View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

Massively Parallel Approximate Distance Sketches

Michael Dinitz
Johns Hopkins University, Baltimore, MD, United States
mdinitz@cs.jhu.edu

Yasamin Nazari
Johns Hopkins University, Baltimore, MD, United States
ynazari@jhu.edu

Abstract

Data structures that allow efficient distance estimation (distance oracles, distance sketches, etc.) have been extensively studied, and are particularly well studied in centralized models and classical distributed models such as CONGEST. We initiate their study in newer (and arguably more realistic) models of distributed computation: the Congested Clique model and the Massively Parallel Computation (MPC) model. We provide efficient constructions in both of these models, but our core results are for MPC. In MPC we give two main results: an algorithm that constructs stretch/space optimal distance sketches but takes a (small) polynomial number of rounds, and an algorithm that constructs distance sketches with worse stretch but that only takes polylogarithmic rounds.

Along the way, we show that other useful combinatorial structures can also be computed in MPC. In particular, one key component we use to construct distance sketches are an MPC construction of the hopsets of [9]. This result has additional applications such as the first polylogarithmic time algorithm for constant approximate single-source shortest paths for weighted graphs in the low memory MPC setting.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms

Keywords and phrases Distance Sketches, Massively Parallel Computation, Distance Oracles, Single-Source Shortest Paths

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.35

Related Version The full version of this paper is available at `https://arxiv.org/pdf/1810.09027.pdf`.

Funding This work supported is in part by NSF awards CCF-1464239 and CCF-1909111.

Acknowledgements The authors are thankful to Goran Zuzic for helpful discussions.

1 Introduction

A common task when performing graph analytics is to compute distances between vertices. This has motivated the study of shortest path algorithms in essentially every interesting model of computation. We focus on two models which correspond to modern big-data graph analytics: Congested Clique [18] and Massively Parallel Computation (MPC) [3]. The MPC model in particular has recently received significant attention, as it captures many modern data analytics frameworks such as MapReduce, Hadoop, and Spark. So since these are important models of distributed storage and computation, and computing distances in graphs is an important primitive, we have an obvious question: in MPC or Congested Clique, can we compute distances between nodes sufficiently quickly to support important graph analytics?

While one side effect of our techniques is indeed a state of the art algorithm for shortest paths in MPC, the focus of this paper is on getting around the limitations of these models by allowing preprocessing of the (distributed) graph. We will first spend some time building a data structure known as *approximate distance sketches* (or an approximate distance oracle), which will then let us (approximately) answer any distance query using only 0, 1, or 2 rounds

© Michael Dinitz and Yasamin Nazari;
licensed under Creative Commons License CC-BY

of network communication (depending on the precise model). Thus after this preprocessing, anyone who is interested in analyzing the massive graph has access to approximate distances essentially for free, making this a powerful tool for distributed graph analytics. Moreover, rather than inventing a brand new structure, we show that we can repurpose centralized data structures (in particular the Thorup-Zwick oracle [25]) by computing them efficiently in these new distributed models. And since our algorithms are derived from centralized data structures we even allow for extremely efficient computation in addition to efficient communication.

So our focus is on how to compute these data structures efficiently, since once they are computed distance estimates become fast and easy. We show that in both the Congested Clique and the MPC models, we can compute oracles/sketches which essentially match the best centralized bounds in time that is only a small polynomial. In MPC, we can go even further and compute slightly suboptimal sketches in time that is only polylogarithmic. So while computing the data structure is still somewhat expensive, it is far more efficient than trivial approaches, and once it is computed, the analyst can receive approximate distances extremely quickly, allowing for low amortized cost or just the ability to do exploratory analysis without constantly waiting for expensive distance queries to complete.

Distance Oracles and Sketches. Even in many centralized applications, the time it takes to compute exact distances in graphs is undesriable, and similarly the memory that it would take to store all $\binom{n}{2}$ distances is also undesirable. This motivated Thorup and Zwick [25] to define the notion of an *approximate distance oracle*: a small data structure which can quickly report an approximation of the true distance for any pair of vertices. In other words, by spending some time up front to compute this data structure (known as the *preprocessing* step) and then storing it (which can be done since the structure is small), any algorithm used in the future can quickly obtain provably accurate distance estimates.

More formally, an approximate distance oracle is said to have *stretch t* if, when queried on $u, v \in V$, it returns a value $d'(u,v)$ such that $d(u,v) \leq d'(u,v) \leq t \cdot d(u,v)$ for all $u, v \in V$, where $d(u,v)$ denotes the shortest-path distance between u and v. The important parameters of an approximate distance oracle are the size of the oracle, the stretch, the query time, and the preprocessing time. For any constant k, Thorup and Zwick's construction (in the sequential setting) has expected size $O(kn^{1+1/k})$, stretch $(2k-1)$, query time $O(k)$, and preprocessing time $O(kmn^{1/k})$, where $n = |V|$ and $m = |E|$.

Since [25], there has been a large amount of followup work on improving the achievable tradeoffs, such as achieving query time of $O(1)$ with size $O(n^{1+1/k})$ [26, 7] or giving more refined bounds [20, 21]. However, with the notable exception of a very interesting construction due to Mendel and Naor [19], the vast majority of followup work has essentially been refinements and improvements to the approach pioneered by Thorup and Zwick. Thus understanding the Thorup-Zwick distance oracle is an important first step to understanding the limits and possibilities of distance oracles, and showing how to construct the Thorup-Zwick oracle in different computational models gives almost state-of-the-art bounds while also developing the basic tools and framework needed to design more sophisticated structures.

Importantly, the Thorup-Zwick distance oracle has the additional property that the data structure can be "broken up" into n pieces, each of size $O(kn^{1/k} \log n)$, so that the estimate $d'(u,v)$ can be computed just from the piece for u and the piece for v (the rest of the structure is unnecessary). These are called *distance sketches* or *distance labelings*, and motivated Das Sarma et al. [24] to initiate the study of Thorup-Zwick distance sketches in distributed networks, and in particular in the CONGEST model of distributed computing [22].

Models. As mentioned, in modern graph analytics we usually abstract away the communication graph by assuming that the datacenter storing the graph is sufficiently well-provisioned. This motivated two different but related models of distributed computation: Congested Clique [22] and MPC [3]. In the Congested Clique model an input graph of $G = (V, E)$ is given, and initially each node $v \in V$ only knows its incident edges. However, the underlying communication graph is an undirected clique, and in each round every node can send a message of $O(\log n)$ bits to any other node. This model was introduced by [22], and has been studied extensively in recent years. The second model that we consider is the *Massively Parallel Computation*, or MPC model. This model was introduced by [3] to model MapReduce and other realistic distributed settings, and is more general than earlier abstractions of MapReduce proposed by [15] and [12]. In this model there is an input of size N which is arbitrarily distributed over N/S machines, each of which has $S = N^\epsilon$ memory for some $0 < \epsilon < 1$. In the standard MPC model, every machine can communicate with every other machine in the network, but each machine in each round can have total I/O of at most S. Specifically, for graph problems the total memory N is $O(|E|)$ words. The low memory setting is the more challenging (but arguably more realistic) setting in which each machine has has $O(n^\gamma), \gamma < 1$ memory, where $n = |V|$, which we denote by MPC(n^γ). We also make the common assumption (e.g. [23, 3]) that machines have unique IDs that other machines can use for direct communication.

