Massively Parallel Algorithms for Distance Approximation and Spanners

Amartya Shankha Biswas
CSAIL, MIT
USA
asbiswas@mit.edu

Michal Dory
ETH Zurich
Switzerland
michal.dory@inf.ethz.ch

Mohsen Ghaffari
ETH Zurich
Switzerland
ghaffari@inf.ethz.ch

Slobodan Mitrović
CSAIL, MIT
USA
slobo@mit.edu

Yasamin Nazari
Johns Hopkins University
USA
ynazari@jhu.edu

ABSTRACT

Over the past decade, there has been increasing interest in distributed/parallel algorithms for processing large-scale graphs. By now, we have quite fast algorithms—usually sublogarithmic-time and often poly$(\log\log n)$-time, or even faster—for a number of fundamental graph problems in the massively parallel computation (MPC) model. This model is a widely-adopted theoretical abstraction of MapReduce style settings, where a number of machines communicate in an all-to-all manner to process large-scale data. Contributing to this line of work on MPC graph algorithms, we present $poly(\log k) \in \text{poly}(\log\log n)$ round MPC algorithms for computing $O(k^{1+o(1)})$-spanners in the strongly sublinear regime of local memory. To the best of our knowledge, these are the first sublogarithmic-time MPC algorithms for spanner construction.

As primary applications of our spanners, we get two important implications, as follows:

- For the MPC setting, we get an $O(\log^2\log n)$-round algorithm for $O(\log^{1+o(1)} n)$ approximation of all pairs shortest paths (APSP) in the near-linear regime of local memory. To the best of our knowledge, this is the first sublogarithmic-time MPC algorithm for distance approximations.
- Our result above also extends to the Congested Clique model of distributed computing, with the same round complexity and approximation guarantee. This gives the first *sublogarithmic* algorithm for approximating APSP in *weighted* graphs in the Congested Clique model.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Theory of computation → Sparsification and spanners**; **Shortest paths**; **Massively parallel algorithms**.

SPAA '21, July 6–8, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8070-6/21/07…$15.00
https://doi.org/10.1145/3409964.3461784

KEYWORDS

Spanners; Shortest Paths; Massively Parallel Computation

ACM Reference Format:
Amartya Shankha Biswas, Michal Dory, Mohsen Ghaffari, Slobodan Mitrović, and Yasamin Nazari. 2021. Massively Parallel Algorithms for Distance Approximation and Spanners. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3409964.3461784

1 INTRODUCTION AND RELATED WORK

1.1 Massively Parallel Computation

Processing large-scale data is one of the indubitable necessities of the future, and one for which we will rely more and more on distributed/parallel computation. Over the past two decades, we have witnessed the emergence and wide-spread usage of a number of practical distributed data processing frameworks, including MapReduce [24], Hadoop [54], Spark [55] and Dryad [41]. More recently, there has also been increasing interest in building a corresponding algorithmic toolbox for such settings. By now, there is a de-facto standard theoretical abstraction of these frameworks, known as the *Massively Parallel Computation* (MPC) model. The model was introduced first by Karloff et al. [44] and refined later by Beam et al. [12] and Goodrich et al. [36].

MPC model. On a very high-level, the model assumes a number of machines, each with a memory capacity polynomially smaller than the entire data, which can communicate in an all-to-all fashion, in synchronous message passing rounds, subject to their memory constraints. More concretely, in the MPC model [12, 36, 44], we are given an input of size N which is arbitrarily distributed among a number of machines. Each machine has a memory of size $S = N^\alpha$ for some $0 < \alpha < 1$, known as the local memory or memory per machine. Since the data should fit in these machines, the number of machines is at least N/S, and often assumed to be at most $O(N/S \,\text{poly}(\log N))$. Hence, the global memory—that is, the summation of the local memories across the machines— is $\tilde{O}(N)$. The machines can communicate in synchronous message-passing rounds, subject to the constraint that the total amount of messages a machine can communicate per round is limited by its memory S.

In the case of graph problems, given a graph $G = (V, E)$ the total memory N is $O(|E|)$ words. Ideally, we would like to be able to work

with machines that have a small local memory, and still have only few rounds. However, the task gets more complex as one reduces the local memory. Depending on how the local memory compares with the number of vertices $n = |V|$, there are three regimes that have been studied in the MPC model:

- the *strongly superlinear* regime where $S \geq n^{1+\varepsilon}$ for a constant $\varepsilon > 0$,
- the *near-linear* regime where $S = \tilde{O}(n)$, and
- the *strongly sublinear* regime where $S = n^{\gamma}$ for a positive constant $\gamma < 1$.

We note that the algorithms that can work in the strongly sublinear memory regime are sometimes referred to as *scalable massively parallel algorithms*. Our focus will be on the more stringent, and also more desirable, regimes of near-linear and strongly sublinear memory.

1.2 Graph Problems and Massively Parallel Computation

At the center of the effort for building algorithmic tools and techniques for large-scale data processing has been the subarea of MPC *algorithms for graph problems*, e.g., [1, 3–5, 7, 8, 10, 12, 13, 15–17, 19–21, 23, 29–36, 38–40, 42, 45, 46, 52]. We discuss a very brief overview here. Please see the full version for a more detailed overview, with quantitative bounds.

Early on [12, 44], it was observed that MPC can simulate classic PRAM parallel algorithms (subject to conditions on the total amount of work) in the same time. This immediately led to poly$(\log n)$ round algorithms for a wide range of graph problems.

Since then, the primary objective in the study of MPC algorithms has been to obtain significantly faster algorithms, e.g., with constant or poly$(\log \log n)$ round complexity. This was achieved initially for the strongly super-linear memory regime, for many problems, and over the past few years, there has also been significant progress on near-linear and strongly sublinear memory regimes. In particular, there has been quite some progress for (A) global graph problems such as connected components, maximal forest, minimum-weight spanning tree, minimum cut, etc, [2, 4, 16, 33, 34, 46] and (B) local graph problems such as maximum matching approximation, graph coloring, maximal independent set, vertex cover approximation, etc [8, 9, 21, 21, 23, 30, 38, 46, 53].

Distance Problems. Despite the substantial progress on various graph problems, one fundamental category of graph problems for which the progress in MPC has been slower is distance computations and, more generally, distance-related graph problems. In particular, considering that a key criteria in the area is to obtain near constant (and especially $o(\log n)$) time algorithms, the following question has remained open.

Question: *Are there* poly$(\log \log n)$*-time MPC algorithms in the near-linear memory regime that compute all pairs shortest paths, or any reasonable approximation of them?*

In fact, to the best of our knowledge, prior to our work, there was no known MPC algorithm even with a sublogarithmic round complexity, for any non-trivial approximation factor, and even for single-source shortest paths.

The known algorithms provide only $poly(\log n)$ round complexity, which is considerably above our target. For instance, one can obtain a $poly(\log n)$ round algorithm for $1+\varepsilon$ approximation of single source shortest paths (SSSP) in the sublinear memory regime of MPC, by adapting the PRAM algorithm of Cohen [22]. While that algorithm requires $m^{1+\Omega(1)}$ global memory, recent PRAM results reduce that to $\tilde{O}(m)$ [6, 47]. Among more recent work, Hajiaghayi et al. [37] give an all pairs shortest path (APSP) algorithm that runs $O(\log^2 n)$ rounds in strongly sublinear local memory, for certain range of edge weights, and uses a large polynomial global memory, which depends on exponents of matrix multiplication.

To tackle the above question, we can naturally ask whether, in the allowed time, one can sparsify the graph considerably, without stretching the distances – as that sparse graph then can be potentially moved to one machine. This directly brings us to the notion of spanners, as introduced by Peleg and Schäffer [51], which are subgraphs with few edges that preserve distances, up to a certain multiplicative factor. A more detailed explanation follows.

Spanners. Given a graph $G = (V, E)$, a k-spanner is a (sparse) spanning subgraph H of G such that the distance between each pair of nodes in V on this subgraph H is at most k times their distance in the original graph G [50]. It is known that every graph admits a $(2k-1)$-spanner of size $O(n^{1+1/k})$, for $k \geq 1$, and assuming Erdös girth conjecture, this tradeoff is also tight. Spanners have found many applications in various models, such as, in constructing sparsifiers in the streaming model [43], designing work-efficient PRAM algorithms [28], and transshipment based distance approximations in PRAM [47] and Congested Clique [14]. In the context of strongly sublinear memory MPC, [26] used spanners to obtain an exponential speed up in preprocessing of distance sketches (though, still requiring polylogarithmic rounds). In particular, they achieve this by using spanners in order to simulate non-work-efficient PRAM algorithms in MPC without using extra memory. In general, spanners can be applied to reduce use of resources such as communication and memory for distance-related computation on denser graphs at the expense of accuracy.

