; or, if no such clusters exist), then for every cluster C of $\mathcal{C}^L$ that contains v, the node recursively initiates the protocol on this cluster. We observe that this happens sequentially, i.e., the node waits until the distributed protocol finishes to build the spanner inside this cluster (and also actively participates in this process, since the node belongs to this cluster too), and then turns to another cluster C' of $\mathcal{C}^L$ that contains it and initiates the protocol on this cluster.

We remark, that all the invocations of procedures in Protocol *Distrib_Spanner* are "critical sections", in the sense that the protocol does not proceed to its next step until it gets an acknowledgement that the previous step was completed.

3.2 Details of Protocol *Distrib_Spanner*

In this section a more detailed description of Protocol *Distrib_Spanner* is given.

We first describe the data structures used by every node in the graph. This data structure will contain a constant number of variables that keep single value, like κ, ν, K, D, n and some other. Also every node will keep a two-dimensional array *Local_Data* of size $(\lceil \log_{1/(1-\nu)} \log_K n \rceil + 1) \times \kappa n^{1/\kappa}$. Every entry *Local_Data*$[i, j]$ of this array is a eight-tuple, which describes a cluster of ith level of the recursion, that contains the node v. The number of such clusters is up to $\kappa n^{1/\kappa}$. There is no specific order, in which the records of these clusters are inserted to the *Local_Data* array.

The record of a cluster, that is stored in an entry of *Local_Data* array, is an eight-tuple of the form $(cluster\_id, tree\_id, parent\_edge\_id, cluster\_size, tree\_edges, cluster\_edges, size\_bit, status)$. The *cluster_id* field stores the unique identity number of the cluster. During the protocol, whenever a new cluster is formed it is assigned a unique *cluster_id*. Assigning the unique identity numbers is a standard procedure that we will remain implicit in the formal description of the protocol. The field *tree_id* stores the identity of the BFS spanning tree of the cluster. The *cluster_size* field keeps the size of the cluster. Recall that the protocol recursively invokes itself on some clusters. Hence, inside these clusters new sub-clusters are formed. For each such sub-cluster there is a *parent cluster*, i.e., a cluster on which the protocol was invoked and this invocation, in particular, formed this sub-cluster. The *size_bit* specifies whether the cluster is "big" or "small", i.e., whether its size, in terms of number of nodes, is greater or smaller than the size of its parent cluster raised to the power of $1 - \nu$. In the former case its value is HIGH and in the latter it is LOW. The *status* field determines the status of the cluster with respect to the execution of the protocol. It may contain either value INVALID, meaning that no cluster is described by the record, or WAIT, or OVER. If the cluster is a small cluster, the value WAIT in its *status* field means that protocol still was not recursively invoked on this cluster, and OVER means that it was and this recursion invocation is over. For a big cluster, the value WAIT in its *status* field means that this cluster still was not interconnected to other "relatively close" to it (i.e., at distance at most $D^{\ell+1}$ from it) big clusters, and OVER means that it already was. It also may happen that the cluster size is smaller than some constant K that determines the threshold size below which the protocol inserts all the cluster edges to the spanner. For such a cluster, the value WAIT of the field *status* means that the edges of this cluster still were not inserted to the spanner, and the value OVER means the opposite. The field *tree_edges* contains the list of edges incident to the node that are contained in the spanning tree of the cluster. The field *parent_edge_id* specifies the identity of one of these edges whose other endpoint is the parent of the node in the tree. The field *cluster_edges* contains the list of edges incident to the node, whose another endpoint belongs to the cluster too.

Next, we describe Protocol *Distrib_Spanner* itself. The protocol starts when its initiator invokes Procedure *Initiate*. This procedure constructs a BFS spanning tree for the graph, broadcasts over the tree the parameters of the algorithm, and delivers the control to Procedure *Recurse*, that will be discussed below.

To construct the BFS spanning tree, Procedure *Initiate* uses Procedure *Constr_ST*. The latter gets as an input the identity of the cluster, for which it is required to build a spanning tree, and maximal depth of the tree. It produces the distributed description of the partial BFS spanning tree of depth not greater than its second parameter for the cluster. In particular, if the second parameter is ∞, it produces the BFS spanning tree for the entire cluster, no matter how big diameter the cluster has. Specifically, upon its completion every node of the cluster has a quadruple containing the identity of the tree *tree_id*, the set of tree edges that are incident to the node stored in the field *tree_edges*, the identity of the edge that belongs to the tree and leads to the parent of the node (*parent_edge_id* field), and, finally, the depth of the node in the tree (field *depth*). For the root of the tree, the field *parent_edge_id* is assigned the value $\perp$.

For broadcasting the values over the spanning tree that was formed by Procedure *Constr_ST*, Procedure *Initiate* uses Procedure *Broadcast_ST*. The latter is given as input the record describing the tree, and a message that it is required to broadcast. When the procedure is terminated all the nodes of the tree have received this message, and the initiator node was notified that the broadcast was completed. This procedure operates in the truly distributed manner, i.e., by sending in parallel the broadcast message to all the neighbors. We remark, that we will later use another variant of broadcast procedure, that we refer to as *Seq_Broadcast_ST*, that has the same functionality, but operates differently. This procedure will always have only one activity center which traverses the tree in some prespecified order. Only once finishing all the activities that were triggered by the broadcasted message in one node, this broadcast will proceed to the next node of the tree.

All the three procedures, *Constr_ST*, *Broadcast_ST* and *Seq_Broadcast_ST*, are rather standard (cf. [25]) and for the sake of brevity, we omit their formal description. We will also use the primitive *Send_Msg*. This primitive is used by a node to send a message to one of its neighbors. Naturally, the primitive has two parameters. The first is the identity of edge over which the message is sent, and the second is the message itself. The message also may contain several arguments.

The formal description of Procedure *Initiate* follows.

Procedure Initiate
Input: $\kappa = 1, 2, \ldots,\ 0 < \nu \leq 1/2 - 1/\kappa,\ D, K = 1, 2, \ldots$

1. $T \leftarrow Constr\_ST(id_G, \infty)$;
2. $Broadcast\_ST(T.tree\_id, SET(\kappa, \nu, D, K, n))$;
3. $Recurse(id_G, n, 1)$;

Upon Receival Message $SET(\kappa_0, \nu_0, D_0, K_0, n_0)$ **do**

1. $\kappa = \kappa_0;\ \nu = \nu_0;\ D = D_0;\ K = K_0,\ n = n_0;$
2. Assign appropriate values to the fields of the record $Local\_Data[0, id_G]$.

Next, we describe Procedure *Recurse*. This procedure is one of the central ingredients of the protocol. Its first parameter is the identity of the cluster C, for which it constructs a spanner. In particular, at the first invocation it is the identity of the entire graph. Note that the identity number is at most $n\lceil\log_{1/(1-\nu)} \log_K n\rceil$, since at each level of the recursion at most n clusters are formed, and there are $\lceil\log_{1/(1-\nu)} \log_K n\rceil$ levels of the recursion. The second input of Procedure *Recurse* is the size of C, and the third is the level of the recursion, which is later used by the nodes as one of the keys to their *Local_Data* array. The other key is the identity of C, that is given as the first parameter. Procedure *Recurse* starts from checking whether the size of C is not small enough. If it is, the message *INSERT_ALL* is broadcasted over the cluster C. Nodes that get this message, insert all the incident to them edges that are internal to the cluster into the spanner. (Compare this to step 1 of Algorithm *Recur_Spanner*.) Note that the broadcasted value of the variable *recur_level* is smaller by 1 than its current value. This is because the clusters that were formed by Procedure *Constr_Cover* are attached the current level of recursion, whereas the clusters over which the message *INSERT_ALL* is broadcasted were formed by invocation of Procedure *Constr_Cover* at the previous level of the recursion. If the size of C is not that small, then Procedure *Constr_Cover* is invoked. This procedure forms an internal cover inside C. Next, the message *TRAVERSE* is broadcasted. Procedure *Constr_Cover* and the procedure invoked by nodes that get a *TRAVERSE* message are described below. The formal description of Procedure *Recurse* follows.

