Geometric Spanners in the MapReduce Model

Sepideh Aghamolaei[1(✉)], Fatemeh Baharifard[2], and Mohammad Ghodsi[1,2]

[1] Department of Computer Engineering,
Sharif University of Technology, Tehran, Iran
aghamolaei@ce.sharif.edu, ghodsi@sharif.edu
[2] School of Computer Science, Institute for Research in Fundamental Sciences (IPM),
Tehran, Iran
f.baharifard@ipm.ir

Abstract. A *geometric spanner* on a point set is a sparse graph that approximates the Euclidean distances between all pairs of points in the point set. Here, we intend to construct a geometric spanner for a massive point set, using a distributed algorithm on parallel machines. In particular, we use the MapReduce model of computation to construct spanners in several rounds with inter-communications in between. An algorithm in this model is called efficient if it uses a sublinear number of machines and runs in a polylogarithmic number of rounds. In this paper, we propose an efficient MapReduce algorithm for constructing a geometric spanner in a constant number of rounds, using linear amount of communication. The stretch factors of our spanner is $1 + \epsilon$, for any $\epsilon > 0$.

Keywords: Computational geometry · Parallel computation
Geometric spanners · MapReduce

1 Introduction

Space limitations are the main challenge in processing massive data i.e. data that do not fit inside the memory of a single machine. Given a bounded memory, an efficient algorithm has a low time complexity. Some space-bounded models allow a type of secondary slower memory or communication between multiple fast memories to reduce the running time of the algorithm. Allowing two types of memory, shifts the challenge in the algorithm design to data communication.

MapReduce is a framework for processing data in large scales in which a set of machines, each have a part of the input, run an algorithm in simultaneous rounds and after each round, they can communicate their data to each other. Efficient MapReduce algorithms have sublinear machines each with sublinear memory that run for polylogarithmic number of rounds and the number of machines used in the algorithm must be asymptotically as many as the input size.

An example of problems that has been discussed in MapReduce framework is Euclidean minimum spanning tree problem, which was studied by Andoni

L. Wang and D. Zhu (Eds.): COCOON 2018, LNCS 10976, pp. 675–687, 2018.
https://doi.org/10.1007/978-3-319-94776-1_56

et al. [3] who presented an algorithm with $O(1)$ round complexity and superlinear memory. Later, Yaroslavtsev and Vadapalli [23] proved lower bounds for this problem and proposed an algorithm for approximating each edge of the minimum spanning tree. Another example is computing the core-set for convex hull using the method proposed in [2], which also works in MapReduce model. Moreover, fixed-dimensional linear programming, 1-dimensional all nearest neighbors, 2-dimensional and 3-dimensional convex hull algorithms were solved in memory-bound MapReduce model [13] and practically proven algorithms for sky-line computation, merging two polygons, diameter and closest pair problems have been discussed in MapReduce model [8,10].

A network is called a *t-spanner*, if there is a short path between any pairs of nodes, within a guaranteed ratio t to the shortest paths between those nodes in an underlying base graph. Most of the time and in this paper, the complete graph which has $\theta(n^2)$ edges is considered as the underlying base graph.

Most efficient spanner construction algorithms are geometric and they find practical applications in areas such as terrain construction [12,21], metric space searching [19], broadcasting in communication networks [11] and solving approximately geometric problems like traveling salesman problem [20].

Recently, a divide and conquer algorithm for constructing geometric spanners has been studied as spanners merging problem in [4]. The size of the spanner created using this method is $O(n \log n)$, which requires $O(\log n)$ times more memory than the input. However, the proposed algorithm uses linear memory in its final merging steps, which is infeasible for MapReduce model. Moreover, in [6,14] the problems of well-separated pair decomposition on PRAM and 3D covex hull in MapReduce model were considered. Using the lifting transformation, 3D convex hull solves planar Delaunay triangulation. Using simulation of PRAM algorithms in MapReduce as discussed in [14] gives algorithms for the two spanners of [4,6]. Direct algorithms for Delaunay triangulation in MapReduce [5,18] are randomized and require a super-linear number of machines unlike the PRAM simulation. A summary of results on geometric spanners in MapReduce is shown in Table 1.