We are not aware of any poly$(\log \log n)$ round MPC algorithm for computing spanners, in the sublinear or near-linear memory regime. Though one can obtain some (weaker) results from those of other computational models, as we will discuss later, when mentioning our results.

1.3 Our Contribution

In this paper, we provide the first sublogarithmic time MPC algorithms for distance approximations and for constructing spanners. Our spanner construction works in the strongly sublinear memory regime, while for using them in distance approximation we need to move to the near-linear memory regime. Since, as stated, spanners are versatile tools that have found applications in various distance-related graph problems, we are hopeful that our spanner construction should also help in a wider range of problems.

Spanner Constructions. Our main technical result, as stated below, provides a family of algorithms which provide a general trade-off between the number of MPC rounds and the stretch of the resulting spanner.

Theorem 1.1. *Given a weighted graph G on n vertices and m edges and a parameter t, there is an algorithm that runs in $O(\frac{1}{\gamma} \cdot \frac{t \log k}{\log(t+1)})$ rounds of* MPC *and w.h.p. outputs a spanner of size $O(n^{1+1/k} \cdot (t + \log k))$, and stretch $O(k^s)$ when memory per machine is $O(n^\gamma)$ for any constant $\gamma > 0$, and where $s = \frac{\log(2t+1)}{\log(t+1)}$. This algorithm uses total memory of $\tilde{O}(m)$.*

Since the theorem statement in Theorem 1.1 might be complex due to the number of variables involved, we next state several interesting corollaries of this theorem.

Corollary 1.2. *Given a total memory of $\tilde{O}(m)$ and memory per machine being $O(n^\gamma)$, for any constant $\gamma > 0$, there is an* MPC *algorithm that w.h.p. outputs a spanner with the following guarantees in terms of round complexity, stretch and size:*

1. *runs in $O(\log k)$ rounds, has $O(k^{\log 3})$ stretch and $O(n^{1+1/k} \cdot \log k)$ size;*
2. *runs in $O\left(2^{1/\varepsilon} \cdot \varepsilon \cdot \log k\right)$ rounds, has $O(k^{1+\varepsilon})$ stretch and $O(n^{1+1/k} \cdot (2^{1/\varepsilon} + \log k))$ size;*
3. *runs in $O(\frac{\log^2 k}{\log\log k})$ rounds, has $O(k^{1+o(1)})$ stretch and $O(n^{1+1/k} \cdot \log k)$ size;*
4. *runs in $O(\frac{\log^2 \log n}{\log\log\log n})$ rounds, has $O(\log^{1+o(1)} n)$ stretch and $O(n \cdot \log\log n)$ size.*

Moreover, and crucially for our distance approximation applications, our algorithms are applicable to weighted graphs.

PRAM algorithms for spanners such as Baswana-Sen [11] have depth at least $\Omega(k)$, which leads to MPC algorithms with $O(k)$ complexity. Our algorithms are significantly faster, at the price of a small penalty in the stretch. Another relevant prior work that we are aware of is a one of Parter and Yogev in the Congested Clique model, on constructing graph spanners in unweighted graphs [49] of $O(k)$ stretch. In the full version we show how by building on this work one can obtain the following result.

Theorem 1.3. *Given an unweighted graph G on n vertices and m edges, there is an algorithm that in $O(\frac{1}{\gamma} \cdot \log k)$* MPC *rounds w.h.p. outputs a spanner of size $O(n^{1+1/k} \cdot k)$ and stretch $O(k)$ when memory per machine is $O(n^\gamma)$, for any constant $\gamma > 0$. This algorithm uses a total memory of $\tilde{O}(m + n^{1+\gamma})$.*

It is worth noting that these round complexities are close to optimal, conditioned on a widely believed conjecture. This is because lower bounds in the distributed LOCAL model imply an $\Omega(\log k)$ conditional lower bound for the closely related problem of finding spanners with optimal parameters, i.e., $(2k-1)$-spanners with $O(n^{1+\frac{1}{k}})$ edges. Specifically, there is an $\Omega(k)$ lower bound for the problem in the LOCAL model [25], and as shown in [31], this implies $\Omega(\log k)$ conditional lower bound in MPC with sublinear memory, under the widely believed conjecture that connectivity requires $\Omega(\log n)$ rounds.[1] This conjecture is also called the cycle vs two cycles conjecture, as even distinguishing between one cycle of n nodes from 2 cycles of $n/2$ nodes, is conjectured to require $\Omega(\log n)$ rounds. Our algorithms have complexity of $\text{poly}(\log k)$ for near-optimal (up to a factor of $k^{o(1)}$) stretch which nearly matches the lower bound.

It is an interesting question whether spanners with *optimal* parameters can be also constructed in $\text{poly}(\log k)$ time. While our main goal is to construct spanners in standard MPC in $\text{poly}\log k$ rounds, we also provide two results that lead to near optimal parameters (stretch $O(k)$ and size $\tilde{O}(n^{1+1/k})$) at the expense of computational resources. As mentioned above, Theorem 1.3 constructs $O(k)$-spanners with $O(n^{1+1/k} \cdot k)$ edges in $O(\log k)$ rounds. This however comes at a price of extra total memory, and also only works for *unweighted* graphs. Additionally, in Section 3, we provide an algorithm that runs in $O(\sqrt{k})$ rounds and computes an $O(k)$-spanner with $O(\sqrt{k} n^{1+1/k})$ edges. While the parameters of the spanner are near-optimal, this comes at a price of significant increase in the round complexity (this is still much faster compared to previous algorithms that require $O(k)$ time).

Distance Approximations. As a direct but important corollary of our spanner construction in Corollary 1.2, in the near-linear memory regime we get an $O(\log^2 \log n)$-round algorithm for $O(\log^{1+o(1)} n)$ approximation of distance related problems, including all-pairs-shortest-paths. Alternatively, we can obtain a faster algorithm that runs in $O(\log\log n)$ rounds, if we relax the approximation factor to $O(\log^{1+\varepsilon} n)$ for a constant $\varepsilon > 0$. It is important to note that as the global memory in MPC is bounded by $\tilde{O}(m)$, we do not have enough space to store the complete output of APSP. Instead, we have one *coordinator* machine that stores the spanner, which implicitly stores approximate distances between all vertices. This machine can then compute distances locally based on the spanner. This gives the following (See the full version for full details).

Corollary 1.4. *There is a randomized algorithm that w.h.p computes $O(\log^s n)$-approximation for APSP in weighted undirected graphs, and runs in $O(\frac{t \log\log n}{\log(t+1)})$ rounds of* MPC*, when memory per machine is $\tilde{O}(n)$, $s = \frac{\log(2t+1)}{\log(t+1)}$, and $t = O(\log\log n)$. This algorithm uses total memory of size $\tilde{O}(m)$.*

Our work leaves an intriguing open question.

Open Problem. *Can we compute a constant, or perhaps even $1 + o(1)$, approximation of all pairs shortest paths in $\text{poly}(\log\log n)$ rounds in the near-linear local memory regime of* MPC*?*

Extension to Other Models. In addition to these two models, we show the generality of our techniques by extending our results to the PRAM and Congested Clique models.

Congested Clique. We can extend our spanner construction to work also in the distributed Congested Clique model (see the full version), leading to the following corollary for approximate shortest paths.

Corollary 1.5. *There is a randomized algorithm in the Congested Clique model that w.h.p computes $O(\log^s n)$-approximation for APSP in weighted undirected graphs, and runs in $O(\frac{t \log\log n}{\log(t+1)})$ rounds, where $s = \frac{\log(2t+1)}{\log(t+1)}$, and $t = O(\log\log n)$.*

As two special cases, this gives $O((\log n)^{\log 3})$-approximation in $O(\log\log n)$ time, and $O(\log^{1+o(1)} n)$-approximation in $O((\log\log n)^2)$

[1] The proof in [31] is for MPC algorithms that are component-stable, see [31] for the exact details.

time. It is important to note that these are the first *sub-logarithmic* algorithms that approximate weighted APSP in the Congested Clique model. Prior results take at least poly-logarithmic number of rounds or only work for *unweighted* graphs (see the full version for a detailed discussion).

PRAM. Our result also extends to the PRAM CRCW model. We would get the same PRAM depth as the MPC round complexity, with an additional multiplicative $\log^* n$ factor that arises from certain PRAM primitives (see the full version). We note that $O(k \log^* n)$ depth PRAM algorithms for $O(k)$ spanners were studied in [48], and [11]. To the best of our knowledge, before this work there were no known algorithms in PRAM with depth $o(k)$, even for suboptimal spanners.