Procedure *Recurse*
Input: $id_C = 0, 1, 2, \ldots,\ size_C = 1, 2, \ldots,\ recur\_level = 1, 2, \ldots, \lceil\log_{1/(1-\nu)} \log_K n\rceil$

1. $\ell \leftarrow \lceil \log_{1/(1-\nu)} \log_K(Local\_Data[recur\_level, id_C].cluster\_size) \rceil$;

2. (a) **If** $size_C \leq K$ **then**
$Broadcast\_ST(Local\_Data[recur\_level, id_C].tree\_id, INSERT\_ALL(recur\_level - 1, id_C))$;
else

   (b) i. $Constr\_Cover(id_C, size_C, \kappa, D^\ell, recur\_level)$;
      ii. $Seq\_Broadcast\_ST(Local\_Data[recur\_level, id_C].tree\_id,$
      $TRAVERSE(id_C, recur\_level, \ell))$;

Upon Receival Message $INSERT\_ALL(recur\_level, id_C)$ **do**
$H \leftarrow H \cup Local\_Data[recur\_level, id_C].cluster\_edges$;

Next, we describe Procedure $Constr\_Cover$. This procedure, given a cluster and parameters κ and W, constructs a (κ, W)-cover for the cluster. Its first parameter is the identity of the cluster, the second is its size, the third and fourth are, respectively, the κ and W parameters of the cover, and, finally, the fifth parameter is the level of the recursion, which is needed as a key to the $Local\_Data$ arrays of the nodes participating in the cover construction. During the procedure, every node creates entries in its $Local\_Data$ array for clusters, that contain the node. For every node v and cluster C such that $v \in C$, the node v writes down the list of edges incident to v that have another endpoint in the same cluster in the entry $Local\_Data[recur\_level, id_C].cluster\_edges$. In the field $tree\_edges$ of the same entry of the array, v writes down the list of edges incident to the node that belong to the BFS spanning tree of the cluster. Construction of such spanning trees can be done during the construction of the cover with no additional overhead, neither in communication nor in time complexities. We will also assume without loss of generality (since this can be easily implemented through the construction of the cover), that during the construction every node checks for every cluster to which it belongs, whether this cluster is "big" or "small", i.e., whether its size is at least the size of its parent cluster raised to the power $1 - \nu$ or not. Efficient distributed constructions, both the synchronous and the asynchronous ones, of such covers were presented in [1]. The covers constructed in [1] satisfy all the properties of (κ, W)-covers, that are listed in Definition 2.1. Furthermore, these covers enjoy an additional property, specifically, that every node appears in at most $O(\kappa n^{1/\kappa})$ clusters.

We next describe the procedure invoked by a node whenever it receives message $TRAVERSE$. This message has three arguments. Its first argument is the identity of the cluster, which is going to be traversed. The second argument is the level of the recursion. The third argument is ℓ, that is, the logarithm on base $1/(1-\nu)$ of the logarithm on base of K of the size of the traversed cluster. In every node v of the tree, the procedure retrieves from the $Local\_Data$ array the list of the clusters that contain it. For each cluster C rooted at v, the procedure checks whether this cluster is small (by examining the field $size\_bit$ of the corresponding entry of the $Local\_Data$ array). If it is, Procedure $Recurse$ is recursively invoked on this cluster. If the cluster is big, Procedure $Interconnect$ is invoked on the cluster. Procedure $Interconnect$ is described later.

For any cluster C denote $id(C)$ (respectively, $size(C)$) its identity (resp., size). The formal description of the procedure that is invoked whenever some node receives a $TRAVERSE$ message follows.

Upon Receival Message $TRAVERSE(parent\_cluster\_id, recur\_level, \ell)$ **do**

1. **If** $\exists$ cluster C whose record appears in $Local\_Data$ array
and s.t. $Local\_Data[recur\_level, id(C)].size\_bit = HIGH$
and $Local\_Data[recur\_level, id(C)].parent\_edge\_id = \perp$ **then**
$Interconnect(parent\_edge\_id, D^{\ell+1} + 4D^\ell\kappa, recur\_level)$;

2. **For all clusters** C whose record appears in array $Local\_Data$ **and** s.t. $Local\_Data[recur\_level, id(C)].size\_bit = LOW$ **and** $Local\_Data[recur\_level, id(C)].parent\_edge\_id = \perp$ **do** $Recurse(id(C), size(C), recur\_level + 1)$;

Finally, we describe Procedure *Interconnect*. This procedure has three parameters. The first one is the identity of the cluster inside which the procedure operates. This cluster is later referred to as the *host cluster*. The second parameter specifies the distance threshold. Clusters located at a distance not greater than the threshold are considered to be close one to another, and other are considered to be far one from another. The procedure builds a BFS spanning tree for the host cluster rooted in the initiator of the procedure. Next, it broadcasts over the spanning tree the message $INTERCONNECT$. Consider some non-leaf node of the tree, whose distance to the initiator is strictly smaller than the threshold, given in the second parameter. When it gets the message $INTERCONNECT$ from its parent node, it propagates the message to all its children and waits for their answers. This way the message $INTERCONNECT$ reaches the nodes, whose depth in the tree is precisely the threshold, and the leaves. When such a node gets the message $INTERCONNECT$, it checks whether it needs to be interconnected to the initiator of the procedure. Specifically, it checks whether it belongs to some big cluster that still was not interconnected to other relatively close big clusters. If the node does belong to such a cluster, it sends to its parent in the tree the message $INSERT(TRUE)$ and inserts the edge to its parent in the tree to its local spanner set. Otherwise, it sends to its parent the message $INSERT(FALSE)$. When some non-leaf node v, whose depth in the tree is strictly smaller than the threshold value, collects all the messages $INSERT$ from its children, it checks whether at least one of them has value $TRUE$. If there is at least one message $INSERT(TRUE)$, it means that there exists a descendant of v that should be connected to the root of the tree through the (unique) path between them in the tree. Thus, in this case v inserts the edge to its parent in the tree to its local spanner set, and sends to its parent the message $INSERT(TRUE)$. Otherwise, if all the children of v sent him the message $INSERT(FALSE)$, it means that there is no descendant of v that should be connected to the root of the tree through the tree path between them. Hence, in this case v checks only whether v himself should be connected to the root or not. If the answer is yes, v inserts the edge to its parent into its local spanner set and sends to its parent in the tree the message $INSERT(TRUE)$, and otherwise it sends to its parent the message $INSERT(FALSE)$.

Generally speaking, Procedure *Interconnect* may be seen as a broadcast and convergecast over the subtree of depth equal to the threshold value of the BFS spanning tree rooted at the initiator of the procedure.

The formal description of Procedure *Interconnect* follows.

Procedure *Interconnect*
Input: $parent\_cluster\_id = 0, 1, 2, \ldots, n\lceil \log_{1/(1-\nu)} \log_K n \rceil$, $Dist = 1, 2, \ldots,$
$recur\_level = 1, 2, \ldots, \lceil \log_{1/(1-\nu)} \log_K n \rceil$

1. $InterTree \leftarrow Constr\_ST(parent\_cluster\_id, Dist)$;

2. **For all** edges $e \in InterTree.tree\_edges$ **do in parallel**
   $Send\_Msg(e, INTERCONNECT(Dist, recur\_level))$;

3. Wait until getting from all the children message $INSERT$;

Upon Receival Message $INTERCONNECT(Dist, recur\_level)$ **do**
If $InterTree.depth < Dist$ **then**
For all edges $e \in InterTree.tree\_edges \setminus \{InterTree.parent\_edge\_id\}$ **do in parallel**

1. $Send\_Msg(e, INTERCONNECT(Dist, recur\_level))$;

2. Wait until getting from all the children message $INSERT$;

3. **If** one of the messages $INSERT$ is $INSERT(TRUE)$ **or**
   $\exists$ cluster C whose record appears in $Local\_Data$ array
   and s.t. $Local\_Data[recur\_level, id(C)].size\_bit = HIGH$
   and $Local\_Data[recur\_level, id(C)].status = WAIT$ **then**

   (a) $Send\_Msg(InterTree.parent\_edge\_id, INSERT(TRUE))$;
   (b) $H \leftarrow H \cup \{InterTree.parent\_edge\_id\}$;

   Else
   $Send\_Msg(InterTree.parent\_edge\_id, INSERT(FALSE))$;

Else /* $InterTree.depth = Dist$ */
If $\exists$ cluster C whose record appears in $Local\_Data$ array
and s.t. $Local\_Data[recur\_level, id(C)].size\_bit = HIGH$
and $Local\_Data[recur\_level, id(C)].status = WAIT$ **then**

1. $Send\_Msg(InterTree.parent\_edge\_id, INSERT(TRUE))$;

2. $H \leftarrow H \cup \{InterTree.parent\_edge\_id\}$;

Else
$Send\_Msg(InterTree.parent\_edge\_id, INSERT(FALSE))$;

3.3 Analysis of Protocol *Distrib_Spanner*

It is easy to see that Protocol *Distrib_Spanner* terminates and that the spanner sets of the nodes are consistent in the sense that for any edge $(u, w) \in E$, the edge is in the spanner set of the node u if and only if it is in the spanner set of w. Also, the analysis analogous to that of Sections 2.2 and 2.3 produces the same bounds on the size and stretch of the spanner as in Theorem 2.15.