1.1 Our Results

In this paper we initiate the study of distance oracles and sketches in two popular computational models for "big data": Congested Clique and MPC. In addition, we show that our techniques can be used to give the first sublinear algorithm (and in fact polylogarithmic) for approximate single-source shortest paths for weighted graphs in (low memory) MPC, and moreover can be applied in straightforward ways to non-distributed models such as the streaming setting. We discuss our results for each model in turn. At a high level, Congested Clique turns out to be relatively easy: we can essentially just combine the known CONGEST algorithm [24] with a slightly modified hopset construction. For MPC, the natural approach is to simulate the Congested Clique algorithm, since it is known [5] that under certain density and memory conditions, Congested Clique algorithms can be simulated in MPC. However, this simulation requires at least $\Omega(n)$ memory per machine. Our task becomes much more challenging if we allow $o(n)$ memory per machine, which we refer to as the *low memory* setting. Designing algorithms for this setting forms the bulk of this paper.

Congested Clique. Since there is no memory restriction for Congested Clique, we assume that some node in the network is the *coordinator* at which the entire distance oracle will be stored (i.e., the machine with which users will interact with the distributed system). So at query time, the user can just query the coordinator locally (avoiding all network delay) rather than initiating an expensive distributed computation. The precise statements of our results are given in the full version and are somewhat technical, so for simplicity we state one particularly interesting corollary obtained by some specific parameter settings:

▶ **Theorem 1.** *Given a weighted graph $G = (V, E, w)$, for all $k \geq 2$ and constant $\epsilon > 0$, we can construct a distance oracle with stretch $(1+\epsilon)(2k-1)$, (local) query time $O(k)$, and space $O(kn^{1+1/k}\log n)$ w.h.p. in the Congested Clique model. If $k = O(1)$, then the number of rounds for preprocessing is*[1] *$\tilde{O}(n^{1/k})$, and if $k = \Omega(\log n)$ then the number of rounds is $\tilde{O}(\log(n))$.*

Note that after a limited amount of preprocessing, distance queries can be computed without any network access whatsoever. Moreover, the computational query time is also extremely small, so these queries are extraordinarily efficient in the context of distributed algorithms. As an interesting extension, we show that the message complexity of computing this distance oracle can be reduced by adding an additional preprocessing step of computing a graph spanner.

MPC. In Section 3 we discuss the MPC model, which is the heart of this paper. Since in the MPC model servers have small memory, it is impossible to fit an entire distance oracle at a single server as we did in the Congested Clique. So we instead focus on distance sketches. After the preprocessing algorithm, for each node $v \in V$, a distance sketch of size $O(kn^{1/k}\log n)$ will be stored and mapped to a machine with key v (this assumes that the memory at each server is at least $\Omega(kn^{1/k}\log n)$, which is reasonable in most settings). This means that after the preprocessing to construct these sketches, only two rounds of communication are needed for for approximating distance queries between a pair of nodes u and v: one for sending requests for the sketches of u and v and one for receiving them. We give the following result:

▶ **Theorem 2.** *Given a weighted graph $G = (V, E, w)$ with polynomial weights*[2] *and parameters $\rho \leq \gamma \leq 1, 1/k \leq \rho, 0 < \epsilon < 1$, we can construct Thorup-Zwick distance sketches with stretch $(2k-1)(1+\epsilon)$ and size $O(kn^{1/k}\log n)$ w.h.p. in $\tilde{O}(\frac{1}{\gamma}\cdot n^{1/k}\cdot\beta)$ rounds of MPC(n^γ), where $\beta = \min(O(\frac{\log n}{\epsilon})^{\log(k)+k}, 2^{\tilde{O}(\sqrt{\log n})})$. In particular, if $k = O(1)$ and ϵ is a constant, then w.h.p. we require $\tilde{O}(n^{1/k})$ rounds, and if $k = \Theta(\log n)$ then w.h.p. we require $2^{\tilde{O}(\sqrt{\log n})}$ rounds.*

In the above theorem the distance sketches have the same guarantees as the centralized Thorup-Zwick distance oracles. However, in MPC a polynomial round complexity, while possibly of theoretical interest, is generally considered not practical. So we give a different (but related) algorithm which achieves polylogarithmic round complexity, at the price of larger stretch.

▶ **Theorem 3.** *Consider a graph $G = (V, E)$ where $m = \Omega(kn^{1+1/k}\log n)$, for any $k \geq 2$. Then there is an algorithm in MPC(n^γ) (with $0 < \gamma < 1$) that constructs Thorup-Zwick distance sketches with stretch $O(k^2)$ and size $O(kn^{1/k}\log n)$ and with high probability completes in $O(\frac{k}{\gamma}\cdot(\frac{\log n\cdot\log k}{\epsilon})^{\log k+k-1})$ rounds.*

As a side effect of our techniques (which we discuss more in Section 1.2), we immediately get an algorithm for computing approximate single-source shortest paths (SSSP) in the MPC model, which is the problem of finding the (approximate) distances from a source node to all

[1] The notation $\tilde{O}(f(n))$ stands for $O(f(n)\cdot\text{polylog}(f(n)))$, e.g. it is suppressing polyloglog(n) terms in $2^{\tilde{O}(\log n)}$.

[2] This assumption can be relaxed using reduction techniques (e.g. from [9]) in exchange for extra polylogarithmic factors in the hopbound and construction time.

other nodes. Unlike in the Congested Clique, there do not seem to be any known nontrivial results for this problem in MPC. We first give an algorithm which computes a $(1+\epsilon)$-approximation in $n^{o(1)}$ time. Then we show that we can compute an $O(1)$-approximation in only polylogarithmic time, if we make an additional assumption about the density of the input graph. We will prove the following theorem in Section 3.2:

▶ **Theorem 4.** *Given a weighted undirected graph $G=(V,E,w)$ with polynomial weights, a source node $s \in V$, and $0<\gamma \le 1, 0<\epsilon<1$ we can compute $(1+\epsilon)$-approximate SSSP w.h.p. in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC with $\Theta(n^\gamma)$ memory per machine. Moreover, if $|E| \ge \Omega(n^{1+1/k}\log(n))$, we can compute $4k(1+\epsilon)$-approximate SSSP in $O(\frac{1}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k - 1})$ rounds of MPC(n^γ), where $1/k < \gamma \le 1, k \ge 2$. In particular, for $k = O(1)$ the algorithm runs in $O(\frac{1}{\gamma} \cdot (\frac{\log n}{\epsilon})^{O(1)})$ rounds.*

Note that while the round complexity is polylogarithmic, it may still be somewhat slow for certain applications: an analyst who has to wait polylogarithmic rounds for every distance query would essentially be unable to perform any analysis which depended on large numbers of distance queries. On the other hand, our main results on distance sketches allows us to pay this round complexity only once, for constructing the sketch.