2 OVERVIEW OF OUR TECHNIQUES

The main objective of our work is to design spanners that can be implemented in small parallel depth, e.g., $\text{poly}(\log \log n)$ rounds of MPC. For the sake of simplicity, we present our main ideas by considering unweighted graphs only. However, our main results apply to weighted graphs as well, and that's one of the key strengths of our approach. Our general approach will be to grow clusters of vertices by engulfing adjacent vertices and clusters. Additionally, each cluster will be associated with a rooted tree that essentially shows the "history" of the cluster's growth, with the root representing the oldest vertex of the cluster. The tree edges will all be part of the spanner, and the maximum depth of the tree will be called the *radius* of the corresponding cluster.

A general paradigm when attaching a cluster to either a vertex or another cluster will be to keep only one edge between them (the one of minimum weight) in the spanner, and discarding the remaining edges. The discarded edges will be spanned by the single included edge and the tree edges within the cluster(s). Thus, (disregarding many details) we can generally upper bound the stretch of any discarded edge by four times the maximum radius of any cluster. Moving forwards, we will talk about stretch and cluster radius interchangeably since they are asymptotically equal.

2.1 Fast Algorithm using Cluster-Cluster Merging

As a starting point, we design an algorithm based on *cluster-cluster merging*, that can be summarized as follows.

Cluster-cluster merging

Given an unweighted $G = (V, E)$ and parameter $k \geq 2$:

- Let C_0 be a set of n clusters ($|V| = n$), where each vertex of V is a single cluster.
- For $i = 1 \dots \log k$:
  - (A) Let C_i be a set of clusters obtained by sampling each cluster from C_{i-1} with probability $n^{(-2^{i-1}/k)}$.
  - (B) For each cluster $c \in C_{i-1} \setminus C_i$:
    - (i) If c has a neighbor $c' \in C_i$, add an edge between c and c' to the spanner, and merge c to c'.
    - (ii) Otherwise, for every $c' \in C_{i-1} \setminus C_i$ that is a neighbor of c add an edge between c and c' to the spanner.
- Add an edge between each pair of clusters remaining.

We start by letting each vertex be a singleton cluster, and in each subsequent iteration, a set of clusters is sub-sampled. These *sampled clusters* are "promoted" to the next iteration (the remaining clusters will not be considered in future iterations). However, the *sampled* clusters grow to absorb the neighboring *unsampled* ones. Conversely, each unsampled cluster joins its nearest neighboring sampled cluster. If no such neighboring (sampled) cluster exists, then we can conclude that, with high probability, there are very few neighboring clusters, and this allows us to deal with these problematic clusters by including edges to all their neighbors in the spanner.

Intuition for the Analysis. The intuition behind this approach is as follows. As in each iteration clusters are merged, the number of clusters decreases significantly between iterations. Since we are aiming for a spanner of size $\tilde{O}(n^{1+1/k})$, if C is the current set of clusters, we can allow each cluster in C to add $\frac{n^{1+1/k}}{|C|}$ adjacent edges to the spanner. This means that when the number of clusters reduces, we can allow each cluster to add significantly more edges to the spanner. This allows us to decrease the sampling probability p between different iterations drastically. As a consequence, the number of clusters decreases rapidly, which results in a $\log k$ complexity. We show that in each iteration the merging grows the stretch by a factor of 3, leading to an overall stretch of $O(3^{\log k}) = O(k^{\log 3})$.

2.2 Obtaining Better Stretch

We now discuss how to reduce the stretch, all the way to $k^{1+o(1)}$, while maintaining the same spanner size and increasing the number of iterations only by an extra $\log k$ factor.

Intuitively, the spanner algorithm described has sub-optimal stretch of $k^{\log 3}$ mainly since it is too aggressive in growing clusters in each iteration, and when two clusters are merged, the radius increases by a factor of three in one iteration. Thus a natural idea would be to grow the clusters much more gradually; that is, instead of merging a cluster c to cluster c' entirely in one iteration, c' could consume parts of c repeatedly (over multiple iterations). In other words, similar to the algorithm of [11] we can grow the clusters incrementally before performing a merge. We call this process *cluster-vertex merging*.

Cluster-vertex merging. The main difference between cluster-cluster and cluster-vertex merging can be outlined as follows: Firstly, in Step A use a smaller sampling probability, e.g. $n^{-1/k}$, instead of $n^{-2^{i-1}/k}$; Secondly, in Step Bi instead of merging two clusters, merge to a sampled cluster c', only the vertices *incident* to c' that *do not belong* to any sampled cluster. This essentially recovers the algorithm in [11], where it was shown that using only the cluster-vertex merging, one can construct a spanner of stretch $O(k)$, but this also requires $O(k)$ rounds. We investigate the interpolation between these two extremes by using a hybrid approach, which uses both incremental cluster growing (cluster-vertex merging) and cluster-cluster merging.

Combining the two approaches using cluster contractions. The basic idea is as follows. Instead of merging clusters in each iteration, we alternate between iterations where we apply the cluster-vertex merging approach and iterations where we apply the cluster-cluster merging approach. After a few iterations where we apply cluster-vertex merging, we merge clusters (an operation we also refer to as *contraction*). this allows to get an improved stretch but still keep a small running time.

Running the *cluster-vertex merging* procedure on a contracted graph also results in the cluster size growing much faster in each step. This accelerated cluster growth is the main reason for the speedup in our algorithm.

Intuitively, contractions result in *loss of information* about the *internal structure* of the cluster, and any time they are performed we incur extra stretch penalties. Our aim is to carefully balance out this loss with the cluster growth rate to get the best tradeoffs. We adjust this by tuning the sampling probabilities, and the intervals at which we perform contractions. This interpolation will allow us to reduce the stretch to $k^{1+o(1)}$, while still having $O(\text{poly}\log k)$ round (iteration) complexity. More generally, our various tradeoffs are a consequence of how much we grow the clusters before each contraction.

2.3 The General Algorithm for Round-Stretch Tradeoffs

Now, we provide a more detailed overview of our general algorithm. We will then look at specific parameter settings that lead to various tradeoffs. The algorithm proceeds in a sequence of *epochs*, where epoch i consists of t iterations as follows: In each iteration we subsample the clusters with probability p_i (to be defined later). As before, we then grow each sampled cluster by adding all neighboring vertices that have not joined any other sampled cluster. At this point we contract the clusters, and move to the next epoch.

Our general algorithms achieve a range of tradeoffs, parameterized by t, which is the number of growth iterations before a contraction, as we outline below.

(A) Assume that the number of *super-nodes* in the current graph is n' (originally n).
(B) Repeat t times (essentially performing t iterations of [11] with adjusted sampling probabilities):
   1. Perform *cluster-sub-sampling* with $p_i = \frac{n'}{n^{1+1/k}}$:
      Ensures that number of added edges $\frac{n'}{p_i} = O(n^{1+\frac{1}{k}})$.
   2. Perform incremental *cluster-vertex merging* to increase the radius of each cluster by one unit in the *current graph*:
      Note that the current graph may contain contracted clusters as *super-nodes*. Assuming that the internal radius of the *super-nodes* is r, the actual radius increase in the original graph can be as large as $2r + 1$ units (see Fig. 1).
(C) Perform *cluster-contraction* on the most recent clusters (clusters become super-nodes).
   This has the effect of reducing the number of clusters in the graph by a factor of p_S^t (probability of a specific cluster surviving for t repititions of *cluster-sub-sample*). So, we get $n' \leftarrow n' \cdot p_S^t$.

Following are some example results we obtain for specific values of parameter t.

- $(\mathbf{t = k})$: This is one extreme case where there is *no contraction*, and the cluster radius only increases by 1 for k repetitions, thus recovering the [11] result. This is the *slowest* algorithm, but it achieves optimal stretch $2k - 1$.
- $(\mathbf{t = \sqrt{k}})$: The immediate generalization involves exactly one contraction (analysis in Section 3), which occurs after the first set of $\sqrt{k}$ iterations. A new sampling probability is introduced after the contraction (reduced size of graph). According to this probability, the remainder of the algorithm is actually just a normal $\sqrt{k}$ stretch spanner construction. This algorithm attains $O(k)$ stretch, but it dramatically reduces the number of rounds to $O(\sqrt{k})$.
- $(\mathbf{t = 1})$: This is the other extreme case (analyzed in Section 4), where the three procedures (sub-sample, grow, contract) are performed repeatedly one after another, i.e., the algorithm contracts immediately after a single grow step. Consequently, the cluster radius grows exponentially, and the algorithm terminates after $\log k$ repetitions, yielding Item 1 of Corollary 1.2. This is the *fastest* algorithm, but only achieves stretch $O(k^{\log 3})$.

One special interesting setting, which we also use for application in distance approximation, is when we set $t = \log k$. This leads to stretch $k^{1+o(1)}$ and requires only $O(\frac{\log^2 k}{\log\log k})$ rounds. The general tradeoffs can be found in the full version of the paper.