So it remains to analyze the time and communication complexities of Protocol *Distrib_Spanner* in the synchronous and asynchronous settings.

First, we observe that the time and communication complexities of Protocol *Distrib_Spanner* are at most the sums of the corresponding complexities of the five following components of the protocol.

The first component consists of steps 1 and 2 of Procedure *Initiate*, i.e., of constructing a BFS spanning tree for the entire graph, and broadcasting a constant number of messages over the tree.

The second component consists of steps 2(b)i of Procedure *Recurse*, i.e, of constructing the covers for the clusters of $\bigcup \mathcal{C}_i^L$, using the notation of Section 2.

The third component consists of steps 1 of the procedure that is invoked whenever a node gets a $TRAVERSE$ message, i.e., of interconnecting the clusters of $\mathcal{C}_i^H$.

The fourth component consists of steps 2(b)ii of Procedure *Recurse*, i.e., of the invocations of Procedure *Seq_Broadcast_ST* on different clusters of $\bigcup \mathcal{C}_i^L$.

Finally, the fifth component consists of steps 2a of Procedure *Recurse*, i.e., of the invocations of Procedure *Broadcast_ST* on different clusters of $\mathcal{C}_\ell$.

Let us denote the synchronous time (resp., communication) complexity of the jth component ($j = 1, 2, \ldots, 5$) by $Time^S_{(j)}(Distrib\_Spanner)$ (resp., $Comm^S_{(j)}(Distrib\_Spanner)$). Analogously, denote the asynchronous time and communication complexities of the jth component by $Time^A_{(j)}(Distrib\_Spanner)$ and $Comm^A_{(j)}(Distrib\_Spanner)$, respectively.

Lemma 3.1 *The first component of Protocol Distrib_Spanner invoked on an unweighted n-vertex graph $G = (V, E)$ has the following time and communication complexities in the synchronous and asynchronous settings:*

$$\begin{aligned} Time^S_{(1)}(Distrib\_Spanner) &= O(Diam(G)) \ , \\ Time^A_{(1)}(Distrib\_Spanner) &= O(Diam(G)\log^3(n)) \ , \\ Comm^S_{(1)}(Distrib\_Spanner) &= O(|E|) \ , \\ Comm^A_{(1)}(Distrib\_Spanner) &= O(|E| + n\log^3 n) \ . \end{aligned}$$

Proof: The construction of a BFS spanning tree for a graph G consumes synchronous time and communication complexities of $O(Diam(G))$ and $O(E)$, respectively. In the asynchronous setting, this can be done in $O(Diam(G)\log^3 n)$ time and $O(|E| + n\log^3 n)$ communication. This result is due to [8, 9]. Broadcasting a constant number of messages over a given BFS spanning tree consumes $O(Diam(G))$ time and and $O(n)$ communication in both the synchronous and asynchronous settings. This concludes the proof. ▮

Lemma 3.2 *The second component of Protocol Distrib_Spanner invoked on an unweighted n-vertex graph $G = (V, E)$ has the following time and communication complexities in the synchronous and asynchronous settings:*

$$\begin{aligned} Time^S_{(2)}(Distrib\_Spanner) &= O(\kappa^2 n^{1+\frac{1}{\kappa\nu}}) \ , \\ Time^A_{(2)}(Distrib\_Spanner) &= O(\kappa^3 n^{1+\frac{1}{\kappa\nu}}\log n) \ , \\ Comm^S_{(2)}(Distrib\_Spanner) &= O(|E|n^{\frac{1}{\kappa\nu}} + \kappa^2 n^{1+\frac{1}{\kappa\nu}}) \ , \\ Comm^A_{(2)}(Distrib\_Spanner) &= O(|E|n^{\frac{1}{\kappa\nu}} + \kappa^3 n^{1+\frac{1}{\kappa\nu}}\log n) \ . \end{aligned}$$

Proof: For any cluster C, denote by $\tau^S_{\mathcal{C}}(C)$ (resp., $\tau^A_{\mathcal{C}}(C)$) the time complexity of constructing a (κ, W)-cover for C in the synchronous (resp., asynchronous) setting. This is well-defined, since κ is fixed through the analysis and the time complexity does not depend on W. By [1], $\tau^S_{\mathcal{C}}(C) = O(|C|^{1+1/\kappa}\kappa^2)$ and $\tau^A_{\mathcal{C}}(C) = O(|C|^{1+1/\kappa}\kappa^3\log|C|)$.

Let τ^S_i (resp., τ^A_i) denote the total time complexity of constructing all the covers for the clusters of $\mathcal{C}^L_i$ in the synchronous (resp., asynchronous) setting.

It follows that for $i = 1, 2, \ldots, \ell$,

$$\tau^S_i \leq \sum_{C\in\mathcal{C}_{i-1}(C)} \tau^S_{\mathcal{C}}(C) = O(\sum_{C\in\mathcal{C}_{i-1}(C)} |C|^{1+1/\kappa}\kappa^2) = O(\kappa^2(MS(\mathcal{C}^L_{i-1}))^{1/\kappa}s(\mathcal{C}^L_{i-1})) \ .$$

Recall that $\mathcal{C}^L_0 = \{V\}$, and thus $MS(\mathcal{C}^L_0) = s(\mathcal{C}^L_0) = n$. Hence, $\tau^S_1 = O(\kappa^2 n^{1+1/\kappa})$.

For $i \geq 2$, Lemmas 2.2 and 2.3 imply that

$$\tau^S_i = O(\kappa^2 n^{\frac{(1-\nu)^{i-1}}{\kappa}} n^{1+\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}}) = O(\kappa^2 n^{1+\frac{1-(1-\nu)^i}{\kappa\nu}}) \ .$$

Hence, $Time^S_{(2)}(Distrib\_Spanner) \leq \sum_{i=1}^{\ell}\tau^S_i = O(\kappa^2 n^{1+\frac{1}{\kappa\nu}})$.

Analogously,

$$\tau^A_i \leq \sum_{C\in\mathcal{C}^L_{i-1}} \tau^A_{\mathcal{C}}(C) = O(\sum_{C\in\mathcal{C}^L_{i-1}} |C|^{1+1/\kappa}\kappa^3\log|C|) = O(\kappa^3(MS(\mathcal{C}^L_{i-1}))^{1/\kappa}\log(MS(\mathcal{C}^L_{i-1}))\cdot s(\mathcal{C}^L_{i-1})) \ .$$

Hence, $\tau_1^A = O(\kappa^3 n^{1+1/\kappa}\log n)$, and for $i \geq 2$,

$$\tau_i^A = O(\kappa^3 n^{\frac{(1-\nu)^{i-1}}{\kappa}} \cdot (1-\nu)^{i-1} n^{1+\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}} \log n) = O(\kappa^3 \cdot (1-\nu)^{i-1} n^{1+\frac{1}{\kappa}\frac{1-(1-\nu)^{i}}{\nu}} \log n) \ .$$

Hence, $Time^A_{(2)}(Distrib\_Spanner) = O(\kappa^3 n^{1+\frac{1}{\kappa\nu}}\log n)$.