Streaming. Finally, we provide an algorithm for constructing distance oracles in the multi-pass streaming model. This is essentially a side-effect of our main results for Congested Clique and MPC, but we include it for completeness. Our general results can be found in the full version. For the specific settings of constant or logarithmic stretch, we have:

▶ **Corollary 5.** *Given a graph $G=(V,E,w)$, there exists a streaming algorithm that constructs a Thorup-Zwick distance oracle of stretch $(2k-1)(1+\epsilon)$ of size $O(kn^{1+1/k}\log n)$ w.h.p. and expected space $O(n^{1+1/k} \cdot \log^2 n)$, such that if $k = O(1)$, w.h.p. we require $O(\log^k n)$ passes , and if $k = \Omega(\log n)$, w.h.p. we require $2^{\tilde{O}(\sqrt{\log n})}$ passes.*

Note that in case of $k = \Omega(\log n)$ we are in the so-called *semi-streaming* setting in which the total memory used is $O(n \cdot \text{polylog } n)$.

1.2 Our Techniques

Our main approach is to combine constructions of *hopsets* with efficient distributed constructions of Thorup-Zwick distance oracles/sketches. In particular, Das Sarma et al. [24] showed that Thorup-Zwick sketches could be computed in the CONGEST model, but the time depended on the graph diameter. So all that we really need to do is to reduce the diameter of the graph, since any CONGEST algorithm also works in the Congested Clique. This is what hopsets do: we discuss them in more detail in Section 2.2, but informally they allow us to reduce the diameter of the graph while preserving distances by adding in a carefully chosen set of weighted "shortcut" edges. Hopset constructions for the Congested Clique were given by Elkin and Neiman [9] (and more recently by[6]) so for Congested Clique we can essentially just combine result of [9] (or [6]) with [24] to get our result (modulo a small number of technicalities).

Moving to MPC introduces some significant technical difficulties, particularly when the space per machine is $o(n)$. Neither [24] nor [9] are written with MPC in mind, so we cannot simply "black-box" them as we could (mostly) in the Congested Clique. However, not surprisingly, both [24] and [9] use as a fundamental primitive a "restricted" version of the classical Bellman-Ford shortest-path algorithm that ends early, and it turns out that implementing this restricted Bellman-Ford is the main (although not the only) technical hurdle in adapting both of them to the MPC model.

When implementing restricted Bellman-Ford in low-memory MPC, the main difficulty is that since the memory at each server is $o(n)$, a single server cannot "simulate" a node in Bellman-Ford. It takes many machines to store the edges incident on any particular node, so we need to show that it is possible for many machines to simulate a single node in MPC without too much overhead. We show that this is indeed possible: Bellman-Ford and related algorithms can be implemented in low-memory MPC with very little additional overhead. Once we develop this tool, we argue that the hopsets of [9] can be constructed in low-memory MPC with essentially the same complexity as in the Congested Clique. Our implementation of Bellman-Ford and this hopset construction, as well as a few other primitives we develop for low-memory MPC (e.g., finding minimum or broadcasting on a range of machines), may be of independent interest.

Even after using hopsets, we would still need polynomial time for constructing constant stretch distance sketches. We overcome this issue and improve the running time using two ideas. First, we show that by relaxing the model to allow small additional total memory (either through extra space per machine or additional machines), we can run our algorithms in polylogarithmic number of rounds. So we just need to argue that there is a way of obtaining extra memory without actually changing the model assumptions. This is our second idea: by constructing a spanner we can sparsify the graph while keeping the memory per machine and number of machines the same. Thus from the perspective of the spanner, it will appear that we do indeed have "extra" memory. The idea of sparsifying the input to obtain extra resources has already proved to be powerful in related contexts (for example, [11] recently used spanners to give a work-efficient PRAM metric embedding algorithm). To the best of our knowledge, though, this idea has not yet appeared in the MPC graph algorithms literature.

1.3 Related Work

Distributed constructions of distance oracles and sketches have been studied extensively in the CONGEST model [24, 17, 10]. All of these algorithms have running times dependent on the graph diameter, while our algorithms run in time independent of the graph diameter. To the best of our knowledge, constructing distance oracles/sketches has not previously been studied for the Congested Clique or the MPC model. Similarly, hopsets have been used extensively in various models of computation for solving approximate SSSP ([14, 9]). Our result on hopset construction in low memory MPC also gives the first (approximate) SSSP algorithm in this model for weighted graphs (in Congested Clique there are more results known [9, 14, 4, 6], but these do not translate obviously to MPC when there is sublinear memory per machine). In a recent result, [6] gave an efficient Congested Clique algorithm that constructs hopsets of size $\tilde{O}(n^{3/2})$ with hopbound $O(\log^2(n)/\epsilon)$. Their hopsets are a special case of hopsets of [9]. In the full version, we explain how their algorithm applies to our Congested Clique result.

In the PRAM model, shortest path computation is well studied (e.g. [8, 9]), and it is known that many PRAM algorithms can be simulated in the MPC model ([15, 12]). However, most of these algorithms use $\omega(|E|)$ number of processors, in which case the simulations of [15] and [12] do not directly apply as they assume that the number of processors is at most the input size. As we argue in Section 3.1 we will still utilize an extension of this simulation. Another recent result for APSP in MapReduce by [13] also has the same drawback of using $\omega(n^2)$ processors. Result of [13] is based on matrix multiplication techniques, which are also well-studied in the PRAM model for computing APSP.

Finally, we note that distance problems have also been studied in related models such as the k-machine model ([16]). In this model [16] shows a low bound of $\Omega(n/k)$ for computing MST and shortest path trees, where k is the number of machines. To the best of our knowledge, the type of distance sketches that we consider here are not studied in the k-machine model.

2 Preliminaries and Notation

2.1 Notation

In a given weighted graph $G = (V, E)$, we denote the (weighted) distance between a pair of nodes $u, v \in V$ by $d_G(u, v)$. We may drop the subscript G when there is no ambiguity. We define the h *hop-restricted* distance between u and v to be the weight of the shortest path between u and v that uses at most h hops and denote this by $d^h(u, v)$.

We will denote the set of neighbors of a node $v \in V$ by $N(v)$. In a weighted graph G, we define the *shortest-path diameter* of G, denoted by Λ, to be the maximum over all $u, v \in V$ of the number of edges in the shortest $u - v$ path (so if the graph is unweighted this is the same as the diameter, but in weighted settings it can be larger than the unweighted diameter). Finally, a t-spanner of G is simply a subgraph which preserves distances up to a multiplicative t factor.

2.2 Algorithmic Building Blocks

In this section we describe the algorithms of [25], [24] and [9], that we will use in next section.