Contributions. Table 1 compares the previous results for geometric spanners with the one given in this paper.

Table 1. A summary of results on geometric spanners in MapReduce. $\frac{1}{\delta} = \log_m^n$, where m is the memory of each machine.

Spanner	$\lvert E\rvert$	Stretch factor	Rounds	Communication	Reference
WSPD	$O(n)$	$1+\epsilon$	$O(\frac{\log n}{\delta})$	$O(\frac{n}{\delta \log n})$	Simulation [14], PRAM [6]
DT	$O(n)$	1.998	$O(\frac{1}{\delta})$	$O(\frac{n}{\delta})$	3D convex hull [14,22]
-	$O(\frac{n}{\epsilon})$	$1+\epsilon$	$O(\frac{1}{\delta})$	$O(\frac{n}{\delta})$	this paper (Algorithm 4)

In this paper, we propose efficient algorithms for constructing a geometric spanner similar to Yao-graph and a special case of dynamic programming in the MapReduce model. Our algorithms run for a constant number of rounds and use linear memory.

2 Preliminaries

In this section we review basic knowledge required to understand the rest of the paper.

2.1 MapReduce Model

Different theoretical models for MapReduce has been introduced over the years [9,14,16]. In MapReduce class (MRC) model, for an input of size n, the following three conditions must be satisfied:

- the number of machines is sublinear: $L = o(n)$
- the memory of each machine is sublinear: $m = o(n)$
- the number of rounds is polylogarithmic: $O(polylog(n))$.

In MRC model, the input of each round is distributed among machines. Let S_i be the part of the input assigned to machine i in each round. Data in MapReduce are stored as (key, value) pairs. A MapReduce algorithm consists of three steps: *map*, *shuffle* and *reduce*:

- *map*: processes data into a set of (key, value) pairs.
- *shuffle*: sends data with the same key to the same machine.
- *reduce*: aggregates data with the same key.

Operations *map* and *reduce* are local, while *shuffle* distributes data between machines.

Two main parallel algorithms operations are semi-group and prefix sum:

- Semi-group: $x_1 \oplus x_2 \oplus \cdots \oplus x_n$, i.e. for a set S and a binary operation $\oplus : S \times S \to S$, the associative property holds: $\forall a, b, c \in S, (a \oplus b) \oplus c = a \oplus (b \oplus c)$.
- Prefix sum: $x_1 \oplus x_2 \oplus \cdots \oplus x_i, i = 1, \ldots, n$
- Diminished prefix sum: $x_1 \oplus x_2 \oplus \cdots \oplus x_{i-1}, i = 1, \ldots, n$

Both of these operations also take $O(\log_m^n)$ rounds and $O(n \log_m^n)$ computation in MapReduce [14]. Parallel algorithms in CRCW PRAM model can be simulated in MapReduce model by a factor 2 slow-down [14]. A class of functions that can be computed with minimum round and communication complexity are known as MRC-parallelizable functions [16]. An example of a MRC-parallelizable functions is computing the frequency of words in a set of documents which is known as word count algorithm [7].

Special cases of dynamic programming have been discuessed in MapReduce [15] with $(1+\epsilon)$-approximation factor, $O(1)$ rounds and $\tilde{O}(n)$ communication, where $\tilde{O}$ ommits a $polylog(n)$ factor. The required conditions for this type

of dynamic programming are monotonicity and decomposability. Monotonicity states that the cost of subproblems is less than the main problem, and decomposability states that the input can be decomposed into two-level laminar family of partial inputs where group is the higher level and block is the lower level, such that the solution to the main problem is a concatenation of the solutions to the subproblems and a nearly optimal solution can be constructed from $O(1)$ blocks.

Algorithms for constructing range searching data structures in MapReduce exist [1]. However, they lack the efficiency required for simultaneous queries.