Turning to the communication complexity, denote by $Comm^S_{\mathcal{C}}(C)$ (resp., $Comm^A_{\mathcal{C}}(C)$) the communication complexity of constructing a (κ, W)-cover (for some W) in a cluster C in the synchronous (resp., asynchronous) setting.

By [1],

$$\begin{aligned} Comm^S_{\mathcal{C}}(C) &= O(|E| + n^{1+1/\kappa}\kappa^2) \ , \\ Comm^A_{\mathcal{C}}(C) &= O(|E| + n^{1+1/\kappa}\kappa^3 \log n) \ . \end{aligned}$$

Denote by $Comm^S_i$ (resp., $Comm^A_i$) the total communication complexity of constructing all the covers for the clusters of $\mathcal{C}^L_i$ in the synchronous (resp., asynchronous) setting.

It follows that for any $i = 1, 2, \ldots, \ell$,

$$Comm^S_i \leq \sum_{C \in \mathcal{C}^L_{i-1}} O(|E(C)| + |C|^{1+1/\kappa}\kappa^2) = O(e(\mathcal{C}^L_{i-1}) + (MS(\mathcal{C}^L_{i-1}))^{1/\kappa}\kappa^2 \cdot s(\mathcal{C}^L_{i-1})) \ .$$

For $\mathcal{C}^L_0 = \{V\}$, $e(\mathcal{C}^L_0) = |E|$. Thus, $Comm^S_1 = O(|E| + n^{1+1/\kappa}\kappa^2)$. By Lemmas 2.2, 2.3 and 2.11, for $i = 2, 3, \ldots, \ell$,

$$Comm^S_i = O(|E| n^{\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}} + \kappa^2 n^{1+\frac{(1-\nu)^{i-1}}{\kappa}+\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}}) = O(|E| n^{\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}} + \kappa^2 n^{1+\frac{1}{\kappa}\frac{1-(1-\nu)^{i}}{\kappa}}) \ .$$

Hence,

$$Comm^S_{(2)} = \sum_{i=1}^{\ell} Comm^S_i = O(|E| n^{\frac{1}{\kappa\nu}} + \kappa^2 n^{1+\frac{1}{\kappa\nu}}) \ .$$

Analogously,

$$Comm^A_i \leq \sum_{C \in \mathcal{C}^L_{i-1}} O(|E(C)| + |C|^{1+1/\kappa}\kappa^3 \log|C|) = O(e(\mathcal{C}^L_{i-1}) + (MS(\mathcal{C}^L_{i-1}))^{1/\kappa}\kappa^3 \log(MS(\mathcal{C}^L_{i-1})) \cdot s(\mathcal{C}_{i-1})) \ .$$

Hence, $Comm^A_1 = O(|E| + \kappa^3 n^{1+1/\kappa}\log n)$.

By Lemmas 2.2, 2.3 and 2.11,

$$Comm^A_i = O(|E| n^{\frac{1-(1-\nu)^{i-1}}{\kappa\nu}} + \kappa^3 n^{1+\frac{1-(1-\nu)^{i}}{\kappa\nu}} (1-\nu)^{i-1} \log n) \ .$$

Therefore,

$$Comm^A_{(2)} = \sum_{i=1}^{\ell} Comm^A_i = O(|E| n^{\frac{1}{\kappa\nu}} + \kappa^3 n^{1+\frac{1}{\kappa\nu}} \log n) \ . \qquad \blacksquare$$

Lemma 3.3 *The third component of Protocol Distrib_Spanner invoked on an unweighted n-vertex graph $G = (V, E)$ has the following time and communication complexities in the synchronous and asynchronous settings:*

$$\begin{aligned} Time^S_{(3)}(Distrib\_Spanner) &= O(D^{\ell+1} n^{1+\frac{1}{\kappa\nu}}) \ , \\ Time^A_{(3)}(Distrib\_Spanner) &= O(D^{2\ell+2} n^{1+\frac{1}{\kappa\nu}}) \ , \\ Comm^S_{(3)}(Distrib\_Spanner) &= O(|E| n^{\frac{1}{\kappa\nu}+\nu}) \ , \\ Comm^A_{(3)}(Distrib\_Spanner) &= O(|E| n^{\frac{1}{\kappa\nu}+\nu} + D^{\ell+1} n^{1+\frac{1}{\kappa\nu}+\nu+\frac{1}{\kappa}}) \ . \end{aligned}$$

Proof: First, note that in each cluster C on which Procedure *Recurse* is invoked, Procedure Interconnect is invoked at most $|\mathcal{C}^H(C)| \le |C|^{1+1/\kappa}/|C|^{1-\nu} = |C|^{1/\kappa+\nu}$ times.

We also observe that $MS(\mathcal{C}^H_{i+1}) \le MS(\mathcal{C}^L_i)$, and by Lemma 2.2, $MS(\mathcal{C}^L_i) \le n^{(1-\nu)^i}$, for any $i = 1, 2, \ldots, \ell$. Hence, $MS(\mathcal{C}^H_{i+1}) \le n^{(1-\nu)^i}$, for any $i = 1, 2, \ldots, \ell$.

Consider some single invocation of Procedure *Interconnect* on a cluster C with distance threshold parameter $Dist$. The procedure constructs a partial BFS spanning tree of depth $Dist$ of the cluster, and then performs broadcast and convergecast over the spanning tree. In the synchronous setting this requires $O(Dist)$ time and $O(|E(C)|)$ communication. In the asynchronous setting this requires $O(Dist^2)$ time and $O(|E(C)| + n \cdot Dist)$ communication using Dijkstra BFS protocol (cf. [25], pp.52-55). Note that we prefer to use Dijkstra protocol instead of the BFS protocol due to [2, 9], although, in general, the latter has better complexities ($O(Dist \cdot \log^3 n)$ time and $O(|E(C)| + \log^3 n)$ communication). This is because the Dijkstra BFS protocol is advantageous whenever $Dist$ is small.

Let us denote by $Dist_i$ the maximum of the values of the parameter $Dist$ with which Procedure *Interconnect* is invoked on the ith level of the recursion.

Then,

$$Dist_i \le D^{\lceil \log_{1/(1-\nu)} \log_K MS(\mathcal{C}^H_i) \rceil + 1} \le D^{\lceil \log_{1/(1-\nu)} \log_K n^{(1-\nu)^{i-1}} \rceil + 1} = D^{\ell+2-i} .$$

Denote by $\bar{\tau}^S_i$ (resp, $\bar{\tau}^A_i$) the total time complexity of all the invocations of Procedure *Interconnect* in the ith level of the recursion in the synchronous (resp., asynchronous) setting. Then, for any $i = 1, 2, \ldots, \ell$,

$$\bar{\tau}^S_i \le \sum_{C \in \mathcal{C}^H_i} Dist_i |\mathcal{C}^H(C)| \le \sum_{C \in \mathcal{C}^H_i} D^{\ell+2-i} |C|^{1/\kappa+\nu} .$$

Since, $1/\kappa + \nu < 1$, the sum $\sum_{C \in \mathcal{C}^H_i} |C|^{1/\kappa+\nu}$ is at most $s(\mathcal{C}^H_i)$. The latter is bounded by Lemma 2.3, specifically, $\bar{\tau}^S_i \le D^{\ell+2-i} n^{1+\frac{1-(1-\nu)^i}{\kappa\nu}}$.