Thorup-Zwick Distance Oracle. In this section, we briefly describe the centralized construction of the well-known Thorup-Zwick distance oracle [25]. Given an undirected weighted graph $G = (V, E, w)$ and $k > 1$, in the preprocessing phase of their algorithm they first create a hierarchy of subsets $A_0, A_1, ..., A_{k_1}$ by sampling from nodes of V in the following manner: set $A_0 = V$, and for $1 \leq i \leq k - 1$, add every node $v \in A_{i-1}$ to the set A_i independently with probability $n^{-1/k}$. Set $A_k = \emptyset$ and for all $u \in V$ define $d(u, A_k) = \infty$. Let $B_i(u) = \{w \in A_i : d(u, w) < d(u, A_{i+1})\}$ for all $u \in V$ and $0 \leq i \leq k - 1$, where $d(u, A_i)$ is the minimum distance between u and a node in the set A_i, and set $B(u) = \cup_{i=0}^{k-1} B_i(u)$. We also denote the node that has the minimum distance to u among all nodes in A_i by $p_i(u)$ and call this the i-center of u, and so $d(u, A_i) = d(u, p_i(u))$. The distance sketch for u consists of $\{p_i(u)\}_{i=0}^{k}$, the set $B(u)$, and the corresponding distances between these nodes and u. The distance oracle is just the union of the sketches for all $u \in V$. Thorup and Zwick showed that this data structure has size $O(kn^{1+1/k} \log n)$ w.h.p., and access to these sketches is enough for approximating distances between every pair of vertices in $O(k)$ time with stretch $2k - 1$. In all the settings we consider, after preprocessing the distance oracle/sketches, we can *locally* perform the query algorithm of [25] in $O(k)$ time.

Next, we explain a distributed construction of Thorup-Zwick distance *sketches* as described by Das Sarma et al. [24] for the CONGEST model. The sampling phase can easily be done in distributed settings. Then for finding $p_i(v), 1 \leq i \leq k$ for all nodes $v \in V$, we will do the following: in iteration i, define a virtual source node s_i, and for all nodes in $u \in A_i$ add an edge between u and s_i where $w(u, s_i) = 0$. Then we will only need to run the Bellman-Ford algorithm from s_i, and after $O(k\Lambda)$ time every node $u \in V$ knows $p_i(u)$ and $d(u, A_i)$. Finally, for all $1 \leq i \leq k$ we need to compute the distance from $w \in A_i \setminus A_{i+1}$ to all the nodes v for which $w \in B(v)$. Simply running a distributed Bellman-Ford independently from all the sources $w \in A_i \setminus A_{i+1}$ would be slow since due to congestion limit on each edge we cannot run all these in parallel at the same time. However, [24] argue that this can be done in $O(\Lambda \cdot kn^{1/k} \log n)$ rounds in total (w.h.p), since each node v needs to forward messages in the runs of Bellman-Ford algorithm for a source w only if $w \in B(v)$. This means that, roughly speaking, each node v participates in $|B(v)| = O(kn^{1/k} \log n)$ runs of Bellman-Ford. Then by a simple round-robin scheduling scheme they show that running these Bellman-Fords for all sources in $A_i \setminus A_{i+1}$ can be done in $O(\Lambda \cdot kn^{1/k} \log n)$ without violating the congestion bound on each edge. For completeness we include a more detailed version of this algorithm in the full version.

Hopsets. For parameter $\epsilon, \beta > 0$, a graph $G_H = (V, H, w_H)$ is called a (β, ϵ)-hopset for the graph G, if in graph $G' = (V, E \cup H, w')$ obtained by adding edges of G_H, we have $d_G(u,v) \leq d^{\beta}_{G'}(u,v) \leq (1+\epsilon)d_G(u,v)$ for every pair $u, v \in V$ of vertices. The parameter β is called the *hopbound* of the hopset.

We first give a high level overview of the (sequential) hopset construction of [9] here. In their algorithm, they consider each distance scale $(2^k, 2^{k+1}], k = 0, 1, 2, ...$ separately. For a fixed distance scale $(2^k, 2^{k+1}]$ the algorithm consists of a set of *superclustering*, and *interconnection* phases. Initially, the set of clusters is $\mathcal{P} = \{\{v\}_{v \in V}\}$. Each cluster in $C \in \mathcal{P}$ has a cluster center which we denote by r_C. The algorithm uses a sequence $\delta_1, \delta_2, ...$ of distance thresholds and a sequence $\deg_1, \deg_2, ...$ of degree thresholds that determines the sampling probability of clusters. At the i-th iteration, every cluster $C \in \mathcal{P}$ is sampled with probability $1/\deg_i$. Let S_i denote the set of sampled clusters. Now a single shortest-path exploration of depth δ_i (weighted) from the set of centers of sampled clusters $R = \{r_C \mid C \in S_i\}$ is performed. Let $C' \in \mathcal{P} \setminus S_i$ be a cluster whose center $r_{C'}$ was reached by the exploration and let r_C be the center in R closest to r'_C. An edge $(r_C, r_{C'})$ with weight $d_G(r_C, r_{C'})$ is then added to the hopset. A supercluster $\hat{C}$ with center $r_{\hat{C}} = r_C$ is now created that contains all the vertices of C and the clusters C' for which a hopset edge was added. In the next stage of iteration i, all clusters within distance $\delta_i/2$ of each other that have not been superclustered at iteration i will be interconnected. In other words, a *separate* exploration of depth $\frac{\delta_i}{2}$ is performed from each such cluster center r_C and if center of cluster C' is reached, an edge (r_C, r'_C) with weight $d_G(r_C, r_{C'})$ will be also added to the hopset. The final phase of their algorithm only consists of the interconnection phase. We denote the hopset edges added for distance scale $(2^k, 2^{k+1}]$ by H_k. For completeness, we review this algorithm in more detail and explain the exact parameters in the full version.

One important property of this hopset construction (proved in Lemma 3.3 of [9]) that we will need for our analysis in Section 3) is the following:

▶ **Lemma 6** ([9]). *In the i-th iteration of a given distance scale $(2^k, 2^{k+1}]$, for each node $v \in V$, w.h.p. the number of explorations of interconnection phase that visit v is at most $O(deg_i \cdot \log n)$, where deg_i is the sampling probability of the superclustering phase.*

Now we turn our attention to efficient construction of hopsets in distributed settings also proposed by [9]. Note that each superclustering phase can be performed by a distributed Bellman-Ford exploration of depth δ_i. For an interconnection phase, a separate distributed Bellman-Ford explorations of depth $\delta_i/2$ from cluster centers is performed. These Bellman-Ford algorithms can easily be implemented sequentially, however, in distributed settings, $O(n)$ rounds may be needed for each of the explorations of the larger scales. To overcome this issue, [9] propose to use the hopsets $\cup_{\log \beta - 1 < j \leq k-1} H_j$, for constructing hopset edges H_k. More precisely, they observe that for any pair of nodes with distance less than 2^{k+1}, hopsets $\cup_{\log \beta - 1 < j \leq k-1} H_j$ provide a $(1+\epsilon)$-stretch approximate shortest path with $2\beta + 1$ hops between these pair of nodes. In other words, it is enough to run each Bellman-Ford exploration only for $O(\beta)$ rounds.

3 Distance Sketches in Massively Parallel Computation Model

In this section we will focus on the MPC model. First we provide MPC algorithms for constructing distance sketches that have the same guarantees (with respect to the stretch/size tradeoff) as the centralized construction of Thorup-Zwick that run in polynomial (or slightly subpolynomial) time. Then in Section 3.1 we show how we can bring down the running time to polylogarithmic in exchange for a loss in accuracy.