2.4 Related Spanner Constructions

Our approach can also be seen as a new contraction-based spanner algorithm with a focus on parallel depth/round efficiency.

In the context of dynamic stream algorithms, another contraction-based algorithm was proposed by [2], but only for unweighted graphs. Their contractions are based on a different type of clustering formed based on vertex degrees. Algorithm of [2] has resemblance to a special case of our algorithm described in Section 4.1, but it has a weaker stretch. In particular, for a spanner of size $\tilde{O}(n^{1+1/k})$ they obtain stretch $k^{\log 5}$ in $\log k$ passes in streaming (where a pass corresponds to one round of communication in MPC) in *unweighted graphs*, whereas in the same time (pass/round) we obtain stretch $k^{\log 3}$ even for *weighted graphs*. Our general algorithm, as a special case, obtains the much stronger stretch of $k^{1+o(1)}$ in $O(\frac{\log^2 k}{\log\log k})$ iterations. These contraction-based algorithms can be seen as an

alternative approach to the well-known algorithm of [11]. Mainly, the goal in [11] is to compute optimal spanners of stretch $2k-1$, whereas our main goal is time efficiency. As a result our algorithm has a slightly weaker stretch/size tradeoff, but requires exponentially fewer iterations.

This general contraction-based framework may be of interest also in other related distance objects. A related work by [18] focuses on (α, β)-spanners and hopsets, and they use a similar type of clustering as one part of their construction. However, they connect the clusters differently and in since their main focus is not computation time, their algorithms run in polynomial time in most models. We hope that our fast clustering techniques also give insight into faster algorithms for these structures, perhaps at an extra cost in the stretch. Such improvements will have immediate implications for distance computation in various models.

After the submission of our initial manuscript in March 2020, we found out that an independent and concurrent work [27] obtained similar bounds for spanner construction in dynamic stream settings. Concretely, [27] (see Theorem 3, and set $g = \epsilon \cdot \log k$), leads to a streaming algorithm with $O(\epsilon \cdot \log k \cdot 2^{1/\epsilon})$ passes, spanner stretch $O(k^{1+\epsilon})$ and size $\tilde{O}(n^{1+1/k})$. This matches our MPC bound (replacing the number of passes by number of rounds), by setting $t = 2^{1/\epsilon}$ in Theorem 1.1. For weighted graphs, they can obtain a spanner with an extra factor of $\log W$ in the size (W is the aspect ratio), whereas in our construction the size is the same for weighted and unweighted graphs.

3 CLUSTER-CONTRACTION ALGORITHM FOR NEAR-OPTIMAL SPANNERS

As a warm-up, in this section we discuss an algorithm that takes $O(\sqrt{k})$ rounds and constructs a spanner with stretch $O(k)$ and size $O(\sqrt{k}n^{1+1/k})$ in unweighted graphs. This simple algorithm is already significantly faster compared to [11], requiring only $O(\sqrt{k})$ rounds instead of $O(k)$ rounds. Many analysis details are omitted from this warm-up section, but are fully presented in Section 4 and readers can skip this section without loss in continuity.

Given an unweighted graph $G = (V, E)$ we compute a spanner of size $O(\sqrt{k}n^{1+1/k})$ with stretch $O(k)$. The high-level idea is as follows. We run the algorithm of [11] *twice*: in the first phase we perform the first $t = \Theta(\sqrt{k})$ iterations of [11] and stop. We will form a supergraph $\hat{G} = (\hat{V}, \hat{E})$ defined by setting each cluster C_t to be a supernode, and will add an edge between supernodes $c_1, c_2 \in \hat{V}$ if the corresponding clusters are connected with at least one edge in the original graph G. Now, for $t' = \Theta(\sqrt{k})$, we compute a t'-spanner on G by running the [11] algorithm on the graph $\hat{G}$ as a black-box, this requires only $t' = O(\sqrt{k})$ additional iterations. Next we describe these two phases in detail.

First Phase: Start with $\mathcal{R}_0 = V$, and $V' = V$, $\mathsf{E} = E$. We will have a sequence of clusterings $\mathcal{R}_1 \supseteq \ldots \supseteq \mathcal{R}_t$ for a parameter t (we will set $t = \sqrt{k}$). V' and E will be the set of vertices and edges that are not yet *settled*. In each iteration of the first phase, V' is the set of vertices with one endpoint in E.

(1) Sample a set of clusters $\mathcal{R}_i$ by choosing each cluster in $\mathcal{R}_{i-1}$ with probability $n^{-1/k}$. Set $C_i = \mathcal{R}_i$.
(2) For all $v \in V'$:
   (i) If v is adjacent to a sampled cluster $\mathcal{R}_i$, then add v to the closest $c \in C_i$ and add one edge from $E(v, c)$ to the spanner. Discard (remove from E) all the edges in $E(v, c)$.
   (ii) If v is not adjacent to any sampled clusters in $\mathcal{R}_i$, then for *each* neighboring cluster $c' \in C_{i-1}$ add a single edge from $E(v, c')$ to the spanner, and discard all other edges between $E(v, c')$.
   (iii) Remove the intra-cluster edges: remove all the edges with both endpoints in C_i from E.

Second Phase: Define a supergraph $\hat{G} = (\hat{V}, \hat{E})$ by setting each cluster C_t to be a node in $\hat{G}$ and adding an edge in $\hat{E}$ for each pair of adjacent clusters in C_t. We then run a black-box algorithm for computing a $(2t'-1)$-spanner (e.g. by running the algorithm of [11]) on $\hat{G}$, for $t' = \sqrt{k}$.

Analysis Sketch. The high-level idea is that we are stopping the algorithm of [11] when there are $O(n^{1-1/\sqrt{k}})$ (or more generally $O(n^{1-t/k})$) clusters in C_t. This means that the supergraph $\hat{G}$ is significantly smaller, and now we can afford to compute a spanner with a better stretch on $\hat{G}$, for fixed size. The radius of the clusters at termination is $O(tt') = O(k)$, and thus the overall stretch is $O(k)$. To formalize this argument, we start with the size analysis.

Theorem 3.1. *The set of edges added by this algorithm is $O(\sqrt{k}n^{1+1/k})$.*

Proof. In the first phase we only add as many edges as the [11] algorithm does and the total number of edges is $O(tn^{1+1/k})$. In the second phase we will add $O(n^{1-t/k})^{1+1/t'})$ edges, which is $O(n^{1-1/k})$ for $t = t' = \sqrt{k}$. ∎

Stretch Analysis Sketch. We next provide a high-level overview of the stretch analysis. More details can be found in the full version. First, we show the following.

Lemma 3.2. *Let $(u, v) \in E$ be an edge not added to the spanner. At the end of iteration i of the first phase, (u, v) is either discarded (removed from E), or both its endpoints belong to clusters in C_i.*

Lemma 3.3. *At iteration i of the first phase, all clusters C_i have radius i.*

We use the properties described to prove that the stretch is $O(k)$. Intuitively, if we take an edge $(u, v) \in E$, by Lemma 3.2, by the end of the first phase this edge is either discarded, or its endpoints belong to clusters in C_t. If it was discarded during the first phase, it follows that there is a path of stretch $O(t) = O(\sqrt{k})$ between u and v based on the analysis of [11]. If (u, v) survived the first phase, its endpoints are either in the same cluster in C_t which implies a path of stretch $O(t)$ between them by Lemma 3.3, or they belong to different super-nodes in $\hat{V}$. In the latter case, since in the second phase we compute an $O(t')$-spanner in $\hat{G}$, there is a path of stretch $O(t') = O(\sqrt{k})$ between the super-nodes corresponding to u and v, which implies a stretch of $O(tt') = O(k)$ between u and v. See the full version for details.

4 CLUSTER-MERGING APPROACH

Following the general paradigm of Baswana and Sen [11], our algorithm proceeds in two phases. In the first phase, the algorithm creates a sequence of growing clusters, where initial clusters are singleton vertices. This is similar to the first phase in the [11] algorithm, but has the following crucial differences:

- In each epoch, a sub-sampled set of clusters (from the previous epoch) expand, by engulfing *neighboring clusters* that were *not* sub-sampled. In [11], the sub-sampled clusters only engulf *neighboring vertices*.
- Consequently, the radius of our clusters increase by a factor of 3 (roughly) in every epoch, and thus the radius after epoch i is $O(3^i)$. On the other hand, the cluster radius in [11] increments by 1 in each epoch, leading to a radius of i at the end of epoch i.
- The sub-sampling probability at epoch i is $n^{-\frac{2^{i-1}}{k}}$, i.e., the probabilities *decrease* as a *double exponential*, as opposed to [11], where the probabilities are always the same.
- Our algorithm proceeds for $O(\log k)$ epochs as opposed to k epochs in [11].
- The final stretch we achieve is $O(k^{\log 3})$, instead of $O(k)$ as in [11].