Hence,

$$Time^S_{(3)}(Distrib\_Spanner) \le \sum_{i=1}^{\ell} \bar{\tau}^S_i = O(D^{\ell+1} n^{1+\frac{1}{\kappa\nu}}) .$$

Analogously,

$$\bar{\tau}^A_i \le \sum_{C \in \mathcal{C}^H_i} Dist^2_i |\mathcal{C}^H(C)| \le D^{2(\ell+2-i)} n^{1+\frac{1-(1-\nu)^i}{\kappa\nu}} .$$

Hence,

$$Time^A_{(3)}(Distrib\_Spanner) \le \sum_{i=1}^{\ell} \bar{\tau}^A_i = O(D^{2\ell+2} n^{1+\frac{1}{\kappa\nu}}) .$$

Similarly, denote by $\overline{Comm}^S_i$ (resp., $\overline{Comm}^A_i$) the total communication complexity of all the invocations of Procedure *Interconnect* in the ith level of the recursion in the synchronous (resp., asynchronous) setting. Then, for any $i = 1, 2, \ldots, \ell$,

$$\begin{aligned} \overline{Comm}^S_i &\le \sum_{C \in \mathcal{C}^H_i} |E(C)||\mathcal{C}^H(C)| \le \sum_{C \in \mathcal{C}^H_i} |E(C)||C|^{1/\kappa+\nu} \le e(\mathcal{C}^L_{i-1}) \cdot (MS(\mathcal{C}^H_i))^{1/\kappa+\nu} \qquad (11) \\ &\le |E| n^{\frac{1-(1-\nu)^i}{\kappa\nu} + (1-\nu)^{i-1}\nu} . \end{aligned}$$

Thus,

$$Comm^S_{(3)}(Distrib\_Spanner) \le \sum_{i=1}^{\ell} \overline{Comm}^S_i = O(|E| n^{\frac{1}{\kappa\nu}+\nu}) . \qquad (12)$$

Analogously,

$$\begin{aligned}\overline{Comm}_i^A &\leq \sum_{C\in\mathcal{C}_i^H} (|E(C)| + |C|D^{\ell+2-i})|C|^{\nu+1/\kappa} \leq \overline{Comm}_i^S + D^{\ell+2-i}s(\mathcal{C}_i^H)(MS(\mathcal{C}_i^H))^{\nu+1/\kappa} \\ &\leq \overline{Comm}_i^S + D^{\ell+2-i}n^{1+1/\kappa(\frac{1-(1-\nu)^i}{\nu}+(1-\nu)^{i-1})+(1-\nu)^{i-1}\nu}\ .\end{aligned}$$

Hence,

$$Comm^A_{(3)}(Distrib\_Spanner) \leq \sum_{i=1}^{\ell} \overline{Comm}_i^A = O(|E|n^{\frac{1}{\kappa\nu}} + D^{\ell+1}n^{1+\frac{1}{\kappa\nu}+1/\kappa+\nu})\ . \quad \blacksquare$$

Lemma 3.4 *The time and communication complexities of the fourth component of Protocol Distrib_Spanner invoked on an unweighted n-vertex graph $G = (V,E)$ in the synchronous and asynchronous settings are all bounded by $O(n^{1+\frac{1}{\kappa\nu}})$.*

Proof: The fourth component of the protocol consists of the different invocations of Procedure *Seq_Broadcast_ST*. Recall that this procedure is used to deliver the control of the execution from one node to another. Some nodes, once getting the "relay stick", initiate some additional activity, like recursively invoking Procedure *Recurse*, or invoking Procedure *Interconnect*. However, the costs of these activities are taken into account in other components of the protocol.

When Procedure *Seq_Broadcast_ST* is invoked on a cluster C, it requires $O(|C|)$ time and communication in both synchronous and asynchronous settings, since it sequentially passes through a spanning tree of the cluster. Since the procedure is invoked on all the clusters that are formed by Protocol *Distrib_Spanner*, except those formed on the very last level, it follows that the total time and communication complexities of the fourth component in both the synchronous and asynchronous settings are at most $\sum_{i=1}^{\ell} s(\mathcal{C}_i) = O(n^{1+\frac{1}{\kappa\nu}})$. $\blacksquare$

Lemma 3.5 *The time and communication complexities of the fifth component of Protocol Distrib_Spanner invoked on an unweighted n-vertex graph $G = (V,E)$ in the synchronous and asynchronous settings are all bounded by $O(n^{1+\frac{1}{\kappa\nu}})$.*

Proof: The fifth component consists of broadcasting the message $INSERT\_ALL$. These broadcasts are performed on all the clusters of $\mathcal{C}_\ell$. These broadcasts are performed over spanning trees that were constructed as a result of executing Procedure *Constr_Cover* on clusters of $\mathcal{C}_{\ell-1}$, and the costs of invocations of Procedure *Constr_Cover* are accounted as a part of the second component (see Lemma 3.2).

Hence, $Time^S_{(5)}(Distrib\_Spanner) \leq \sum_{C\in\mathcal{C}_\ell} |C| = s(\mathcal{C}_\ell) = O(n^{1+\frac{1}{\kappa\nu}})$, and analogous computation yields the same results for $Time^A_{(5)}(Distrib\_Spanner)$, $Comm^S_{(5)}(Distrib\_Spanner)$ and $Comm^A_{(5)}(Distrib\_Spanner)$. $\blacksquare$

For any protocol Π, denote by $Time^S(\Pi)$ (resp., $Comm^S(\Pi)$), the synchronous time (resp., communication) complexity of the protocol. Analogously, let $Time^A(\Pi)$ and $Comm^A(\Pi)$ be the asynchronous time and communication complexities of Π, respectively.

Theorem 3.6 *For any sufficiently large integer n, any n-vertex graph G, $\kappa = 1,2,\ldots$, $0 < \nu \leq 1/2 - 1/\kappa$, $D, K = 1,2,\ldots$, such that $D \geq \kappa$, the time and communication complexities of Protocol Distrib_Spanner when it is invoked on the five-tuple (G,κ,ν,D,K) in the synchronous and asynchronous settings are as* *follows (we denote $\ell = \lceil \log_{1/(1-\nu)} \log_K n \rceil$):*

$$\begin{aligned}
Time^S(Distrib_Spanner) &= O(D^{\ell+1} n^{1+\frac{1}{\kappa\nu}}) \ , \\
Time^A(Distrib_Spanner) &= O((\kappa^3 \log n + D^{2\ell+2}) n^{1+\frac{1}{\kappa\nu}}) \ , \\
Comm^S(Distrib_Spanner) &= O(|E| n^{\frac{1}{\kappa\nu}+\nu}) \ , \\
Comm^A(Distrib_Spanner) &= O(|E| n^{\frac{1}{\kappa\nu}+\nu} + D^{\ell+1} n^{1+\frac{1}{\kappa\nu}+1/\kappa+\nu}).
\end{aligned}$$

Proof: By Lemmas 3.1, 3.2, 3.3, 3.4, 3.5, and since $D^{\ell+1} \geq \kappa^2$, it follows that

$$\begin{aligned}
Time^S(Distrib_Spanner) &\leq \sum_{j=1}^{5} Time^S_{(j)}(Distrib_Spanner) \\
&= O(Diam(G)) + O(\kappa^2 n^{1+\frac{1}{\kappa\nu}}) + O(D^{\ell+1} n^{\frac{1}{\kappa\nu}}) + O(n^{1+\frac{1}{\kappa\nu}}) + O(n^{1+\frac{1}{\kappa\nu}}) \\
&= O(D^{\ell+1} n^{\frac{1}{\kappa\nu}}) \ , \\
Time^A(Distrib_Spanner) &\leq \sum_{j=1}^{5} Time^S_{(j)}(Distrib_Spanner) \\
&= O(Diam(G)\log^3 n) + O(\kappa^3 \log n \cdot n^{1+\frac{1}{\kappa\nu}}) + O(D^{2\ell+2} n^{1+\frac{1}{\kappa\nu}}) + O(n^{1+\frac{1}{\kappa\nu}}) \\
&+ O(n^{1+\frac{1}{\kappa\nu}}) = O((\kappa^3 \log n + D^{2\ell+2}) n^{1+\frac{1}{\kappa\nu}}) \ , \\
Comm^S(Distrib_Spanner) &= \sum_{j=1}^{5} Comm^S_{(j)}(Distrib_Spanner) \\
&= O(|E|) + O(|E| n^{\frac{1}{\kappa\nu}} + \kappa^2 n^{1+\frac{1}{\kappa\nu}}) + O(|E| n^{\frac{1}{\kappa\nu}+\nu}) + O(n^{1+\frac{1}{\kappa\nu}}) + O(n^{1+\frac{1}{\kappa\nu}}) \\
&= O(|E| n^{\frac{1}{\kappa\nu}+\nu}) \ , \\
Comm^A(Distrib_Spanner) &= \sum_{j=1}^{5} Comm^A_{(j)}(Distrib_Spanner) \\
&= O(|E| + n\log^3 n) + O(|E| n^{\frac{1}{\kappa\nu}} + \kappa^3 n^{1+\frac{1}{\kappa\nu}} \log n) \\
&+ O(|E| n^{\frac{1}{\kappa\nu}+\nu} + D^{\ell+1} n^{1+\frac{1}{\kappa\nu}+1/\kappa+\nu}) + O(n^{1+\frac{1}{\kappa\nu}}) + O(n^{1+\frac{1}{\kappa\nu}}) \\
&= O(|E| n^{\frac{1}{\kappa\nu}+\nu} + D^{\ell+1} n^{1+\frac{1}{\kappa\nu}+1/\kappa+\nu}) \ . \quad \blacksquare
\end{aligned}$$

By substituting the values of κ, ν, D and K as in Theorem 2.15, we get the following.