First, we note that it is known from [5] that for *dense graphs* with $O(n^2)$ edges every Congested Clique algorithm (in which nodes use local memory of $O(n)$) can be implemented in the MPC(n) model. Therefore, when memory per machine is $\Omega(n)$ and the graph is dense all our Congested Clique results also hold, except that we store the distance sketches rather than a central distance oracle. The more interesting case is when memory per machine is strictly sublinear in n. For the rest of this section we will turn our attention to the case where the memory is n^γ, where $0 < \gamma \geq 1$ (i.e., strictly sublinear). For simplicity we assume that we can store the sketches in a single machine. Namely, we require $\tilde{O}(n^{1/k})$ memory per machine for stretch $O(k)$ distance sketches. This assumption can be relaxed (and in exchange the query algorithm will take $O(k)$ rounds instead of 2 rounds).

One main subroutine that we need is the *restricted Bellman-Ford* algorithm. We then need to run many instances of this algorithm in parallel and handle other technicalities both for constructing hopsets, and then the distance sketches. First, we require following subroutines that will allow us to simulate one round of Bellman-Ford in MPC(n^γ):

Sorting [12]. Given a set of N comparable items, the goal is to have the items sorted on the output machines, i.e. the output machine with smaller ID holds smaller items.

Indexing [1]. Suppose we have sets $S_1, S_2, ..., S_k$ of N items stored in the system. The goal is to compute a mapping f such that $\forall i \in [k], x \in S_i$, x is the $f(S_i, x)$-th element of S_i. After running this algorithm the tuple $(x, f(S_i, x))$ is stored in the machine that stores x.

Find Minimum (x, y). Finds the minimum of N values stored over a contiguous set of machines given ID x of the first machine and ID y of the last machine.

Broadcast (b, x, y). Broadcasts a message b to a contiguous group of machines given ID x of the first machine and ID y of the last machine.

The sorting and indexing subroutines can be performed in $O(1/\gamma)$ rounds of MPC(n^γ) ([1, 12]). We argue that we can solve the Find Minimum and Broadcast problems also in $O(1/\gamma)$ rounds of MPC(N^γ) in the following theorem. At a high-level we use an *implicit* aggregation tree of depth $O(\log_{N^\gamma} N) = \frac{1}{\gamma}$.

▶ **Theorem 7.** *Given N items over a contiguous range of machines x to y, subroutines Find Minimum(x, y) can be implemented in $O(1/\gamma)$ rounds of MPC(N^γ). Moreover, the subroutine Broadcast(x, y) can also be implemented in $O(1/\gamma)$ rounds of MPC(N^γ).*

Proof. We will first define a rooted *aggregation tree* $\mathcal{T}$ with branching factor N^γ where the machines $M_x, ..., M_y$ are placed at the leaves (here M_x denotes the machine with ID x). W.l.o.g assume that the machines in this range have increasing and sequential IDs. Note that we don't need to store this tree explicitly, and we only need each node to know its parent. Consider level ℓ of the tree (leaves have $\ell = 0$). Each node in this level is a machine associated with the label ℓ. For each node in level $\ell - 1$ that has the i-th machine in its subtree, we set as its parent $M_{p(i,\ell)}$ where $p(i,\ell) = x + \lfloor \frac{i}{N^{\ell\gamma}} \rfloor$. Thus each machine can compute its parent given the label ℓ. Similarly, each machine can compute the indices of its children (as a range). In other words, at each level ℓ, we assign each group of N^{γ} nodes of this tree to a parent node at level $\ell + 1$.

The algorithm Find Minimum proceeds as follows: at each round ℓ, each machine first computes minimum over its the values it knows, and then sends the outcome to the parent machine. Finally, the minimum will be computed and stored at the root machine, which may forward the value to another destination. The algorithm Broadcast will similarly use an aggregation tree, but this time it routes the message top-down. First message b is sent to the first machine M_x, and then starting from M_x in each round any machine that receives message b sends this value to all of its children, which can be determined from the machine's ID and y. Eventually all the machines at the leaves will receive b. The number of rounds each of these subroutines take are the height of the aggregation tree which is $O(\log_{N^\gamma} N) = \frac{1}{\gamma}$. ◀

Running the (restricted) Bellman-Ford algorithm in MPC is not as straightforward as it is in the Congested Clique. One challenge is that for high-degree nodes, the edges corresponding to a single node are distributed over a set of machines. Therefore, for each round of Bellman-Ford these machines must communicate for computing and updating the distance estimates. Another hurdle is the fact that since nodes have different degrees, we do not have the range in which edges corresponding to a given node are stored a priori. To overcome these challenges we need to use the described subroutines, and for that we need to perform some preprocessing to append each edge with a tuple that we will describe shortly.

We will show how we can create and maintain the following setting: Given a graph $G = (V, E)$, the goal is to store all the edges incident to each node v in a contiguous group of machines, which we denote by $M(v)$. More precisely, let $M_1, ..., M_P$, where $P= O(\frac{m}{n^\gamma})$, be the list of machines ordered by their ID, and let $v_1, ..., v_n$ be the list of vertices sorted by their ID. $M(v_i)$ consists of the i-th smallest contiguous group of machines, such that $|M(v_i)| = \lceil \frac{\deg(v_i)}{n^\gamma} \rceil$.

Throughout the algorithm, let $M_{(u,v)}$ denote the machine that stores the edge (u, v). Also, for all $u \in V$, let r_u be the first machine in $M(u)$, and for any edge $(u, v) \in E$ let $i_u(v)$ be the index of (u, v) (based on the lexicographic order) among all the edges incident to v. We need to compute and store the following information at $M_{(u,v)}$: $\deg(u), \deg(v)$, r_u, r_v, i_u, i_v (here by storing r_u we mean ID of r_u, and for simplicity we refer to $i_u(v)$ as i_u). We first explain how these labels can be computed for all edges in $O(\frac{1}{\gamma})$ rounds in the following lemma.

▶ **Lemma 8.** *Let $M_{(u,v)}$ be the machine that stores a given edge (u, v). We can create tuples of the form $((u, v), \deg(u), \deg(v), r_u, r_v, i_u, i_v)$, stored at $M_{(u,v)}$ for all edges in $O(\frac{1}{\gamma})$ rounds in MPC(n^γ), where $\gamma < 1$.*

Proof. Let $N(v)$ be the set of edges incident on node v. Without loss of generality, let us assume that both tuples of form (u, v) and (v, u) are present in the system for each edge and we assume $(u, v) \in N(u)$ and $(v, u) \in N(v)$ (note that the graph is still undirected). First, we use the indexing subroutine of [1] on the sets $\{N(v)\}_{v \in V}$ to store index i_u at $M_{(u,v)}$ and index i_v at $M_{(v,u)}$. After this step tuples of form $((u, v), w(u, v), i_u)$ are stored at $M_{(u,v)}$.

Then we sort the tuples based on edge IDs lexicographically, using sorting algorithm proposed in [12]. This will result in the setting described above in which edges incident to each node u are stored in a contiguous group of machines $M(u)$. Now in order to compute $\deg(u)$, machines will check whether they are the last machine in $M(u)$ either by scanning their local memory or communicating with the next machine. Then the last machine in

$M(u)$ sets $\deg(u)$ to the maximum index i_u it holds. This machine can also compute r_u, ID of the first machine in $M(u)$ (using $\deg(u)$), and then broadcasts $\deg(u)$ and r_u to all machines in $M(u)$. At the end of these computations, each tuple $((u,v), w(u,v), i_u)$ will be replaced by the tuple $((u,v), w(u,v), r_u, i_u, \deg(u))$. Next, we sort these tuples again but this time based on the ID of the smallest endpoint. In other words, for each edge $(u,v) \in E$, both tuples $((u,v), w(u,v), i_u, \deg(u))$ and $((v,u), w(v,u), i_v, \deg(v))$ will be at the same machine. Now we can easily merge these two tuples to create tuples of form $((u,v), w(u,v), i_u, i_v, \deg(u), \deg(v))$. ◀

After computing the tuples, we use the sorting subroutine again to redistribute the edges into the initial setting of having contiguous group of machines $M(u)$ for all $u \in V$. After these preprocessing steps, we are ready to perform updates required for the restricted Bellman-Ford algorithm. A summary of this algorithm is presented in Algorithm 1.

Algorithm 1 Restricted Bellman-Ford in MPC(n^γ).

```
Input  : Graph $G = (V, E)$ distributed among machines $M_1, ..., M_P$ and source $s$.
Output : $h$-hop restricted distances from the source $s$ to all nodes $u \in V$, $d^h(s, v)$.
1 Create the tuple $((u,v), i_u, i_v, r_u, r_v, \deg(u), \deg(v))$ at $M_{(u,v)}$ for each edge
   $(u,v) \in E$ (by Lemma 8).
2 Sort the edges lexicographically so that edges incident to $v$ are stored in a contiguous
   group of machines $M(v)$ (by [12]).
3 for $i = 0$ to $h$ do
4     for $v \in V$ do
5         Compute $\hat{d}(s,v)$ by finding (using Theorem 7 $\min_{u \in N(v)} \hat{d}(s,u) + w(u,v)$).
6         Broadcast updated distances to everyone in $M(v)$ (also by Theorem 7).
7         Each machine in $M_{(v,u)}$ sends $\hat{d}(s,v)$ to $M_{(u,v)}$ (located at $r_u + \lfloor \frac{i_u}{n^\gamma} \rfloor$).
```

▶ **Theorem 9.** *Given a graph $G = (V, E)$ and a source node $s \in V$ the restricted Bellman-Ford algorithm (Algorithm 1) computes distances $d^h(s, v)$ for all $v \in V$ in $O(\frac{h}{\gamma})$ rounds of MPC(n^γ).*

Proof. After storing the tuples $(i_u, i_v, r_u, r_v, \deg(u), \deg(v))$ at $M_{(u,v)}$ for each $(u,v) \in E$, the restricted Bellman-Ford algorithm proceeds as follows: in each round, for each node v, we first find the minimum distance estimate for v and send it to r_v. Then r_v will broadcast the minimum distance found to all the machines in $M(v)$. By Theorem 7 both of these operations take $O(1/\gamma)$ rounds. Then for each $(v,u) \in N(v)$, $M_{(v,u)}$ sends the updated distance directly to $M_{(u,v)}$, which islocated at index $r_u + \lfloor \frac{i_u}{n^\gamma} \rfloor$. All the operations for each of the h iterations of Bellman-Ford take $O(1/\gamma)$ rounds. ◀

We now need to argue that hopsets of [9] can be constructed in MPC(n^γ). We show this in the following theorem. Here we assume that the weights are polynomial in n, which is not unrealistic since in MPC the total memory is assumed to be $\tilde{O}(m)$ bits.

▶ **Theorem 10.** *For any graph $G = (V, E, w)$ with n vertices, and parameters $\rho \leq \gamma \leq 1, 1 \leq \kappa \leq (\log n)/4, 1/2 > \rho \geq 1/\kappa$ and $0 < \epsilon < 1$, there is an algorithm in MPC(n^γ) model that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}} \log n)$ in $O(\frac{n^\rho}{\rho} \cdot \log^2 n \cdot \beta)$ rounds whp, where $\beta = O((\frac{\log n}{\epsilon} \cdot (\log \kappa + 1/\rho))^{\log \kappa + \frac{1}{\rho}})$.*

Proof. The distributed implementation of this algorithm just performs multiple restricted Bellman-Ford algorithms in each phase. Recall also that it is enough to run each of the Bellman-Ford instances only for $O(\beta)$ rounds, by using the fact that for constructing hopset edges H_k for a distance scale of $(2^k, 2^{k+1}]$, the hopsets $\cup_{\log\beta-1<j\leq k-1} H_j$ can be used recursively.

Each round of a single Bellman-Ford algorithm can be simulated in $O(\frac{1}{\gamma})$ rounds of MPC(n^γ) by running the algorithm of Theorem 9 on each node, whose edges may be distributed over multiple machines. Hence each superclustering phase can be performed in $O(\frac{\beta}{\gamma})$ rounds. But at each interconnection phase multiple separate Bellman-Fords will run from each cluster center remaining. Thus we need to argue that these runs of Bellman-Ford will not violate the memory (and IO memory) limit of each machine. This can be shown using Lemma 6, which states that for each vetex $v \in V$, w.h.p. the number of explorations of interconnection phase that visit v is at most $O(\deg_i \cdot \log n)$. In other words, each node only forwards messages to at most $O(\deg_i \cdot \log n)$ in each depth $\delta_i/2$ Bellman-Ford explorations performed for an interconnection phase. Moreover, the parameters of their construction is set so that $\deg_i = O(n^\rho)$ throughout the algorithm. Hence, each node $v \in V$ need to store and forward distance estimates corresponding to at most $O(n^\rho \log n)$ sources for $O(\log(\kappa\rho) + \frac{1}{\rho})$ iterations, and each Bellman-Ford runs for $O(\beta)$ rounds. These separate Bellman-Ford runs can be pipelined. Overall, all of the Bellman-Ford explorations can be implemented in $O(\frac{\beta}{\gamma} \cdot n^\rho \log n)$. ◀

We can now construct a hopset first and then run the distributed variant of the algorithm in Section 2.2 due to [24] for constructing the distance sketches on the new graph. The sketch of a given node v can be stored at a machine in $M(v)$.