Based on the aforementioned sampling probabilities, the final number of clusters will be $n^{1/k}$. Subsequently, we enter the second phase, where we add edges between vertices that still have unprocessed edges and the final clusters. The final output is a set of edges E_S that represent the spanner.

The main benefit of our approach, compared to [11], is that it provides a significantly faster way of constructing spanners in MPC. Namely, the algorithm of [11] inherently requires $O(k)$ iterations and it is not clear how to implement it in $o(k)$ MPC rounds while not exceeding a total memory of $\tilde{O}(m)$. On the other hand, in the full version we show that each epoch of our algorithm can be implemented in $O(1)$ MPC rounds. This implies the above approach can be implemented in $O(\log k)$ MPC rounds.

4.1 Algorithm

In this section we describe the cluster-merging approach. We will use the following notation for inter-cluster edges.

Definition 4.1. *We define* $\mathsf{E}(c_1, c_2)$ *to be the set of edges in* E *that have one endpoint in cluster* c_1 *and the other endpoint in cluster* c_2*. We will also abuse this notation, and use* $\mathsf{E}(v, c)$ *to denote all edges between* vertex v *and cluster* c*.*

Before a formal description of our algorithm, we also need two more definitions.

Definition 4.2. *A* cluster c *is a set of vertices* $V_c \in V$ *along with a rooted tree* $T(c) = (V_c, E_c)$*. The "center" of the cluster is defined as the root of* T *(the oldest member), and the "radius" of the cluster is the depth of* T *(from the root).*

Definition 4.3. *A clustering of a graph* $G = (V, E)$ *is a partition of* V *into a set of disjoint clusters* $\mathbf{C}$*, such that for all* $c, c' \in \mathbf{C}$*, we have* $V_c \cap V_{c'} = \emptyset$*.*

We next describe the two phases of our algorithm.

Phase 1: The first phase proceeds through $\log k$ epochs. Let C_0 be the clustering of V where each $v \in V$ is a cluster. At a high-level, during epoch i, we sub-sample a set of clusters $\mathcal{R}^{(i)} \subseteq C^{(i-1)}$, we will connect the clusters to each other as follows: Consider a cluster c that is *not* sampled at epoch i. If c does *not* have a *neighboring sampled cluster* in $\mathcal{R}^{(i)}$, we merge it with *each* of the (un-sampled) neighboring clusters in $C^{(i-1)}$, using the lowest weight edge. On the other hand, if c has *at least one* neighboring sampled cluster, then we find the closest such cluster and merge it to c using the lowest weight edge, say e. Additionally, in the weighted case, we also add an edge to each of the other neighboring clusters, that are adjacent to c with an edge of *weight strictly lower than* that of e.

Throughout, we maintain a set E (initialized to E) containing the unprocessed edges. During each epoch, edges are removed from E. During execution, some edges from E are added to the set of spanner edges E_S, and some are discarded. Specifically, when we merge clusters c_1 and c_2, only the lowest weight edge in $\mathsf{E}(c_1, c_2)$ is added to the spanner E_S, and all other edges in $\mathsf{E}(c_1, c_2)$ are discarded from E (the notation $\mathsf{E}(\cdot, \cdot)$ is in Definition 4.1). We will use $\mathsf{E}^{(i)}$ to denote the state of the set E at the end of epoch i. During epoch i, we also construct a set of edges $\mathcal{E}^{(i)} \subseteq E_S$, containing the edges that are used to connect (merge) *sampled* old clusters with *un-sampled* clusters to form the new ones $C^{(i)}$.

At epoch i, for $i = 1 \ldots \log k$, we perform the following steps:

(1) Sample a set of clusters $\mathcal{R}^{(i)} \subseteq C^{(i-1)}$, where each $c \in C^{(i-1)}$ is chosen to be a member of $\mathcal{R}^{(i)}$ with probability $n^{-\frac{2^{i-1}}{k}}$. Initialize $\mathcal{E}^{(i)}$ to the subset of edges in $\mathcal{E}^{(i-1)}$ that are contained in some cluster $c \in \mathcal{R}^{(i)}$.
(2) Consider a cluster $c \in C^{(i-1)} \setminus \mathcal{R}^{(i)}$ that has a neighbor in $R^{(i)}$. Let $\mathcal{N}^{(i)}(c) \in \mathcal{R}^{(i)}$ be the *closest neighboring sampled cluster* of c.
   (a) Add the lowest weight edge $e \in \mathsf{E}(c, \mathcal{N}^{(i)}(c))$ to both $\mathcal{E}^{(i)}$ and E_S and remove the entire set $\mathsf{E}(c, \mathcal{N}^{(i)}(c))$ from E.
   (b) For all clusters $c' \in C^{(i-1)}$ adjacent to c with any edge of weight strictly less than e, add the lowest weight edge in $\mathsf{E}(c, c')$ to E_S and then discard all the edges in $\mathsf{E}(c, c')$ from E.
(3) Consider a cluster $c \in C^{(i-1)} \setminus \mathcal{R}^{(i)}$ that has *no* neighbor in $\mathcal{R}^{(i)}$. Let $C' \subseteq C^{(i-1)}$ be all the clusters of $C^{(i-1)}$ in the *neighborhood* of c. For each $c_j \in C'$ move the lowest weight edge in $\mathsf{E}(c, c_j)$ to E_S and discard all edges in $\mathsf{E}(c, c_j)$ from E.
(4) The clustering $C^{(i)}$ is formed by taking the clusters in $\mathcal{R}^{(i)}$, and then extending them using all the edges in $\mathcal{E}^{(i)}$ to absorb other clusters that are connected to $\mathcal{R}^{(i)}$ (using only edges in $\mathcal{E}^{(i)}$).
   - Specifically, let $c \in \mathcal{R}^{(i)}$ be a sampled cluster, and let $\Delta^{(i-1)}(c) = \{\bar{c} \in C^{(i-1)} \mid E(c, \bar{c}) \cap \mathcal{E}^{(i)} \neq \emptyset\}$ be the set of adjacent clusters that will be absorbed. Each such c results a new cluster $c' \in C^{(i)}$, where c' has the same root node as c, and the tree $T(c')$ is formed by attaching the trees $T(\bar{c})$ (for each $\bar{c} \in \Delta^{(i-1)}(c)$), to the corresponding leaf node of the tree $T(c)$, using the appropriate edge in $E(c, \bar{c}) \cap \mathcal{E}^{(i)}$ (by construction, there is exactly one such edge).
(5) Remove all edges $(u, v) \in \mathsf{E}$ where u and v belong to the same cluster in $C^{(i)}$. This set E at the end of the i^{th} epoch is denoted

$\mathsf{E}^{(i)}$. We then contract and form the new quotient graph (supergraph) and proceed to the next epoch.

Phase 2: In the second phase, let V' be the set of all endpoints of the remaining edges $\mathsf{E}^{(\log k)}$. For each $v \in V'$ and each $c \in C^{(\log k)}$, we add the lowest edge in $\mathsf{E}(v, c)$ to E_S before discarding the edges in $\mathsf{E}(v, c)$.

4.2 Analysis of Phase 1

We first show that, for each edge $e = (u, v)$ that is discarded (not added to E_S), there exists a path from u to v in E_S, of weight at most $k \cdot w_e$ (see Theorem 4.10), i.e., the edge e is *spanned* by existing spanner edges in E_S. Next, in Section 4.2.2, we show that the number of edges added to the spanner E_S during phase 1, is $O(n^{1+1/k} \log k)$ in expectation (see Theorem 4.13).

4.2.1 Stretch Analysis. We begin by providing some definitions used throughout the analysis, and then get into a formal stretch analysis.

Definition 4.4 (Weighted-Stretch Radius). *C is a clustering of weighted-stretch radius r with respect to an edge set E in a graph G if and only if*

(A) For all $c \in C$, the cluster c has radius at most r (equivalently, T_c has depth at most r).

(B) For each edge $e = (x, v) \in \mathsf{E}$ such that $x \in c \in C$, all edges on the path from x to the root of T_c have weight less than or equal to w_e.

Definition 4.5 (Cluster of a vertex). *For a vertex v, $\mathbf{c}^{(i)}(v)$ refers to the cluster of $C^{(i)}$ containing v.*

Definition 4.6 (Cluster center). *For a vertex v, $\mathcal{F}^{(i)}(v)$ denotes the center of $\mathbf{c}^{(i)}(v)$.*

First, an inductive argument shows that all the remaining edges in $\mathsf{E}^{(i)}$ are between the current set of clusters $C^{(i)}$.