Theorem 3.7 *For any sufficiently large integer n, any unweighted n-vertex graph $G = (V, E)$, and any three real numbers $0 < \epsilon, \rho, \zeta < 1$ such that $\zeta/2 + 1/3 > \rho > \zeta/2$, ρ, ζ are constant, and $\epsilon = \Omega(\frac{1}{\text{polylog}\, n})$, running Protocol Distrib_Spanner on the five-tuple $(G, \kappa = \frac{2}{\zeta(\rho-\zeta/2)}, \nu = \rho - \zeta/2, D = 8\frac{2}{\zeta(\rho-\zeta/2)}\epsilon^{-1}\lceil \log_{1-(\rho-\zeta/2)} \zeta/2 \rceil, K = n^{\zeta/2})$ produces a $(1+\epsilon, \beta(\zeta,\rho,\epsilon))$-spanner of size $O(n^{1+\zeta})$. The time and communication complexities of Protocol of Distrib_Spanner on this input in the synchronous and asynchronous settings are*

$$\begin{aligned}
Time^S(Distrib_Spanner) &= O(\beta(\zeta,\rho,\epsilon) n^{1+\zeta/2}) \ , \\
Time^A(Distrib_Spanner) &= O((\beta(\zeta,\rho,\epsilon)^2 + \frac{1}{\zeta^3(\rho-\zeta/2)^3} \log n) n^{1+\zeta/2}) \ , \\
Comm^S(Distrib_Spanner) &= O(|E| n^{\rho}) \ , \\
Comm^A(Distrib_Spanner) &= O(|E| n^{\rho} + \beta(\zeta,\rho,\epsilon) \cdot n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}}) \ .
\end{aligned}$$

$\beta(\zeta, \rho, \epsilon)$ is defined by (10), and it is constant whenever ζ, ρ and ϵ are.

Note that if we fix $\epsilon > 0$ as a constant, then the expressions for time and communication complexities become simpler. Specifically,

$$\begin{aligned}
Time^S(Distrib_Spanner) &= O(n^{1+\zeta/2}) , \\
Time^A(Distrib_Spanner) &= O(\log n \cdot n^{1+\zeta/2}) , \\
Comm^S(Distrib_Spanner) &= O(|E|n^{\rho}) , \\
Comm^A(Distrib_Spanner) &= O(|E|n^{\rho} + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}}) .
\end{aligned}$$

Next, let us consider the following protocol for computing almost shortest paths between all the pairs of nodes $S \times V$, where $S \subseteq V$ is a given set of s sources. The protocol, later referred to as *Distrib_ASP*, (standing for "Distributed Almost Shortest Paths"), starts with invoking Protocol *Distrib_Spanner*, and then performs s BFS searchs over the constructed spanner, initiated from the s sources. The following theorem is a direct consequence of Theorem 3.7.

Theorem 3.8 *For any sufficiently large integer n, any unweighted n-vertex graph $G = (V, E)$, any real numbers $0 < \zeta, \rho, \epsilon < 1$ such that $\zeta/2 + 1/3 > \rho > \zeta/2$, ρ, ζ are constant and $\epsilon = \Omega(\frac{1}{\mathrm{polylog}\,(n)})$, any subset $S \subseteq V$ of s vertices, Protocol Distrib_ASP computes the paths $P_{u,w} \subseteq E$ between all pairs of nodes $\{u, w\} \in S \times V$ using the following time and communication complexities in the synchronous and asynchronous settings:*

$$\begin{aligned}
Time^S(Distrib_ASP) &= O(\beta(\zeta,\rho,\epsilon)n^{1+\zeta/2} + s \cdot (Diam(G) + \beta(\zeta,\rho,\epsilon))) , \\
Time^A(Distrib_ASP) &= O((\beta(\zeta,\rho,\epsilon)^2 + \log n)n^{1+\zeta/2} + s \cdot Diam(G)\log^3 n + s\log^3 n\beta(\zeta,\rho,\epsilon)) , \\
Comm^S(Distrib_ASP) &= O(|E|n^{\rho} + n^{1+\zeta}s) , \\
Comm^A(DistribASP) &= O(|E|n^{\rho} + n^{1+\zeta}s + \beta(\zeta,\rho,\epsilon)n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}}) .
\end{aligned}$$

These paths satisfy

1. $L(P_{u,w}) \leq (1+\epsilon)d_G(u,w) + \beta(\zeta,\rho,\epsilon)$.
2. $|\bigcup_{\{u,w\}\cap S \neq \emptyset} P_{u,w}| = O(n^{1+\zeta})$.

$\beta(\zeta,\rho,\epsilon)$ is defined by (10), and it is constant whenever ζ, ρ and ϵ are.

Proof: By Theorem 3.7, the spanner H has $O(n^{1+\zeta})$ edges. In the synchronous setting the time and communication complexities of constructing a BFS spanning tree for an n-vertex spanner H with $O(n^{1+\zeta})$ edges is $O(Diam(H)) = O(Diam(G) + \beta(\zeta,\rho,\epsilon))$ and $O(n^{1+\zeta})$, respectively. Hence, the statements concerning $Time^S(Distrib\_ASP)$ and $Comm^S(Distrib\_ASP)$ follow.

In the asynchronous setting, we use the result of [8, 9], stating that constructing a BFS spanning tree for an n-vertex spanner H with $O(n^{1+\zeta})$ edges can be performed in $O(Diam(H)\log^3 n)$ time using $O(n^{1+\zeta} + n\log^3 n) = O(n^{1+\zeta})$ communication. Hence, the results concerning $Time^A(Distrib\_ASP)$ and $Comm^A(Distrib\_ASP)$ follow, completing the proof. ∎

Note that if fix ϵ as a constant, the expressions for the synchronous and asynchronous complexities of Protocol *Distrib_ASP* can be simplified to

$$\begin{aligned}
Time^S(Distrib_ASP) &= O(n^{1+\zeta/2} + s \cdot Diam(G)) , \\
Time^A(Distrib_ASP) &= O(\log n \cdot n^{1+\zeta/2} + s \cdot Diam(G)\log^3 n) , \\
Comm^S(Distrib_ASP) &= O(|E|n^{\rho} + n^{1+\zeta}s) , \\
Comm^A(DistribASP) &= O(|E|n^{\rho} + n^{1+\zeta}s + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}}) .
\end{aligned}$$

This result should be compared with a straightforward protocol for computing the *shortest paths* between all pairs of nodes $S \times V$ (we are not aware of any existing protocol for computing *almost shortest paths*). The naive protocol constructs s BFS spanning trees rooted at the sources (nodes of S). Protocol $Distrib\_ASP$, on the other hand, starts with computing a spanner, and then invokes the naive protocol on the constructed spanner. The time complexity of the naive protocol is $O(s \cdot Diam(G))$ in the synchronous setting, and $O(s \cdot Diam(G) \log^3 n)$ in the asynchronous one. This is instead of $O(s \cdot Diam(G) + n^{1+\zeta/2})$ and $O(s \cdot Diam(G) \log^3 n + \log n \cdot n^{1+\zeta/2})$ of our protocol. In other words, Protocol $Distrib\_ASP$ has essentially the same time complexity in both the synchronous and asynchronous settings, whenever $s \cdot Diam(G) \geq n^{1+\zeta/2}$ (recall that ζ may be chosen as an arbitrarily small constant). In particular, when one is interested in computing the distances between *all* the pairs of nodes (i.e., $s = n$), then the condition becomes equivalent to $Diam(G) \geq n^{\zeta/2}$.