Proof of Theorem 2. After constructing a (β, ϵ)-hopset (by setting $\kappa = k$), we store the edges added to each node v by redistributing them among machines $M(v)$ that simulate v. Let $G' = (V, E \cup H, w')$ be the graph obtained by adding hopset edges. For constructing distance sketches with stretch $2k-1$, we run the algorithm of [24] on G'. We run the restricted Bellman-Ford algorithm (Algorithm 1) in $O(\frac{\beta}{\gamma})$ rounds. Overall, $O(\frac{\beta n^\rho \log^2 n}{\rho\gamma})$ rounds are needed for the hopset construction (by Theorem 10), and $O(kn^{1/k}\log n \cdot \frac{\beta}{\gamma})$ rounds for building the distance sketches on G'. In case $k = O(1)$ we set $\rho = 1/\kappa$, and $\kappa = k$ to get $\beta = O(1)$ and total running time $\tilde{O}(n^{1/k})$. In case $k = \Theta(\log n)$, we will set $1/\kappa = \rho = \sqrt{\frac{\log\log n}{\log n}}$. ◀

3.1 Polylogarithmic Round Complexity

In this section we describe how we can modify our algorithm to run in a polylogarithmic number of rounds in exchange for increasing the stretch. We do this by first constructing a spanner, which sparsifies the graph ("shrinking" the input) and thus allows us to act as if we have "extra" total space. It turns out that this extra space is incredibly powerful, and will let us build distance sketches in polylogarithmic time. But in the end we have to pay for both the stretch of the spanner and the stretch of the sketch, so we only achieve stretch $O(k^2)$ rather than stretch $2k-1$ for sketches of size $\tilde{O}(n^{1/k})$.

There are intuitively two reasons why this extra space is so helpful. First, in MPC having extra space (or extra machines) is equivalent to having larger total communication bandwidth. This intuitively allows us to speed up the main construction algorithm by running the Bellman-Ford algorithms "in parallel". There are some technical details but it is not surprising that extra bandwidth is helpful.

The second reason why extra space is helpful is less obvious. Goodrich et al. [12] gave a powerful simulation argument, showing that PRAM algorithms can be efficiently simulated in MPC as long as the total number of processors used and the total space used by the PRAM algorithm are bounded by the size of the input. This is a very useful theorem, but the requirement that the number of processors is only the size of the input is very restrictive. For example, the state of the art PRAM algorithms for constructing hopsets use $\Omega(mn^{\rho})$ processors rather than $O(m)$ (for some value ρ determined by the parameters of the hopset). It turns out to be easy to extend [12] to show that if we have extra total space, we can use that extra space and communication to simulate PRAM algorithms that use slightly more processors or space. Thus by using a spanner first to sparsify the input, we give ourselves extra space and thus the ability to efficiently simulate a wider class of PRAM algorithms (hopsets in particular).

MPC with Extra Space. First we define a variant of MPC with extra machines (and thus extra space) denoted by MPC(S, S') where S is memory per machine, the number of machines is $\Theta(\frac{mS'}{S})$ and m is the total input size. This also implies the total memory available is $\Theta(mS')$ rather than $\Theta(m)$. We are first going to analyze our algorithm in this variant of MPC, and then switch back to the standard setting.

In [12] it was shown that with a small overhead PRAM algorithms can be simulated in MPC under certain assumptions on the number of processors and the memory used. We use a simple extension of their result for our new MPC variant.

▶ **Theorem 11.** *Given a PRAM algorithm using $\mathcal{P} = O(m\alpha)$ processors that runs in time $\mathcal{T}$, and uses $O(m\alpha)$ total memory at any time, this algorithm can be simulated in $O(\mathcal{T}/\gamma)$ rounds of MPC(m^{γ}, α), for any $0 < \gamma < 1$.*

This stronger variant of MPC also lets us extend Theorem 7 for larger message sizes. We define a generalized variant of Find Minimum that takes a collection of vectors and computes their coordinate-wise minimum, and a generalizes version of Broadcast which broadcasts a vector of messages (rather than just a single message). We get the following lemma.

▶ **Lemma 12.** *We can compute generalized Find Minimum(x, y) over N vectors of length α stored on a contiguous range of machines x to y in $O(1/\gamma)$ rounds of MPC(N^{γ}, α). Moreover, the generalized Broadcast$(\mathbf{b}, x, y)$ subroutine can also be implemented in $O(1/\gamma)$ rounds.*

Proof. In the new settings we have $\Theta(N^{1-\gamma} \cdot \alpha)$ machines that can be used for computation over N items in range (x, y), rather than $\Theta(N^{1-\gamma})$ machines used in Theorem 7. Therefore we can assign each coordinate to a group of $N^{1-\gamma}$ machines and then use a similar aggregation tree argument as in Theorem 7 on all the coordinates in parallel in $O(1/\gamma)$ rounds for both problems. ◀

Next, we describe how the algorithm of Theorem 2 can be modified to utilize the extra resources in MPC$(n, n^{1/k}\log n)$ to improve the round complexity. We use an argument similar to [24] with a few changes. The complete argument can be found in the full version.

▶ **Theorem 13.** *Given a graph $G = (V, E)$ with shortest path diameter Λ, there is an algorithm in MPC$(n^{\gamma}, n^{1/k}\log n)$ that runs in time $O(k\Lambda)$ w.h.p. and constructs Thorup-Zwick distance sketches of size $O(kn^{1/k}\log n)$ with stretch $2k - 1$.*

A straightforward extension of Theorem 13 implies that given a (β, ϵ)-hopset for a graph, we can compute distance sketches with stretch $(1+\epsilon)(2k-1)$ in $O(\frac{\beta}{\gamma})$ rounds of $\mathrm{MPC}(n^{\gamma}, n^{1/k}\log n)$. Next, we show that in addition to proving Theorem 13, the extra memory also lets us improve the number of rounds for the hopset construction. To show this, we use a result in [9] that constructs hopsets in PRAM, which is as follows:

▶ **Theorem 14** ([9]). *For any graph $G = (V, E, w)$ with n vertices, and parameters $2 \leq \kappa \leq (\log n)/4, 1/2 > \rho \geq 1/\kappa$ and $0 < \epsilon < 1$, there is a PRAM algorithm that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}} \log n)$ in $O(\frac{1}{\rho} \cdot \log^2 n \cdot \log \kappa \cdot \beta)$ PRAM time whp, where $\beta = O(\frac{\log n(\log \kappa + 1/\rho)}{\epsilon})^{\log \kappa + \frac{1}{\rho}}$ using $\tilde{O}((m + n^{1+1/\kappa})n^{\rho})$ processors.*

We now argue that by having more space/machines, we are can implement the algorithm in Theorem 14 with the same guarantees in low-memory MPC settings. We will not discuss the details of the PRAM construction but the intuition here is similar to Theorem 13. At a high level, having more communication/memory will allows us to perform all the $\tilde{O}(n^{\rho})$ Bellman-Ford explorations required in the algorithm of Theorem 14 in parallel. By setting $\rho = 1/\kappa$ in Theorem 14 and then applying the simulation in Theorem 11 we get,

▶ **Corollary 15.** *For any graph $G = (V, E, w)$, and parameters $0 < \epsilon < 1, 1/\kappa < \gamma \leq 1, \kappa \geq 2$, there is an algorithm that computes a (β, ϵ)-hopset with size $O(n^{1+\frac{1}{\kappa}} \log n)$ w.h.p. in $O((\kappa/\gamma) \cdot \log^2 n \cdot \log \kappa \cdot \beta)$ rounds of MPC($n^{\gamma}, n^{1/\kappa}$), where $\beta = O(\frac{\log n(\log \kappa)}{\epsilon})^{\log \kappa + \kappa + 1}$.*

Obtaining Extra Space. Our modified algorithm for $\mathrm{MPC}(n^{\gamma})$ now proceeds as follows: we first construct a spanner, then construct a hopset on this spanner, and then use Theorem 13. Intuitively, by sparsifying the graph we can "buy" more memory and hence more communication. In other words, by building a spanner we can extend the results of the extra memory setting to the standard MPC setting.

There are several efficient PRAM algorithms for constructing spanners that we can simulate in MPC. We use an algorithm proposed by [2] that constructs a $(2k-1)$-spanner of size $O(kn^{1+1/k} \log n)$ with high probability. We then use Theorem 11 with $\alpha = 1$ (i.e. the original simulation of [12]) to construct the spanner in $O(\frac{k}{\gamma} \log n \log^* n)$ rounds of $\mathrm{MPC}(n^{\gamma})$, and then redistribute the spanner edges (e.g., by sorting), to make the input distribution uniform over all the machines. We can now put everything together to get the polylogarithmic construction.

Proof of Theorem 3. We first construct a $4k-1$-spanner with size $O(kn^{1+\frac{1}{2k}})$. We denote this spanner by G'. Since G' has size $m' = O(n^{1+\frac{1}{2k}})$, while our total memory (and consequently overall communication bound) is still based on the original graph. Equivalently, the number of machines is $\frac{m}{n^{\gamma}} = \Omega(\frac{m' n^{1/2k} \log n}{n^{\gamma}})$ (since $m = \Omega(kn^{1+1/k} \log n)$), and therefore we are exactly in the $\mathrm{MPC}(n^{\gamma}, n^{\frac{1}{2k}})$ setting, but where the input graph is G'. Then we use Corollary 15 to construct a (β, ϵ)-hopset for G' with $\beta = O(\frac{k}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + 1 + k})$ rounds of $\mathrm{MPC}(n^{\gamma})$. Finally, after adding the hopset edges to G' we use Theorem 13. The new stretch is clearly $O(k^2(1+\epsilon))$. ◀

3.2 Single-source shortest path

In various models (such as PRAM, CONGEST and Congested Clique) hopsets are used for solving shortest path problems (e.g. [8, 14, 9]), and thus it is natural to see how they can be used for this application in the MPC model. In particular, we discuss application of Theorem 10 in solving the (approximate) single-source shortest path problem. As stated earlier, while this problem is well-studied in many distributed models, including the Congested Clique model, we are not aware of any non-trivial results for this problem in the low memory MPC setting.

▶ **Theorem 16.** *Given a weighted undirected graph $G = (V, E, w)$, a source node $s \in V$, and $0 < \gamma \leq 1, 0 < \epsilon < 1$ we can compute $(1 + \epsilon)$-approximate distances from s to all nodes in V w.h.p. in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC with $\Theta(n^\gamma)$ memory per machine.*

Proof. We first construct a hopset using Theorem 10 by setting $\rho = \sqrt{\frac{\log n}{\log \log n}}$, and $\kappa = \Theta(\log n)$. This will let us build a hopset with hopbound $2^{\tilde{O}(\log n)}$ in time $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\log n)}$. We then run the restricted Bellman-Ford algorithm (Algorithm 1) in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC(n^γ). The idea behind this choice of parameters is the following: any attempt to improve the running time by getting a smaller hopbound (e.g. constant) will increase the time required to construct the hopset. In other words, this choice of parameters will make the time required for preprocessing (construction of the hopset) almost the same as the time required for running the Bellman-Ford algorithm. ◀

Finally, we show that we can used the technique in Section 3.1 to find constant approximation to single source shortest path in polylogarithmic time for graphs with a certain density. In particular, by first constructing a spanner and then using Corollary 15, we can also solve $4k(1 + \epsilon)$-approximate SSSP (for any $2 \leq k \leq O(\log n)$) on any graph with $m = \Omega(n^{1+1/k} \log n)$ edges in fewer number of rounds. After constructing a $4k - 1$-spanner, we construct a (β, ϵ)-hopset for an appropriate hopbound β using the extra space and then run a single restricted Bellman-Ford (Algorithm 1) from the source in $O(\beta/\gamma)$ rounds of MPC(n^γ). By setting $\kappa = k$ we get,

▶ **Corollary 17.** *For any graph $G = (V, E, w)$ with n vertices, $m = \Omega(n^{1+1/k})$ edges, and $0 < \epsilon < 1, 1/k < \gamma \leq 1, k > 2$, and a source node $s \in V$, there is an algorithm that w.h.p. finds a $4k(1 + \epsilon)$-approximation of shortest path distance from s to all nodes in $O(\frac{1}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k + 1})$ rounds of MPC(n^γ). In particular, for $k = O(1)$ the algorithm runs in $O(\frac{1}{\gamma} \cdot (\frac{\log n}{\epsilon})^{O(1)})$ rounds.*

References

1 Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2018.

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

3 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

4 Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

5 Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief Announcement: Semi-MapReduce Meets Congested Clique. *arXiv preprint arXiv:1802.10297*, 2018.

6 Keren Censor-Hillel, Michal Dory, Janne H Korhonen, and Dean Leitersdorf. Fast Approximate Shortest Paths in the Congested Clique. In *Proceedings of Symposium on Principles of Distributed Computing*. ACM, 2019.

7 Shiri Chechik. Approximate distance oracles with constant query time. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663. ACM, 2014.

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

9 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 128–137. IEEE, 2016.

10 Michael Elkin and Ofer Neiman. On efficient distributed construction of near optimal routing schemes. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 235–244. ACM, 2016.

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

12 Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pages 374–383. Springer, 2011.

13 MohammadTaghi Hajiaghayi, Silvio Lattanzi, Saeed Seddighin, and Cliff Stein. MapReduce Meets Fine-Grained Complexity: MapReduce Algorithms for APSP, Matrix Multiplication, 3-SUM, and Beyond. *arXiv preprint arXiv:1905.01748*, 2019.

14 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 489–498. ACM, 2016.

15 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

16 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 391–410. Society for Industrial and Applied Mathematics, 2015.

17 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 381–390. ACM, 2013.

18 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in O (log log n) communication rounds. *SIAM Journal on Computing*, 35(1):120–131, 2005.

19 Manor Mendel and Assaf Naor. Ramsey partitions and proximity data structures. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 109–118. IEEE, 2006.

20 Mihai Patrascu and Liam Roditty. Distance Oracles beyond the Thorup–Zwick Bound. *SIAM Journal on Computing*, 43(1):300–311, 2014.

21 Mihai Patrascu, Liam Roditty, and Mikkel Thorup. A new infinity of distance oracles for sparse graphs. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 738–747. IEEE, 2012.

22 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

23 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *Journal of the ACM (JACM)*, 65(6):41, 2018.

24 Atish Das Sarma, Michael Dinitz, and Gopal Pandurangan. Efficient distributed computation of distance sketches in networks. *Distributed Computing*, 28(5):309–320, 2015.

25 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.

26 Christian Wulff-Nilsen. Approximate distance oracles with improved query time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 539–549. SIAM, 2013.