Lemma 4.7. *During the execution of Phase 1, any edge $e = (u, v) \in \mathsf{E}$ at the end of epoch i is such that both end-points are members of distinct clusters in $C^{(i)}$.*

Proof. We prove this statement by induction.

Base case: Before the first epoch, $\mathsf{E} = E$ and all the edges have endpoints in C_0 since this is the set of all vertices.

Inductive hypothesis: Assume that at the beginning of epoch i, all edges in E have both endpoints in distinct clusters of $C^{(i-1)}$.

Induction: Towards a contradiction, assume that there is an edge $e = (u, v) \in \mathsf{E}$ that survives to the end of epoch i and has at least one endpoint that is not in any cluster of $C^{(i)}$. Without loss of generality, assume that this endpoint is v. Note that $\mathbf{c}^{(i-1)}(v)$ and $\mathbf{c}^{(i-1)}(u)$ exists by the inductive hypothesis. (See Definition 4.5 for the definition of $\mathbf{c}^{(i-1)}(\cdot)$.)

If $\mathbf{c}^{(i-1)}(v)$ is adjacent to any cluster in $\mathcal{R}^{(i)}$, then it would have been processed in Step 2. Therefore, some edge between $\mathbf{c}^{(i-1)}(v)$ and $r \in \mathcal{R}^{(i)}$ was added to $\mathcal{E}^{(i)}$. In this case, $\mathbf{c}^{(i-1)}(v)$ will be absorbed into a new cluster in $C^{(i)}$ (see Step 4), and consequently, v is also a member of $C^{(i)}$. Hence, $\mathbf{c}^{(i-1)}(v)$ was not adjacent to any cluster in $\mathcal{R}^{(i)}$.

So, $\mathbf{c}^{(i-1)}(v)$ was processed in Step 3. In this case, all edges in $\mathsf{E}(\mathbf{c}^{(i-1)}(v), \mathbf{c}^{(i-1)}(u))$ were discarded (one of the edges was added to the spanner E_S), and hence this case could not happen neither. This now leads to a contradiction (as Step 2 or Step 3 has to occur) and implies that both u and v belong to some clusters in $C^{(i)}$.

It remains to show that u and v belong to distinct clusters of $C^{(i)}$. But this follows directly from Step 5. ∎

Next, we argue inductively that in each epoch the cluster radius grows by a factor of 3.

Theorem 4.8. *At the end of epoch i, $C^{(i)}$ is a clustering of weighted-stretch radius (see Definition 4.4) at most $\frac{3^i-1}{2}$ with respect to the current set $\mathsf{E}^{(i)}$.*

Proof. We first prove Property (A) and then Property (B) of Definition 4.4. Each of the properties are proved by an inductive argument.

Property (A) of Definition 4.4. As a base case, notice that before the first epoch, each cluster has radius 0.

During epoch i, each cluster $c' \in C^{(i)}$ is a union of a cluster $c \in \mathcal{R}^{(i)} \subseteq C^{(i-1)}$, and some number of clusters in $C^{(i-1)}$ that are adjacent to c. The root of cluster c' remains the same as the root of c. However, the radius of c' becomes the depth of the new rooted tree, which can be at most three times the old radius plus one (for the edge connecting the adjacent cluster). Thus the radius (not necessarily strong) of the new larger cluster is at most $3r + 1$, where r is the radius of $C^{(i-1)}$. For an illustration, see Fig. 1 and consider the distance between the center of cluster c and vertex x. Assuming that the inductive hypothesis is satisfied for $C^{(i-1)}$ i.e., $r \le \frac{3^{i-1}-1}{2}$, we see that the radius of $C^{(i)}$ is at most $\frac{3^i-1}{2}$.

Note that this proof implies that Property (A) holds independently of Property (B).

Property (B) of Definition 4.4. As an inductive hypothesis, assume that the clustering $C^{(i-1)}$ has weighted-stretch radius $\frac{3^{i-1}-1}{2}$ with respect to $\mathsf{E}^{(i-1)}$. Now, consider an edge $e = (x, y) \in \mathsf{E}^{(i)}$, such that $x \in c' \in C^{(i)}$ and $y \notin c'$. Note that by Lemma 4.7, each edge $e \in \mathsf{E}^{(i)}$ is of this form, i.e., the endpoints of e belong to distinct clusters of $C^{(i)}$. According to Step 4, cluster c' was formed from a sampled cluster $c \in \mathcal{R}^{(i)}$ that *engulfed* the adjacent clusters in $\Delta^{(i-1)}(c)$

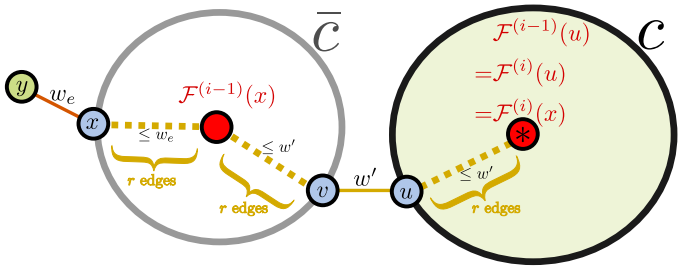

Figure 1: Sampled super-node c will engulf $\bar{c}$, which is also a supernode (not sampled). Both correspond to clusters of radius r.

Case $x \in \bar{c} \in \Delta^{(i-1)}(c)$: Let w' be the weight of the edge $(u,v) \in \mathcal{E}^{(i)}$ that connects $u \in T_c$ to $v \in T_{\bar{c}}$, and let w_e be the weight of the edge $e = (x,y)$ (see Fig. 1). We construct a path from x to $\mathcal{F}^{(i)}(x)$, by concatenating the following paths $x \longrightarrow \mathcal{F}^{(i-1)}(x) \longrightarrow v \longrightarrow u \longrightarrow \mathcal{F}^{(i)}(u) = \mathcal{F}^{(i)}(x)$. Since we know that the edge $e = (x,y)$ survived Step 2b, it must have weight at least as the edge (u,v), i.e., $w' \leq w_e$. By the inductive hypothesis for level $i-1$, all edges on the first two segments of the above path have weight at most w'. Similarly, the first two segments of the path only contain edges with weight at most w_e (again using the inductive hypothesis at $i-1$). Furthermore, the number of edges on the path is at most $r + r + 1 + r = 3r + 1$, where r is the weighted-stretch radius of clustering at the previous epoch.

Case $x \in c$: Since $e = (x,y) \in \mathsf{E}^{(i)} \implies (x,y) \in \mathsf{E}^{(i-1)}$, the inductive hypothesis implies that the path from x to the root of T_c (the center is $\mathcal{F}^{(i-1)}(x) = \mathcal{F}^{(i)}(x)$) only uses edges of weight at most w_e. ∎

Corollary 4.9. *The weighted-stretch radius of the final clustering $C^{(\log k)}$ is $\frac{k^{\log 3}-1}{2}$.*

Using Theorem 4.8, we show the following.

Theorem 4.10. *For all edges $e = (u,v)$ removed in Phase 1, there exists a path between u and v in E_S of weight at most $w_e \cdot k^{\log 3}$.*

Proof. Depending on when $e = (u,v)$ was removed from E, there are three cases to consider: Step 2, Step 3 and Step 5. Before analyzing these cases, note that by Lemma 4.7 after a cluster is formed no edge is removed from it in the subsequent epochs. This fact will be important in the rest of this proof as when we show that there is a path between u and v of weight at most $k^{\log 3} \cdot w_e$, we will show that this path belongs to a cluster (or to two adjacent ones). Hence, once this path belongs to a cluster it also belongs to E_S.

Case e was removed in Step 2 in epoch i. Let $e_1 = (x,y)$ be the edge that was kept between $\mathbf{c}^{(i-1)}(u)$ and $\mathbf{c}^{(i-1)}(v)$. Let $x \in \mathbf{c}^{(i-1)}(u)$ and $y \in \mathbf{c}^{(i-1)}(v)$. Then, by Theorem 4.8 there exists a path from u to x within $\mathbf{c}^{(i-1)}(u)$ of weight at most $2\frac{3^{i-1}-1}{2} \cdot w_e$. Similarly, there is a path between v and y within $\mathbf{c}^{(i-1)}(v)$ of weight at most $2\frac{3^{i-1}-1}{2} \cdot w_e$. Since E_S also contains the edge e_1 and $w_{e_1} \leq w_e$ by Steps 2a and 2b, we have that E_S contains a path between u and v of weight at most $(2(3^{i-1}-1)+1) \cdot w_e < 3^i \cdot w_e \leq k^{\log 3} w_e$.

Case e was removed in Step 3 in epoch i. This case is analogous to the previous one.