2.2 Geometric Spanners

A geometric network G is a *t-spanner* for a point set P, if a $t > 1$ exists such that for each pair of points u and v in P, there is a path in G between u and v, whose length is less than or equal to the t times of the Euclidean distance between u and v. The minimum t such that G is a t-spanner of P is the *spanning ratio* of G.

Many spanner algorithms exist which excel in different quality measures. Here, we describe an algorithm for constructing a t-spanner of a set of points in Euclidean space which constructs a spanner with a linear number of edges in $O(n \log n)$ time.

Yao-Graph. One of the most common spanners is Yao-graph which is denoted by Y_k-graph. The Y_k-graph is constructed as follows. Given a set P of points in the plane, for each vertex $p \in P$, partition the plane into k disjoint cones (regions in the plane between two rays originating from the same point) with apex p, each defined by two rays at consecutive multiples of $\theta = \frac{2\pi}{k}$ radians from the negative y-axis and label the cones $C_0(p)$ through $C_{k-1}(p)$, in counterclockwise order around p. Then for each cone with apex p, connect p to its closest vertex q inside that cone.

It is proven that for any θ with $0 < \theta < \pi/3$, the Yao-graph with cones of angle θ, is a t-spanner of P for $t = \frac{1}{1-2\sin(\theta/2)}$ with $O(\frac{n}{\theta})$ edges, that can be constructed in $O(\frac{n \log n}{\theta})$ time [17].

3 Mergeable Dynamic Programming

In Algorithm 1 we solve the special dynamic programming, which is defined below, in MapReduce model. Actually, the idea behind dynamic programming in MapReduce is similar to the parallel prefix sum in PRAM.

Definition 1 (Mergeable DP). *A dynamic program (DP) with input set S and output set Q with a recurrence relation $f : S^k \to Q$ and a table T with a valid filling order $\Phi : T \to \mathbb{N}$ and size $|T| = n^d$, and mappings between $S \leftrightarrow T$ and $Q \leftrightarrow T$, is a mergeable DP if the following three conditions hold:*

- *Sparsity: The number of cells of T required for computing Q is $O(|S|)$.*

- *Neighbors: Computing f on each block requires data only from $O(1)$ previous blocks (in the order of Φ).*
- *Order Preserving: The value of each cell must only depend on the cells with smaller or equal index based on Φ and the order of each dimension of the table T.*
- *Parallelizable: Function f must be a semi-group function.*
- *Summarizable: There is an integer ℓ sublinear in $|S|$ ($\ell = o(|S|)$), such that using the last ℓ values of T in the order of Φ, it is possible to compute the rest of the table, i.e. function h exists such that $T[\Phi^{-1}(k)] = h(T[\Phi^{-1}(k-1)], \ldots, T[\Phi^{-1}(k-\ell)], S[\Phi^{-1}(k)])$.*

The sparsity and summarizability of the DP table allow us to summarize the computed part of the table into a sublinear subset of cells, which we call a frontier. A formal definition of frontier is given in Definition 2.

Definition 2 (Frontier). *The frontier is a subset of cells along with their indices F (denoted by $F.cells$ and $F.indices$) of a DP table T, such that cells with indices $R = \{E(F)+1, \ldots, |T|\}$ can be computed using cells with indices $F.indices \cup R$, where $E(F) = \max_{f \in F.indices} f$.*

Definition 3 (Frontier Merging). *Given two frontiers a, b, frontier merging operation creates a frontier c which is a frontier for cells from 1 to $k = \max\{E(a), E(b)\}$. Build a hypotetical table T_X similar to T but only store data from set $X = a.cells \cup b.cells$. Fill the table T_X from cell 1 to cell k. Using the summarizabilty property of the mergeable DP with table T_X and ordering Φ, there is a sequence $T_X[\Phi^{-1}(k)], \ldots, T_X[\Phi^{-1}(k-\ell-1)]$ for each cell $T_X[\Phi^{-1}(k)]$. Report this as c.*

In Lemma 1, we show that a frontier of a mergeable DP can be constructed using a (diminished) parallel prefix algorithm.