On the other hand, Protocol $Distrib\_ASP$ has significantly better communication complexity in both the synchronous and asynchronous settings. Specifically, the naive protocol achieves $O(s \cdot |E|)$ and $O(s \cdot (|E| + n \log^3 n))$ communication complexities in the synchronous and asynchronous settings, respectively. Protocol $Distrib\_ASP$ achieves $O(|E|n^\rho + s \cdot n^{1+\zeta})$ and $O(|E|n^\rho + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}} + s \cdot n^{1+\zeta})$ communication complexities in the synchronous and asynchronous settings, respectively. Note that, in particular, when one is interested in computing the distances between all the pairs of nodes, the synchronous communication complexity of Protocol $Distrib\_Spanner$ is $O(|E|n^\rho + n^{2+\zeta})$ instead of $O(|E|n)$ of the naive protocol, where ρ and ζ may be fixed as arbitrarily small constants. Analogous improvement holds for the asynchronous setting too.

Next, we present the two modifications of Protocol $Distrib\_Spanner$ that enable it to deal with weighted graphs. The obtained protocol is henceforth referred to as *Protocol* $Distrib\_Weighted$. The first modification is that in step 2(b)i of Procedure $Recurse$ the fourth parameter of Procedure $Constr\_Cover$, that is invoked on this step, is now $D^\ell \omega_{max}$ instead of D^ℓ. The second modification is that in step 1 of the procedure that is invoked whenever some node gets a $TRAVERSE$ message, the second parameter is now $(D^{\ell+1}\omega_{max} + 4D^\ell \omega_{max}\kappa)$ instead of $(D^{\ell+1} + 4D^\ell\kappa)$. Note that these changes are completely analogous to the changes between Algorithm $Recur\_Weighted$ and Algorithm $Recur\_Spanner$, that were described in Section 2.5.

Let us now consider the effect of these modifications on the time and communication complexities of the protocol in the synchronous and asynchronous settings.

Theorem 3.9 *For any sufficiently large integer n, any n-vertex weighted graph $G = (V, E, \omega)$, any $\kappa = 1, 2, \ldots$, $0 < \nu < 1/2 - 1/\kappa$, $D, K = 1, 2, \ldots$, Protocol $Distrib\_Weighted$ invoked on the same five-tuple as in Theorem 2.15 produces a $(1+\epsilon, \beta(\zeta, \rho, \epsilon)\omega_{max})$-spanner with $O(\omega_{max} n^{1+\zeta})$ edges using the following time and communication complexities in the synchronous and asynchronous settings:*

$$\begin{aligned}
Time^S(Distrib_Weighted) &= O(D^{\ell+1}\omega_{max} n^{1+\frac{1}{\kappa\nu}}) , \\
Time^A(Distrib_Weighted) &= O(D^{\ell+1}\omega_{max} n^{1+\frac{1}{\kappa\nu}} \log^3 n) , \\
Comm^S(Distrib_Weighted) &= O(|E| n^{\frac{1}{\kappa\nu}+\nu}) , \\
Comm^A(Distrib_Weighted) &= O(|E| n^{\frac{1}{\kappa\nu}+\nu} + n^{1+\frac{1}{\kappa\nu}+\nu+1/\kappa} \log^3 n) ,
\end{aligned}$$

where $\beta(\zeta, \rho, \epsilon)$ is defined by (10).

Proof: First, observe the first, the fourth and the fifth components of the protocol involve only constructing BFS spanning trees and broadcasting over these trees. These operations can be done on weighted graphs as efficiently as on unweighted ones. Thus, the time and communication complexities of these components do not change.

Secondly, recall that the second component involves only constructing (κ, W)-covers. Note that the complexities of these constructions depend neither on W, nor on the fact whether the graph on which these constructions are invoked is weighted or unweighted [1]. Hence, the complexities of the second component are not affected either.

However, this is not the case with the third component, since the value of the distance threshold of Procedure *Interconnect* is greater by a factor of ω_{max} in Protocol *Distrib_Weighted* than in Protocol *DistribSp*. Hence, the synchronous time complexity of the former protocol is

$$Time^S_{(3)}(Distrib\_Weighted) \le \sum_{i=1}^{\ell} \bar{\tau}_i^S \le \omega_{max} \sum_{i=1}^{\ell} D^{\ell+2-i} n^{1+\frac{1-(1-\nu)^i}{\kappa\nu}} = O(D^{\ell+1}\omega_{max} n^{1+\frac{1}{\kappa\nu}}) \ .$$

The communication complexity in the synchronous setting is not affected, because by inequalities (11) and (12),

$$Comm^S_{(3)}(Distrib\_Weighted) \le \sum_{i=1}^{\ell} \sum_{C \in \mathcal{C}_i^H} |E(C)| \cdot |\mathcal{C}^H(C)| = O(|E| n^{\frac{1}{\kappa\nu}+\nu}) \ .$$

For the asynchronous setting we use the protocol of [8, 9] for constructing partial BFS tree. Cast to the notation of Lemma 3.3, it follows that

$$\begin{aligned} \bar{\tau}_i &\le \sum_{C \in \mathcal{C}_i^H} Dist_i \log^3 |C| \cdot |\mathcal{C}^H(C)| \le Dist_i \log^3 (MS(\mathcal{C}_i^H)) \sum_{C \in \mathcal{C}_i^H} |C|^{1/\kappa+\nu} \\ &\le D^{\ell+2-i} \omega_{max} (1-\nu)^{3(i-1)} n^{1+\frac{1-(1-\nu)^{i-1}}{\kappa\nu}} \ . \end{aligned}$$

Hence,

$$Time^A_{(3)}(Distrib\_Weighted) = O(\omega_{max} D^{\ell+1} n^{1+\frac{1}{\kappa\nu}} \log^3 n) \ .$$

Finally,

$$\overline{Comm}_i^A \le \sum_{C \in \mathcal{C}_i^H} (|E(C)| + |C| \log^3 |C|) |\mathcal{C}^H(C)| \le \overline{Comm}_i^S + \sum_{C \in \mathcal{C}_i^H} |C|^{1+1/\kappa+\nu} \log^3 |C| \ .$$

$$\begin{aligned} \sum_{C \in \mathcal{C}_i^H} |C|^{1+1/\kappa+\nu} \log^3 |C| &\le (MS(\mathcal{C}_i^H))^{1/\kappa+\nu} \log^3 (MS(\mathcal{C}_i^H)) \cdot s(\mathcal{C}_i^H) \\ &\le n^{(1-\nu)^{i-1}(1/\kappa+\nu)} (1-\nu)^{3(i-1)} \log^3 n \cdot O(n^{1+\frac{1-(1-\nu)^i}{\kappa\nu}}) \ . \end{aligned}$$

Hence,

$$Comm^A_{(3)}(Distrib\_Weighted) \le Comm^S_{(3)} + O(n^{1+\frac{1}{\kappa\nu}+1/\kappa+\nu} \log^3 n) = O(|E| n^{\frac{1}{\kappa\nu}+\nu} + n^{1+\frac{1}{\kappa\nu}+1/\kappa+\nu} \log^3 n) \ .$$

▮

By substituting appropriate values of the parameters κ, ν, D and K, we get the following corollary.

Corollary 3.10 *For any sufficiently large integer n, any weighted n-vertex graph $G = (V, E, \omega)$, and any three real numbers $0 < \epsilon, \rho, \zeta < 1$ such that $\zeta/2 + 1/3 > \rho > \zeta/2$, ρ, ζ are constant, and $\epsilon = \Omega(\frac{1}{\text{polylog } n})$, running Protocol Distrib_Weighted on the five-tuple $(G, \kappa = \frac{2}{\zeta(\rho-\zeta/2)}, \nu = \rho - \zeta/2, D = 8\frac{2}{\zeta(\rho-\zeta/2)}\epsilon^{-1}\lceil \log_{1-(\rho-\zeta/2)} \zeta/2 \rceil, K = n^{\zeta/2})$ produces a $(1+\epsilon, \beta(\zeta, \rho, \epsilon))$-spanner of size $O(n^{1+\zeta})$. The time* *and communication complexities of Protocol of $Distrib\_Weighted$ on this input in the synchronous and asynchronous settings are*

$$\begin{aligned}
Time^S(Distrib_Weighted) &= O(\beta(\zeta,\rho,\epsilon)\omega_{max}n^{1+\zeta/2}) \ , \\
Time^A(Distrib_Weighted) &= O(\beta(\zeta,\rho,\epsilon)\omega_{max}\log^3 n \cdot n^{1+\zeta/2}) \ , \\
Comm^S(Distrib_Weighted) &= O(|E|n^{\rho}) \ , \\
Comm^A(Distrib_Weighted) &= O(|E|n^{\rho} + \log^3 n \cdot n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}}) \ .
\end{aligned}$$

$\beta(\zeta,\rho,\epsilon)$ is defined by (10), and it is constant whenever ζ, ρ and ϵ are.

Like the result of Corollary 2.16, this result is important mainly for weighted graphs with $\omega_{max} = O(1)$.

Finally, consider a generalization of Protocol $Distrib\_ASP$ to weighted graphs. This protocol, henceforth referred to as Protocol $Distrib\_Weighted\_ASP$, first invokes Protocol $Distrib\_Weighted$ for constructing a spanner, and then constructs s BFS spanning trees of the spanner, rooted in the sources. In the way analogous to Theorem 2.18, we conclude the following.

Corollary 3.11 *For any sufficiently large integer n, any weighted graph $G = (V,E,\omega)$, any real numbers $0 < \zeta,\rho,\epsilon < 1$ such that $\zeta/2 + 1/3 > \rho > \zeta/2$, ρ, ζ are constant and $\epsilon = \Omega(\frac{1}{\mathrm{polylog}\ (n)})$, any subset $S \subseteq V$ of s vertices, the paths $P_{u,w} \subseteq E$ between all pairs of nodes $\{u,w\} \in S \times V$ can be computed using the following time and communication complexities in the synchronous and asynchronous settings.*

$$\begin{aligned}
Time^S(Distrib_Weighted_ASP) &= O(\beta(\zeta,\rho,\epsilon)\omega_{max}n^{1+\zeta/2} + s \cdot Diam(G)) \ , \\
Time^A(Distrib_Weighted_ASP) &= O(\beta(\zeta,\rho,\epsilon)\omega_{max}n^{1+\zeta/2}\log^3 n + s \cdot Diam(G)\log^3 n) \ , \\
Comm^S(Distrib_Weighted_ASP) &= O(|E|n^{\rho} + sn^{1+\zeta}) \ , \\
Comm^A(Distrib_Weighted_ASP) &= O(|E|n^{\rho} + sn^{1+\zeta} + n^{1+\rho+\frac{\zeta(\rho-\zeta/2)}{2}}\log^3 n) \ .
\end{aligned}$$

These paths satisfy

1. $L(P_{u,w}) \le (1+\epsilon)d_G(u,w) + \beta(\zeta,\rho,\epsilon)$.
2. $|\bigcup_{\{u,w\}\cap S \neq \emptyset} P_{u,w}| = O(\omega_{max}n^{1+\zeta})$.

$\beta(\zeta,\rho,\epsilon)$ is defined by (10) and it is constant whenever ζ, ρ and ϵ are.

Acknowledgements

The author is very grateful to David Peleg for motivating discussions, helpful comments and remarks, and to Uri Zwick for helpful comments and remarks.

References

[1] B. Awerbuch, B. Berger, L. Cowen, D. Peleg, Near-linear time construction of sparse neighborhood covers, *SIAM Journal of Computing*, Vol. 28, **1**, 263-277, 1998.

[2] B. Awerbuch, A. Baratz, and D. Peleg. Efficient broadcast and light-weight spanners. Unpublished manuscript, November 1991.

[3] D. Aingworth, C. Chekuri, P. Indyk and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). In *Proc. 7th ACM-SIAM Symp. on Discrete Algorithms*, Atlanta, Georgia, pp. 547-553, 1996.

[4] I. Althöfer, G. Das, D. Dobkin and D. Joseph. Generating sparse spanners for weighted graphs, *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, Lect. Notes in Comput. Sci., Vol. 447, pp. 26-37, Springer-Verlag, New York/Berlin, 1990.

[5] B. Awerbuch and G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Trans. on Information Theory*, IT-33(3):315-322, 1987.

[6] N. Alon, Z. Galil and O. Margalit. On the exponent of the all pairs shortest paths problem. *Journal of Computer and System Sciences*, 54:255-262, 1997.

[7] N. Alon, Z. Galil, O. Margalit and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, Pittsburgh, Pennsylvania, pp. 417-426, 1992.

[8] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead, In *Proc. 31st Symp. on Foundations of Computer Science*, pp. 514-522, October, 1990.

[9] Y. Afek and M. Ricklin. A paradigm for running distributed algorithms. In *Proc. 6th Workshop on Distributed Algorithms*, LNCS Vol. 647, pp. 1-10, 1992.

[10] B. Awerbuch. Complexity of network synchronization. *Journal of ACM*, 4:804-823, 1985.

[11] B. Awerbuch. Reducing complexities of the distributed max-flow and the breadth-first-search algorithms by means of network synchronization. *Networks*, 15:425-437, 1985.

[12] B. Awerbuch. Distributed shortest paths algorithms. In *Proc. 21st ACM Symp. on Theory of Computing*, pp. 230-240, May, 1989.

[13] B. Chandra, G. Das, G. Narasimhan and J. Soares, New Sparseness Results on Graph Spanners, *Proc 8th ACM Symposium on Computational Geometry*, pages 192-201, 1992.

[14] L. P. Chew. There is a planar graph almost as good as complete graph. In *2nd Symp. on Computational Geometry*, pp. 169-177, 1986.

[15] E. Cohen, Fast Algorithms for constructing t-spanners and paths of stretch t, in *Proc. 34th IEEE Symp. on Foundations of Computer Science*, IEEE, Piscataway, NJ, 1993, 648-658.

[16] E. Cohen, Polylog-time and near-linear work approximation scheme for undirected shortest paths, in *Proc. of 26th ACM Symp. on Theory of Computation*, pp. 16-26, 1994.

[17] D. Dor, S. Halperin, U. Zwick, All pairs almost shortest paths, *SIAM Journal on Computing*, 29 (2000), pp. 1740-1759.

[18] D. P. Dobkin, S. J. Friedman and K. J. Supowit. Delaunay graphs are almost as good as complete graphs, In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pp. 20-26, 1987.

[19] M. L. Elkin, Computing Almost Shortest Paths, in *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pp.53-63, Newport, Rhode Island, August, 2001.

[20] M. L. Elkin and D. Peleg. $(1+\epsilon,\beta)$-Spanner Constructions for General Graphs, to appear in *Proc. 33rd ACM Symp. on Theory of Computing*, Crete, Greece, July, 2001.

[21] G. N. Frederickson. Trade-offs for selection in distributed networks. In *Proc. 2nd Symp. on Theoretical Aspects of Computer Science*, LCNS Vol. 182, pp. 143-150, Springer, Berlin, 1985.

[22] R. G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-175, Lab. for Information and Decision Systems, MIT, Cambridge, MA, January, 1982.

[23] Z. Galil and O. Margalit. Witnesses for boolean matrix multiplication. *Journal of Complexity*, 9:201-221,1993.

[24] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54:243-254,1997.

[25] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*, SIAM, Philadelphia, PA, 2000.

[26] D. Peleg and A. Schäffer. Graph Spanners, *Journal of Graph Theory* **13** (1989), 99-116.

[27] D. Peleg and J.D. Ullman, An optimal synchronizer for the hypercube, *SIAM Journal of Computing* **18** (1989), pp. 740-747.

[28] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51:400-403, 1995.

[29] A. Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, Vol. IT-29, No. 1, January, 1983.