Case e was removed in Step 5 in epoch i. Let $c \in C^{(i)}$ be the cluster from which $e = (v_1, v_2)$ was removed. By construction, this edge is between two clusters c_1 and c_2 from $C^{(i-1)}$ that are merged with c in this epoch. Assume that $v_1 \in c_1$. Let $e_1 = (x_1, x)$ be the edge via which c_1 was merged to c, and let $x_1 \in c_1$. Also, let $e_2 = (y_2, y)$ be the edge with which c_2 was merged with c, and let $y_2 \in c_2$. Since e was not removed before Step 5, it means that when c_1 and c_2 got merged with c the edge e was not discarded in Step 2b. This in turn implies that $w_e \geq w_{e_1}$ and $w_e \geq w_{e_2}$.

Now, similar to the analysis of Step 2, using Theorem 4.8 we have that there is a path in E_S between v_1 and x_1 of weight at most $(3^{i-1}-1) \cdot w_e$. Also, there is a path between x and y in E_S of weight at most $(3^{i-1}-1) \cdot w_{e_1} \leq (3^{i-1}-1) \cdot w_e$. Finally, there is a path between v_2 and y_2 in E_S of weight at most $(3^{i-1}-1) \cdot w_e$. Combining these together, E_S contains a path between v_1 and v_2 of weight at most $(3 \cdot (3^{i-1}-1) + 2) \cdot w_e = (3^i - 1) \cdot w_e \leq k^{\log 3} \cdot w_e$, as desired. ∎

Stretch Analysis of Phase 2. Recall, that in the second phase, we let V' be the set of all endpoints of the *un-processed* edges in $\mathsf{E}^{(\log k)}$. Subsequently, we add the lowest edge in $\mathsf{E}(v,c)$ to E_S before discarding the edges in $\mathsf{E}(v,c)$, for each $v \in V'$ and $c \in C^{(\log k)}$.

Using the weighted stretch radius of the final clustering from Corollary 4.9, we can prove the following lemma, using an argument similar to Theorem 4.10. We omit the formal proof to avoid repetition.

Lemma 4.11. *For each edge $w_e = (v,c) \in V' \times C^{(\log k)}$ removed in Phase 2, there exists a path between u and v in E_S of weight at most $w_e \cdot k^{\log 3}$.*

4.2.2 Size Analysis. Next, we provide an upper-bound on E_S. First, we upper-bound the number of clusters in each $C^{(i)}$.

Lemma 4.12. *For each $i \leq \log k$, in expectation it holds $|C^{(i-1)}| \in O\left(n^{1-\frac{2^{i-1}-1}{k}}\right)$.*

Proof. A cluster c belongs to $C^{(i-1)}$ only if c was sampled to $\mathcal{R}_j$ in Step 1 for each $1 \leq j \leq i-1$.

This happens with probability $\prod_{j=1}^{i-1} n^{-\frac{2^{j-1}}{k}} = n^{-\frac{2^{i-1}-1}{k}}$. Therefore, $\mathbb{E}\left[|C^{(i-1)}|\right] = n^{1-\frac{2^{i-1}-1}{k}}$. ∎

Building on Lemma 4.12 we obtain the following claim.

Theorem 4.13. *During Phase 1, in expectation there are $O\left(n^{1+1/k} \cdot \log k\right)$ edges added to E_S.*

Proof. Steps 1, 4 and 5 do not affect E_S. Hence, we analyze only the remaining steps.

Consider epoch i. Fix a cluster $c \in C^{(i-1)}$ (which might or might not be in $\mathcal{R}^{(i)}$). We will upper-bound the number of edges that in expectation are added when considering c.

Let $p = n^{-\frac{2^{i-1}}{k}}$. Recall that each cluster c' is added from $C^{(i-1)}$ to $\mathcal{R}^{(i)}$ independently and with probability p. Order the clusters c' of $C^{(i-1)}$ adjacent to c in the non-decreasing order by the lowest-edge in $\mathsf{E}(c,c')$. Consider the first A among those sorted clusters.

Taking into account both Steps 2 and 3, an edge from c to the A-th cluster is added to E_S if and only if all previous clusters are not sampled, which happens with probability $(1-p)^{A-1}$. Hence, the expected number of edges added by c is upper-bounded by $\sum_{A=1}^{n}(1-p)^{A-1} < \sum_{A=1}^{\infty}(1-p)^{A-1} = \frac{1}{p}$. Hence, we have that in expectation the number of edges added to the spanner when considering c is $O(1/p)$.

∎

Size Analysis of Phase 2. Finally, As a corollary of Lemma 4.12, we see that in expectation $|C^{(\log k)}| \in O\left(n^{1/k}\right)$. Therefore, the number of edges added in Phase 2 is at most $|V'| \cdot |C^{(\log k)}| \in$

$O\left(n^{1+1/k}\right)$. This concludes our spanner construction, and yields the main theorem.

Theorem 4.14. *Given a weighted graph G, the cluster-merging algorithm builds a spanner of stretch $O(k^{\log 3})$ and expected size $O(n^{1+1/k} \cdot \log k)$, within $O(\log k)$ epochs.*

5 GENERAL TRADE-OFF BETWEEN STRETCH AND NUMBER OF ROUNDS

In this section, we provide an overview of an algorithm that combines ideas of Section 3 and Section 4, with the cluster-vertex merging concept that we described earlier. This gives us a general tradeoff between number of rounds, and stretch. For instance, we can construct a spanner with stretch $k^{1+o(1)}$ in $\frac{\log^2(k)}{\log\log(k)}$ rounds. At a high-level, the algorithm runs in a sequence of epochs, and each epoch performs t iterations of [11].

We can imagine the algorithm of [11] as being one extreme of this tradeoff (when $t = k$). The algorithm of Section 3 generalizes this, by splitting the k iterations of [11] over two epochs, each with $\sqrt{k}$ iterations. After the first set of $\sqrt{k}$ iterations (the first epoch), we *contract the most recent clusters*, and then repeat $\sqrt{k}$ iterations (the second epoch) on the contracted graph. Importantly, the second epoch uses different sampling probabilities – as if we were actually trying to construct a stronger $O(\sqrt{k})$ stretch spanner.

Meanwhile, the algorithm of Section 4 occupies the other extreme of our tradeoff. this algorithm immediately contracts after a single "[11]-like step". Consequently, each step now becomes an "epoch", and they all use different sampling probabilities. With $\log k$ epochs, this is the *"fastest"* algorithm in our tradeoff, and thus has the worst stretch.

We interpolate between these extremes by repeating the following two steps:

- In each epoch, we grow clusters, based on the cluster-vertex merging approach, till a certain radius t on the *quotient* graph, where each *super-node* in the graph is a contracted cluster from the previous epoch.
- At the end of an epoch, we contract the clusters of radius t to obtain the quotient graph for the next epoch, adjust the sampling probabilities and continue.

The parameter t can be varied, thus resulting in a family of algorithms that achieve our trade-offs. For instance, $t = 1$ corresponds to Section 4, $t = \sqrt{k}$ corresponds to Section 3, and $t = k$ brings us back to the algorithm of [11].

The idea behind the generalization is that now, the stretch (equivalently, the radius of clusters), rather than increasing by a *multiplicative* factor of 3 in each epoch (as it did in Section 4), now it grows by a factor of $(2t + 1)$ in each epoch. We again use the intuition that after each contraction, since the remaining graph is smaller in size, we can afford to grow clusters faster, and we adjust this rate by decreasing the sampling probabilities.

The algorithm and analysis are deferred to the full version.

6 IMPLEMENTATION IN MPC

In this section, we provide a brief sketch of MPC implementation. For more details we refer the readers to the full version. For implementing our algorithms we need to perform operations such as contractions, clustering, find minimum and merging. These can be implemented using standard subroutines such as sorting and aggregation ([36]) in $O(1/\gamma)$ rounds. The basic idea is that we can first sort the input (edges) based on their ID in such a way that edges corresponding to the same vertex will be stored in a contiguous set of machines. The merging and contraction operations can then be implemented by sorting the input multiple times (based on different "tuples"). For instance for performing contractions we can sort the edges based on cluster IDs of their endpoints and then relabel them based on the new cluster IDs. For operations such as find minimum and broadcast we can use an implicit aggregation tree with branching factor n^{γ}. At a high level, this mean in $O(1/\gamma)$ iterations (which is the depth of the aggregation tree) we aggregate the information on a set of machines and send them to *a parent machine*. For example, for broadcasting a message, the root of the tree will receive the message after $O(1/\gamma)$ iterations and then send back the message using the same (implicit) tree.

ACKNOWLEDGEMENTS

We thank Merav Parter, Michael Dinitz and Aditya Krishnan for fruitful discussions.