Lemma 1. *The operation of Definition 3 builds a frontier for $\cup_{i=k}^{k-\ell} T[\Phi_{-1}(i)]$ and it is a semi-group function.*

Proof. Since T is a mergeable DP, T_X is also a mergeable DP. For two sets A and B, using the semi-group property of f, computing T_A and T_B and applying f on their results in the order of Φ, gives $T_{A \cup B}$. For $A = \{T[1], \ldots, T[E(a)]\}$ and $B = \{T[1], \ldots, T[E(b)]\}$, the result is $A \cup B = \{T[1], \ldots, T[k]\}, k = \max(E(a), E(b)) = E(c)$.

Assume three frontiers a, b and c of a DP table in the order of Φ with the set of indices denoted by S_a, S_b and S_c, respectively. Now, we prove the two possible orders of computation result in the same result. Since the frontiers in the computation follow the order of Φ:

$$a \oplus b = T_{S_a \cup S_b}[E(b)] \Rightarrow (a \oplus b) \oplus c = T_{S_a \cup S_b \cup S_c}[E(c)]$$

The other case can be proven similarly:

$$b \oplus c = T_{S_b \cup S_c}[E(c)] \Rightarrow a \oplus (b \oplus c) = T_{S_a \cup S_b \cup S_c}[E(c)]$$

which proves the lemma. □

Now, we give an algorithm (Algorithm 1) for solving the mergeable DP (Definition 1) problems in MapReduce.

Algorithm 1. Mergeable DP in MapReduce

Input: A mergeable DP (S, Q, f, T, Φ)
Output: The output of DP
1: Compute a valid ordering Φ on S and distribute the points by Algorithm 2.
2: Map each cell $S[\Phi^{-1}(i)]$ to a ℓ-tuple $(\emptyset, \ldots, \emptyset, S[\Phi^{-1}(i)])$.
3: Run the diminished parallel prefix algorithm with the frontier merging as the operation on ℓ-tuples.
4: Compute T by applying f on ℓ-tuples and S and store the result.
5: Use f to update the local values of T to compute Q.
6: **return** Q

Theorem 1. *Algorithm 1 solves mergeable dynamic programming problems correctly.*

Proof. By Definition 1, for a mergeable dynamic programming, the order (Φ) in which the table is filled can be determined. Using this order, Algorithm 2 finds partitions that are ordered based on the order of the dynamic program (Theorem 4). Then, a parallel prefix computation can be used for computing the frontier in each cell (Lemma 1) and the minimum of the related cells gives the value of the cell in T. □

Theorem 2. *Algorithm 1 takes $O(\log_m^n)$ rounds and it has $O(n \log_m^n)$ communication complexity in MRC model, if $\ell = O(\sqrt[d+1]{n})$ for a d-dimensional DP.*

Proof. The algorithm consists of a parallel prefix computation and running Algorithm 2 once. The round and communication complexity of Algorithm 2 is $O(\log_m^n)$ and $O(n \log_m^n)$ respectively. The round complexity of parallel prefix algorithm is $O(\log_m^n)$ and the communication complexity of the algorithm is $O(\ell.\ell^d \log_m^n)$, since instead of data with $O(1)$ dimension, vectors of length $O(\ell)$ have been used, and there are $O(\ell^d)$ cells.

Using Theorem 4, the number of points in each row and column is $O(m)$, so the overall communication and space for sending data from a row and a column is also $O(m)$. Based on Definition 1, the amount of data from other cells is $O(m)$. Therefore, all the steps of the algorithm can be run using $O(n)$ communication per round. □

4 Application: Geometric Spanners in MapReduce

In this section, we present an efficient algorithm for constructing a geometric spanner in the MapReduce model.

4.1 A Balanced Grid in MapReduce

We can store a grid by its separating lines. A partitioning based on a regular grid and an indexing scheme is used to distribute data among machines. Algorithm 2 takes as input a set of point-sets and an ordering scheme and builds a regular grid and a partitioning of that grid based on the given ordering.

In Algorithm 2, $\mathbf{Sort_f(S)}$ means sorting based on function f of points in S in MapReduce model. Sorting points means they are distributed among L machines such that $i < j \Rightarrow \forall a \in S_i, b \in S_j, f(a) \leq f(b)$. Sorting n points in this model can be done in $O(\log_m^n)$ rounds [14], which is $O(\frac{1}{\delta})$ for $m = n^\delta$ where δ is a constant $(0 \leq \delta \leq 1)$.

Algorithm 2. A Regular Grid in MapReduce

Input: a set of points set $S = \{S_1, \cdots, S_L\}$, an ordering function $f(.,.)$
Output: a space partitioning of S, the grid lines
1: $\text{Sort}_x(S)$
2: $X_i \leftarrow \min_{(x,y)\in\cup_i S_i} x$
3: send x_i to all other machines $(X = \cup_i X_i)$
4: $\text{Sort}_y(S)$
5: $Y_i \leftarrow \min_{(x,y)\in\cup_i S_i} y$
6: send Y_i to all other machines $(Y = \cup_i Y_i)$
7: locally compute the index of each cell (i, j) using the ordering function $f(i, j)$
8: $\text{Sort}_{f(x,y)}(S)$
9: Re-index the sets in the order of $\min_{(x,y)\in S_i} f(x, y)$
10: **return** $S = \{S_1, \cdots, S_L\}$, X, Y

Here, we present some properties of Algorithm 2, which are used later.

Lemma 2. *The number of points in each cell of the grid in Algorithm 2 is at most m.*

Proof. Based on the sorting on x, the number of points between X_i and X_{i+1} is $O(m)$. Similarly, the number of points between Y_i and Y_{i+1} is $O(m)$. So, in the grid built on $X \times Y$, the number of points in each cell is $O(m)$. By indexing the points based on $f(x, y)$, the partitions lie inside cells of $X \times Y$. Since all equal keys in a MapReduce computation go to the same machine, there are no half-cells.

Theorem 3. *The round complexity of Algorithm 2 is $O(\log_m^n)$ and its communication complexity is $O(n \log_m^n)$.*

Proof. Each sorting takes $O(\log_m^n)$ rounds, which is constant for $m = O(n^\delta)$ and $O(\log_m^n)$ communication. Since the round and communication complexities of the algorithm are the sum of the complexities of these sorting steps, they are $O(\log_m^n)$ and $O(n \log_m^n)$ respectively. □

Theorem 4. *There are at most $O(L)$ partitions in the output of Algorithm 2, each with $O(m)$ points.*

Proof. The last sorting step of the algorithm which sorts points based on the value of f, divides points into sets $S_1, \ldots, S_L$ such that $\forall (x, y) \in S_i, (x', y') \in S_j, i < j \Rightarrow f(x, y) \leq f(x', y')$. The size of the sets created using a sorting algorithm is $O(m)$ (Lemma 2) and the number of sets is $O(L)$, so the number of partitions is $O(L)$ and each of them has $O(m)$ points. □

4.2 Simultaneous 2-Sided Queries

Now we present an offline algorithm for solving the 2-sided range queries in Algorithm 3, which can be extended to MapReduce in the same way as parallel prefix computation.

In Algorithm 3, the closest point to (X_i, Y_j) using distance function ℓ_1 that lies inside the 2-sided range $(x \leq X_i, y \leq Y_j)$ is computed. Also, $\|x\|_1$ denotes the length of vector x under ℓ_1.

Algorithm 3. Simultaneous 2-Sided Queries

Input: A point set S, a rectangular grid $\{X_i\}_{i=1}^{\ell} \times \{Y_j\}_{i=1}^{\ell}$
Output: The nearest neighbor to (X_i, Y_j) using points of S inside the 2-sided range

1: Run Algorithm 2 to build a $\ell \times \ell$ table T and index it using $\Phi(x, y) = x + y \times \ell$.
2: Run Algorithm 1 with S as point set, T as table, $T[i, j] = \arg \min_{t \in S[i,j] \cup T[i-1,j] \cup T[i,j-1]} \|t - (X_i, Y_j)\|_1$ as f and Φ as the ordering.
3: **return** T

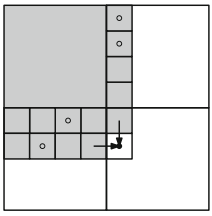


Fig. 1. For a 2D DP table which is filled in 2 directions (down and right), the data of the same row and column cells are needed in addition to C_i of the previous cells.

Lemma 3. *Algorithm 3 is a mergeable dynamic programming.*

Proof. The algorithm is mergeable since it satisfies the conditions of mergeable dynamic programming as defined in Definition 1:

– Sparsity: The number of cells required to answer 2-sided queries is $O(n)$, since we only need to know the value of cells which contain a point.

– Neighbors: The value of a cell (i, j) can be determined using the values of cells $T[i-1, j], T[i, j-1], S[i, j]$.
– Parallelizable: Minimum distance to the grid point corresponding to the corner of cell $[i, j]$ is a semi-group function, since minimum computations are semi-group.
– Summarizable: The anti-diagonal of T that passes through each cell is the frontier of that cell under 2-sided queries. So this problem is summarizable for $\ell = n$. □

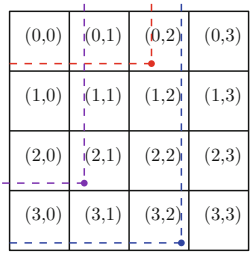


Fig. 2. Three 2-sided range queries for computing 3 nearest neighbors.

Lemma 4. *The nearest neighbor of a point inside a 2-sided range using ℓ_1 distance can be computed using Algorithm 3.*

Proof. Each 2-sided range query on the grid (Fig. 2), can be computed using Algorithm 1 for computing the recurrence relation. The nearest point to (X_i, Y_j) using ℓ_1 distance is either in cell $S[i, j]$ or in one of its neighbors: $T[i-1, j]$, $T[i, j-1]$ (Fig. 1). Since the algorithm checks all these values, it finds the exact nearest neighbor. □

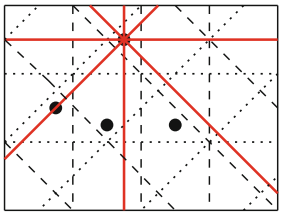
4.3 A Geometric Spanner in MapReduce

To build a spanner similar to Yao-graph, we first solve simultaneous 2-sided range queries, using dynamic programming. These queries are then used in the spanner algorithm to find an approximate nearest neighbor in each cone.

Our algorithm for constructing a spanner (Algorithm 4) creates a grid and applies nearest neighbor search to find the edges. Algorithm 4 creates a set of oriented rhomboid grids with lines parallel to the ones creating cones around each point, as shown in Fig. 3.

In Algorithm 4, the distances computed in Algorithm 3 are ℓ_1 distances of the affine transformations of a grid, as shown in Fig. 4. Lemma 5 computes the approximation factor between this distance and the Euclidean distance.

Lemma 5. *The distance between two points on a grid with unit vectors that have an angle θ $(0 \leq \theta \leq \frac{\pi}{2})$ between them, is $1 + O(\theta)$ times the Euclidean distance between those points.*



$k = 8$

Fig. 3. The overlay of two out of $k = 8$ square grids of Algorithm 4, indicated by dotted lines and dashed lines. The bold red lines denote the cones around a point. (Color figure online)

Algorithm 4. A Geometric Spanner in MapReduce

Input: A set of points set $S = \{S_1, \ldots, S_L\}$, an integer $k \geq 7$
Output: A spanner with k cones around each point
1: $\Theta = \{\frac{2\pi i}{k} | i = 1, \ldots, k\}$
2: locally create pairs of consecutive directions from Θ in clockwise order.
3: locally build a grid for each pair of directions from previous step.
4: repeat each point p once in each grid.
5: run Algorithm 3 with $d(.,.)$ defined as the ℓ_1 distance and $\cup_i S_i$ as the point set, using Algorithm 1 in each grid to find the nearest neighbor of each point inside its cone.
6: add an edge between each point and one of its nearest neighbors in each direction (cone).
7: **return** the edges of the spanner.

Proof. Assume w.l.o.g. that one of the points is $(0, 0)$ and the other one is $p = (x, y), x, y > 0$. The angle between $\overrightarrow{op}$ and $\overrightarrow{i}$ is $\alpha \leq \theta$, since p lies inside the cone. Using basic trigonometry, the distance computed in our algorithm is $x + y$ and the Euclidean distance between these points is $\frac{y \sin(\theta)}{\sin(\alpha)}$. Also, $x = \cos(\alpha)\frac{y \sin(\theta)}{\sin(\alpha)} - y\cos(\theta)$. So, using Taylor series for cosine and Maclaurin series for $\frac{1}{1-X}$, the approximation factor is proved:

$$\frac{x+y}{|\overrightarrow{op}|} = \frac{\cos(\alpha)\frac{y\sin(\theta)}{\sin(\alpha)} - y\cos(\theta) + y}{\frac{y\sin(\theta)}{\sin(\alpha)}}$$

$$= \frac{y\cos(\alpha)\sin(\alpha) - y\cos(\theta)\sin(\alpha) + y\sin(\alpha)}{y\sin(\theta)} = \frac{\sin(\theta-\alpha)+\sin(\alpha)}{\sin(\theta)}$$

$$= \frac{2\sin(\frac{\theta}{2})\cos(\frac{\theta-2\alpha}{2})}{2\sin(\frac{\theta}{2})\cos(\frac{\theta}{2})} = \frac{\cos(\frac{\theta-2\alpha}{2})}{\cos(\frac{\theta}{2})} \leq \frac{1}{1-\frac{\theta^2}{8}} = 1 + \frac{\theta^2}{8} + o(\theta^2).$$

□

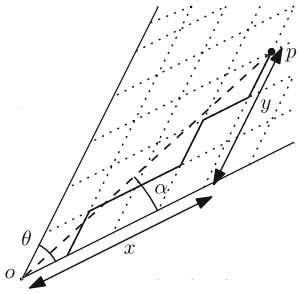


Fig. 4. A grid built inside a cone and the path on the grid compared to the distance between the point inside the cone and the apex.

Theorem 5. *The stretch factor of the spanner of Algorithm 4 is* $1 + O(\theta)$.

Proof. Applying Lemma 5 proves using ℓ_1 distance instead of the Euclidean distance (Algorithm 4) adds a factor $1 + O(\theta^2)$, and Lemma 4 proves Algorithm 3 computes the exact ℓ_1 distance. Using induction on the length of the path, similar to the proof of the stretch factor of Yao-graph, proves the approximation factor. □

Theorem 6. *Algorithm 4 has* $O(k \log_m^n)$ *round complexity and* $O(nk \log_m^n)$ *communication complexity.*

Proof. The algorithm solves one instance of range query per cone to compute the nearest neighbors simultaneously. Based on Lemma 3, Algorithm 3 is a mergeable dynamic program, and using Theorem 1, it takes $O(\log_m^n)$ rounds and $O(n \log_m^n)$ communications to solve it. Since there are k cones, the overall complexity of the algorithm is $O(k \log_m^n)$ rounds and $O(kn \log_m^n)$ communication. □

5 Conclusion

We introduced a $(1 + \epsilon)$-spanner in Euclidean plane and presented a MRC algorithm for constructing it in optimal round and communication complexities. However, the number of machines used in our algorithm is sub-quadratic. Finding algorithms that use fewer machines and algorithms for spanners with other geometric properties such as bounded degree spanners and bounded diameter spanners are also important.

We also proved conditions for parallelizable dynamic programming problems. Solving other problems in MapReduce using our method might also be interesting. Also, finding algorithms for other simultaneous queries can reduce the complexity of some MapReduce algorithms.

References

1. Agarwal, P.K., Fox, K., Munagala, K., Nath, A.: Parallel algorithms for constructing range and nearest-neighbor searching data structures. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 429–440. ACM (2016)
2. Agarwal, P.K., Har-Peled, S., Varadarajan, K.R.: Geometric approximation via coresets. Comb. Comput. Geom. **52**, 1–30 (2005)
3. Andoni, A., Nikolov, A., Onak, K., Yaroslavtsev, G.: Parallel algorithms for geometric graph problems. In: Proceedings of the 46th Annual ACM Symposium on Theory of Computing, pp. 574–583 (2014)
4. Bakhshesh, D., Farshi, M.: Geometric spanners merging and its applications. In: Proceedings of the 28th Canadian Conference on Computational Geometry, pp. 133–139 (2016)
5. Birn, M., Osipov, V., Sanders, P., Schulz, C., Sitchinava, N.: Efficient parallel and external matching. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 659–670. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_66
6. Callahan, P.B.: Dealing with higher dimensions: the well-separated pair decomposition and its applications. Ph.D. thesis, Johns Hopkins University (1995)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
8. Eldawy, A., Li, Y., Mokbel, M.F., Janardan, R.: CG_Hadoop: computational geometry in MapReduce. In: Proceedings 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 294–303 (2013)
9. Eldawy, A., Mokbel, M.F.: Communication steps for parallel query processing. In: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 273–284 (2013)
10. Eldawy, A., Mokbel, M.F.: SpatialHadoop: A MapReduce framework for spatial data. In: Proceedings of the 31st International Conference on Data Engineering, pp. 1352–1363 (2015)
11. Farley, A.M., Proskurowski, A., Zappala, D., Windisch, K.: Spanners and message distribution in networks. Discrete Appl. Math. **137**, 159–171 (2004)
12. Ghodsi, M., Sack, J.: A coarse grained solution to parallel terrain simplification. In: Proceedings of 10th Canadian Conference on Computational Geometry (1998)
13. Goodrich, M.T.: Simulating parallel algorithms in the MapReduce framework with applications to parallel computational geometry. arXiv preprint arXiv:1004.4708 (2010)
14. Goodrich, M.T., Sitchinava, N., Zhang, Q.: Sorting, searching, and simulation in the MapReduce framework. In: Proceedings of the 22nd Annual International Symposium on Algorithms and Computation, pp. 374–383 (2011)
15. Im, S., Moseley, B., Sun, X.: Efficient massively parallel methods for dynamic programming. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, pp. 798–811. ACM (2017)
16. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms, pp. 938–948 (2010)
17. Narasimhan, G., Smid, M.: Geometric Spanner Networks. Cambridge University Press, Cambridge (2007)

18. Nath, A., Fox, K., Munagala, K., Agarwal, P.K.: Massively parallel algorithms for computing TIN DEMs and contour trees for large terrains. In: Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (2016)
19. Navarro, G., Paredes, R., Chávez, E.: t-spanners as a data structure for metric space searching. In: Laender, A.H.F., Oliveira, A.L. (eds.) SPIRE 2002. LNCS, vol. 2476, pp. 298–309. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45735-6_26
20. Rao, S.B., Smith, W.D.: Approximating geometrical graphs via "spanners" and "banyans". In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing, pp. 540–550 (1998)
21. van Kreveld, M.: Algorithms for triangulated terrains. In: Plášil, F., Jeffery, K.G. (eds.) SOFSEM 1997. LNCS, vol. 1338, pp. 19–36. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63774-5_95
22. Xia, G.: The stretch factor of the Delaunay triangulation is less than 1.998. SIAM J. Comput. **42**, 1620–1659 (2013)
23. Yaroslavtsev, G., Vadapalli, A.: Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances. arXiv preprint arXiv:1710.01431 (2017)