Funding. AS. Biswas is supported by MIT-IBM Watson AI Lab and research collaboration agreement No. W1771646, NSF awards CCF-1733808, IIS-1741137, Big George Ventures Fund Fellowship. M. Dory is supported in part by the Swiss National Foundation No. 200021_184735. M. Ghaffari is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program No. 853109. S. Mitrović is supported by the Swiss NSF grant No. *P*400*P*2_191122/1, MIT-IBM Watson AI Lab and research collaboration agreement No. W1771646, NSF award CCF-1733808, and FinTech@CSAIL. Y. Nazari is supported by NSF award CCF-190911.

REFERENCES

[1] Kook Jin Ahn and Sudipto Guha. 2015. Access to Data and Number of Iterations: Dual Primal Algorithms for Maximum Matching Under Resource Constraints. In *SPAA*. 202–211.
[2] K. J. Ahn, S. Guha, and A. McGregor. 2012. Analyzing graph structure via linear measurements. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 459–467.
[3] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. 2014. Parallel Algorithms for Geometric Graph Problems. In *Proc. Symposium on Theory of Computation (STOC)*. 574–583.
[4] Alexandr Andoni, Clifford Stein, Zhao Song, Zhengyu Wang, and Peilin Zhong. 2018. Parallel Graph Connectivity in Log Diameter Rounds. In *Proc. Foundations of Computer Science (FOCS)*. 674–685.
[5] Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2019. Log Diameter Rounds Algorithms for 2-Vertex and 2-Edge Connectivity. In *Proc. ICALP*, Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi (Eds.), Vol. 132. 14:1–14:16.
[6] Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2020. Parallel Approximate Undirected Shortest Paths Via Low Hop Emulators. In *Proc. Symposium on Theory of Computation (STOC)*.
[7] Sepehr Assadi. 2017. Simple round compression for parallel vertex cover. *arXiv preprint arXiv:1709.04599* (2017).
[8] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. 2019. Coresets Meet EDCS: Algorithms for Matching and Vertex

Cover on Massive Graphs. In *Proc. Symposium on Discrete Algorithms (SODA)*, Timothy M. Chan (Ed.). 1616–1635.

[9] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. 2019. Sublinear Algorithms for $(\Delta + 1)$ Vertex Coloring. In *Proceedings 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[10] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. 2019. Massively Parallel Algorithms for Finding Well-Connected Components in Sparse Graphs. In *Proc. Principles of Distributed Computing (PODC)*, Peter Robinson and Faith Ellen (Eds.). 461–470.

[11] Surender Baswana and Sandeep Sen. 2007. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms* 30, 4 (2007), 532–563.

[12] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication Steps for Parallel Query Processing. In *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*. 273–284.

[13] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 212–223.

[14] Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. 2017. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In *Proc. Symposium on DIStributed Computing (DISC)*, Vol. 91. 7:1–7:16.

[15] Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. 2019. Massively Parallel Computation of Matching and MIS in Sparse Graphs. In *Proc. Principles of Distributed Computing (PODC)*, Peter Robinson and Faith Ellen (Eds.). 481–490.

[16] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab S. Mirrokni. 2019. Near-Optimal Massively Parallel Graph Connectivity. In *Proc. Foundations of Computer Science (FOCS)*, David Zuckerman (Ed.). 1615–1636.

[17] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. 2019. Exponentially Faster Massively Parallel Maximal Matching. In *Proc. Foundations of Computer Science (FOCS)*, David Zuckerman (Ed.). 1637–1649.

[18] Uri Ben-Levy and Merav Parter. 2020. New (α, β) Spanners and Hopsets. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1695–1714.

[19] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. 2018. Approximating edit distance in truly subquadratic time: quantum and MapReduce. In *Proc. Symposium on Discrete Algorithms (SODA)*. 1170–1189.

[20] Sebastian Brandt, Manuela Fischer, and Jara Uitto. 2018. Breaking the Linear-Memory Barrier in MPC: Fast MIS on Trees with n^ϵ Memory per Machine. *arXiv preprint arXiv:1802.06748* (2018).

[21] Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. 2019. The Complexity of (Δ+1) Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. In *Proc. Principles of Distributed Computing (PODC)*. 471–480.

[22] Edith Cohen. 2000. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)* 47, 1 (2000), 132–166.

[23] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. 2018. Round Compression for Parallel Matching Algorithms. In *Proc. Symposium on Theory of Computation (STOC)*. 471–484.

[24] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[25] Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. 2008. On the locality of distributed sparse spanner construction. In *Proc. Principles of Distributed Computing (PODC)*. 273–282.

[26] Michael Dinitz and Yasamin Nazari. 2019. Massively Parallel Approximate Distance Sketches. In *Proceedings of International Conference on Principles of Distributed Systems (OPODIS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[27] Arnold Filtser, Michael Kapralov, and Navid Nouri. 2021. Graph spanners by sketching in dynamic streams and the simultaneous communication model. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM.

[28] Stephan Friedrichs and Christoph Lenzen. 2018. Parallel metric tree embedding based on an algebraic view on moore-bellman-ford. *Journal of the ACM (JACM)* 65, 6 (2018), 43.

[29] Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. 2019. Weighted Matchings via Unweighted Augmentations. In *Proc. Principles of Distributed Computing (PODC)*, Peter Robinson and Faith Ellen (Eds.). ACM, 491–500.

[30] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. [n.d.]. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *PODC*
vfill
.

[31] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. 2019. Conditional Hardness Results for Massively Parallel Computation from Distributed Lower Bounds. In *Proc. Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1650–1663.

[32] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrovic. 2019. Improved Parallel Algorithms for Density-Based Network Clustering. In *Proceedings of the International Conference on Machine Learning (ICML)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 2201–2210.

[33] Mohsen Ghaffari and Krzysztof Nowicki. 2020. Massively Parallel Algorithms for Minimum Cut. In *Proc. Principles of Distributed Computing (PODC)*. to appear.

[34] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. 2020. Faster Algorithms for Edge Connectivity via Random 2-Out Contractions. In *Proc. Symposium on Discrete Algorithms (SODA)*. 1260–1279.

[35] Mohsen Ghaffari and Jara Uitto. 2019. Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation. In *Proc. Symposium on Discrete Algorithms (SODA)*. 1636–1653.

[36] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, searching, and simulation in the MapReduce framework. In *Proc. ISAAC*. Springer, 374–383.

[37] MohammadTaghi Hajiaghayi, Silvio Lattanzi, Saeed Seddighin, and Cliff Stein. 2019. MapReduce meets fine-grained complexity: MapReduce algorithms for APSP, matrix multiplication, 3-SUM, and beyond. *arXiv preprint arXiv:1905.01748* (2019).

[38] Nicholas J. A. Harvey, Christopher Liaw, and Paul Liu. 2018. Greedy and Local Ratio Algorithms in the MapReduce Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Vienna, Austria). ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/3210377.3210386

[39] James W Hegeman and Sriram V Pemmaraju. 2015. Lessons from the congested clique applied to MapReduce. *Theoretical Computer Science* 608 (2015), 268–281.

[40] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. 2017. Efficient Massively Parallel Methods for Dynamic Programming. In *Proc. Symposium on Theory of Computation (STOC)*. 798–811.

[41] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[42] Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. 2019. Dynamic Algorithms for the Massively Parallel Computation Model. In *SPAA*. 49–58.

[43] Michael Kapralov and David Woodruff. 2014. Spanners and sparsifiers in dynamic streams. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 272–281.

[44] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A model of computation for MapReduce. In *Proc. Symposium on Discrete Algorithms (SODA)*. 938–948.

[45] Jakub Lacki, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. 2020. Walking Randomly, Massively, and Efficiently. In *Proc. Symposium on Theory of Computation (STOC)*.

[46] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. 2011. Filtering: a method for solving graph problems in MapReduce. In *SPAA*. 85–94.

[47] Jason Li. 2020. Faster Parallel Algorithm for Approximate Shortest Path. In *Proc. Symposium on Theory of Computation (STOC)*.

[48] Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. 2015. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. 192–201.

[49] Merav Parter and Eylon Yogev. 2018. Congested Clique Algorithms for Graph Spanners. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[50] David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach*. SIAM.

[51] David Peleg and Alejandro A Schäffer. 1989. Graph spanners. *Journal of graph theory* 13, 1 (1989), 99–116.

[52] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. 2016. Shuffles and Circuits: (On Lower Bounds for Modern Parallel Computation). In *SPAA*. 1–12.

[53] Vaclav Rozhon and Mohsen Ghaffari. 2020. Polylogarithmic-Time Deterministic Network Decomposition and Distributed Derandomization. In *Proc. Symposium on Theory of Computation (STOC)*.

[54] Tom White. 2012. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.".

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets