GeoGraph: A Framework for Graph Processing on Geometric Data

Yiqiu Wang
MIT CSAIL
yiqiuw@mit.edu

Shangdi Yu
MIT CSAIL
shangdiy@mit.edu

Laxman Dhulipala
MIT CSAIL
laxman@mit.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

In many applications of graph processing, the input data is often generated from an underlying geometric point data set. However, existing high-performance graph processing frameworks assume that the input data is given as a graph. Therefore, to use these frameworks, the user must write or use external programs based on computational geometry algorithms to convert their point data set to a graph, which requires more programming effort and can also lead to performance degradation.

In this paper, we present our ongoing work on the GeoGraph framework for shared-memory multicore machines, which seamlessly supports routines for parallel geometric graph construction and parallel graph processing within the same environment. GeoGraph supports graph construction based on k-nearest neighbors, Delaunay triangulation, and β-skeleton graphs. It can then pass these generated graphs to over 25 graph algorithms. GeoGraph contains high-performance parallel primitives and algorithms implemented in C++, and includes a Python interface. We present four examples of using GeoGraph, and some experimental results showing good parallel speedups and improvements over the Higra library. We conclude with a vision of future directions for research in bridging graph and geometric data processing.

1 Introduction

Graphs are a fundamental way to represent relationships in data, and have a variety of real-world applications. For example, they are used in social network analysis, Internet analysis, machine learning, bioinformatics, and transportation planning. Due to the massive sizes of graphs today, analyzing graphs efficiently necessitates high-performance parallel programs. However, writing such programs can be challenging for non-experts in high-performance computing. Fortunately, there exists a variety of programming frameworks for efficient graph processing that reduce the burden on the user by allowing them to write programs using high-level functions, which the frameworks provide highly-optimized parallel implementations for under the hood (see [6, 7, 9, 16, 29, 32, 36, 41, 51, 66] for surveys of graph processing frameworks).

As far as we know, existing high-performance graph processing frameworks assume that the user provides input data in the format of a graph. While the data that one wishes to process is sometimes naturally provided in the form of a graph (e.g., social networks and Internet graphs), oftentimes the data is presented in the form of points in n-dimensional space (we refer to this type of data as *geometric data*), without any relationship information among the points. Although data analysis can be performed on the points themselves, it may be desirable to convert the geometric data into a graph and take advantage of graph algorithms to uncover better insights into the data. In particular, the graph would contain vertices that correspond to the original points, with an edge appearing between two vertices if their corresponding points are "similar enough". The output of the graph algorithms may then be used for further processing with geometric algorithms.

The approach of converting the original data into a graph is commonly used in machine learning to perform semi-supervised learning [54]. Here the data points are associated with feature vectors, and two data points are connected in the graph if their features are similar enough based on a function chosen for the application. One can then run a graph clustering algorithm on this graph, and each resulting cluster will correspond to objects that should have the same label in the original data set [11, 25, 34, 37, 38]. This approach can potentially produce higher-quality clusters than using a spatial clustering algorithm on the original data [34, 43]. Transportation planning is another example where the approach of converting data to a graph format is commonly used [4, 5, 14]. Here the original points may correspond to physical locations, and the edges between points are determined by route availability.

With most existing graph processing frameworks today, a user who wishes to process data that is not given in graph format is responsible for writing or using another tool to convert their data into a graph format that is compatible with

the graph framework that they are using. To ensure that the end-to-end running time is fast, the user needs to write or use efficient algorithms for data conversion, which can be non-trivial. This process often involves using routines from computational geometry, such as the Delaunay triangulation, nearest-neighbor searches, range searches, well-separated pair decompositions, and visibility tests. While there exists various parallel libraries that support graph generation from geometric data [3, 17, 24, 45], they do not have an interface with existing graph processing frameworks. Linking these libraries with graph frameworks significantly increases the burden on the user. Furthermore, even if the user is able perform the data conversion efficiently, the process will still perform unnecessary disk I/O's because existing graph frameworks often assume that the input data is stored on disk. These extra disk accesses can become a performance bottleneck if the rest of the application is running in memory. To improve programmability and performance, it is therefore important to have a unifying framework that supports both graph algorithms and computational geometry routines, with efficient methods for data conversion between the graph and geometric data formats. Such a framework can also benefit geometric algorithms that use graph algorithms as subroutines, such as density-based spatial clustering [63, 64] and motion planning [18].

This paper introduces our ongoing work on designing a high-performance framework, called GeoGraph, that bridges the gap between parallel graph processing and parallel computational geometry routines that are used for graph construction. GeoGraph is currently implemented for shared-memory multicore machines. GeoGraph is a C++ library with a Python interface, consisting of parallel algorithms for geometric graph generation and graph processing, as well as functions for reading and writing data. It combines geometric graph construction algorithms currently being developed within the Pargeo computational geometry library [1] with graph algorithms and data formats from the Graph Based Benchmark Suite [19, 21]. Users of GeoGraph will be able to generate a variety of common geometric graphs, construct efficient graph data structures, and run graph algorithms seamlessly within one Python session.

We demonstrate how to use GeoGraph API to write four examples of applications that combine geometric graph construction with graph algorithms: connected components on a filtered k-NN graph, hierarchical clustering on a k-NN graph, Euclidean minimum spanning tree using a Delaunay triangulation, and shortest paths on a β-skeleton graph. Experimentally, we show that running these algorithms completely in memory using GeoGraph is 3.72–7.35x faster than that of having to write the graph to disk and load it back into memory, which represents what users would have to do with existing tools. We also compare with Higra [46], an existing library that supports graph algorithms and geometric graph construction using SciPy [61] and scikit-learn [45]. Higra focuses on hierarchical clustering, and therefore supports a narrower set of algorithms than GeoGraph. We show that GeoGraph achieves 7.5–94.57x speedups over Higra. Our code is publicly available at https://github.com/ParAlg/GeoGraph.

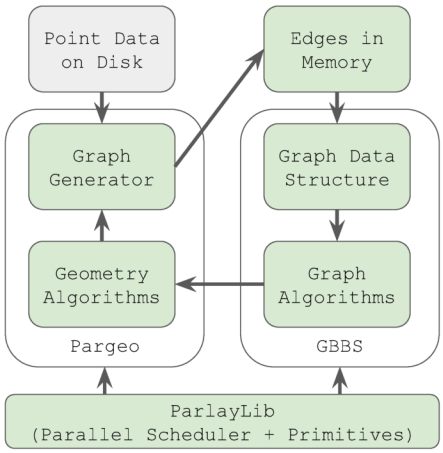

Figure 1: System architecture of GeoGraph.

We conclude with a vision of what we believe are important problems to address in order to fully bridge geometric data processing with graph processing.

2 GeoGraph Framework and API

In this section, we outline the architecture of GeoGraph and describe its application programming interface (API). We illustrate the components of GeoGraph in Figure 1.

GeoGraph is written in C++ and includes a Python interface, which enables both high performance and ease of use. GeoGraph uses graph algorithms from the publicly-available Graph Based Benchmark Suite (GBBS) [19, 21], and geometric graph construction algorithms from the Pargeo library [1], which is a work in progress. Internally, both libraries use the parallel scheduler and primitives of the ParlayLib library [8] to express parallelism and exploit a set of highly-optimized shared-memory parallel primitives.

The user can load a geometric point data set from disk using our data loader. Graphs can be generated from the geometric data using geometric algorithms from Pargeo, which we describe in Section 2.1. The output is an edge list, which is then passed to GBBS, where it is converted into a compressed graph representation, on which over 25 graph algorithms can then be run. Section 2.2 describes in more detail the graph representation and the algorithms available in GBBS. Depending on the application, the user can either directly use the output of the graph algorithm, or feed the output back into a geometry algorithm in Pargeo for further processing. GeoGraph combines the functionalities provided by GBBS and Pargeo seamlessly, and enables a user to use these functionalities in a Python interface without switching context.

2.1 Geometric Graph Construction

We now describe the geometric graph construction algorithms that are currently provided by GeoGraph.

***k*-Nearest Neighbor Graphs.** Our framework supports computing the k-nearest neighbor (k-NN) graph of a point data set. k-NN graphs have a variety of applications, such as graph clustering [11, 25, 34, 37, 38], manifold learning [56], outlier detection [31], and proximity search [13, 44, 49]. The k-NN graph is a directed graph on a set of P points in a metric space, such that P represents the vertex set, and a directed edge exists from vertex p to vertex q if the distance between p and q is among the k smallest distances from p to points in $P \setminus \{p\}$. We compute the k-NN by traversing a kd-tree, a binary tree data structure commonly used for k-NN queries [26]. A kd-tree traversal to compute k-NNs will first visit subtrees close to the input point, and prune farther tree nodes that cannot possibly contain the k-NNs. We first construct a kd-tree, then apply k-NN queries for all of the points in P, and finally generate k-NN graph based on the query results. To build the tree, we use a parallel splitting algorithm to split the points across the two children subtrees, and recursively construct each subtree in parallel. The queries are run in parallel in a data-parallel fashion.

Spatial Network Graphs. Spatial network graphs are a class of geometric graphs on which various graph metrics are often computed [4, 5]. We discuss the spatial network graphs in the context of point data sets in the Euclidean plane, which usually arise from geographic coordinates. The *Delaunay graph* is directed related to the Delaunay triangulation of a point set [18], where each edge of the triangulation is treated as an undirected edge with weight equal to the Euclidean distance between the two endpoints. The Delaunay graph is useful because its edges are a superset of that of other graphs, such as the Euclidean minimum spanning tree and β-skeleton graphs [35], both of which have a variety of real-world applications [2, 33, 47, 48, 57, 60, 62, 65]. We use the parallel incremental Delaunay triangulation implementation from the Problem Based Benchmark Suite [52].

The β-*skeleton* is defined for a point set P in the Euclidean plane, where each point in P is a vertex of the graph. There is an undirected edge between a pair of points p and q if for any other point r, the angle prq is smaller than a threshold derived from parameter β. The β-skeleton shares the same vertex set as the Delaunay graph, but only contains a subset of the Delaunay edges [60]. We use the kd-tree to construct the β-skeleton graph efficiently in parallel. Specifically, for each edge of the Delaunay graph in parallel, we determine whether to keep the edge by checking whether there exists a third point in a region defined by the edge and the parameter β. The check can be reduced to several range searches in a kd-tree. The β-skeleton generalizes other well known spatial network graphs, such as the Gabriel graph and the relative neighborhood graph [33, 35].

2.2 Parallel Graph Processing

In this section, we present our approach to parallel graph processing of geometric data sets in GeoGraph, which builds on the algorithms and data structures from the Graph Based Benchmark Suite (GBBS) for parallel graph processing [19, 21]. In what follows, we describe some of the key features of GeoGraph in the context of parallel graph processing.

Representing and Building Geometric Graphs. GeoGraph supports two graph representations, namely the *compressed sparse row (CSR)* and *edge/coordinate list (COO)* formats. In CSR, we are given two arrays, I and A, where the incident edges of a vertex v are stored in $\{A[I[v]], \ldots, A[I[v+1]-1]\}$. In COO, we are given an array of pairs (u,v) corresponding to edge endpoints. Our framework supports weighted graphs, where edge weights are interleaved with the neighbors of the vertex in the CSR format, and stored as the third entry in each edge tuple in the COO format.

When generating geometric graphs, we typically do not know the number of edges that will be present in the graph, or the number of edges that will be incident to each vertex before running the generation algorithm, and thus generating geometric graphs directly in a CSR format is difficult. Instead, we first generate the (weighted) edge list corresponding to the graph in COO format, and then supply this edge list to a procedure which builds a (weighted) graph in the CSR format. There are two main advantages to representing the graph in CSR format. First, representing the graph in this format enables us to apply lossless compression techniques from the Ligra+ framework [53], which are provided in GBBS. Second, representing the graph in CSR format is crucial in many parallel graph algorithms that perform random access to the edges incident to arbitrary vertices.

Applying Graph Algorithms to Geometric Graphs. GBBS provides fast and theoretically-efficient parallel solutions to over 25 important graph problems, ranging from basic problems such as connectivity and breadth-first search, to more challenging and computationally difficult problems such as k-truss, k-clique counting [50], minimum spanning forest, strong connectivity, and biconnectivity, among others [19, 21]. These algorithms are implemented using high-level primitives, such as functions over subsets of vertices and edges, and operations on parallel priority queues. GBBS provides optimized multicore implementations of these primitives under the hood. We describe some natural examples of running GBBS algorithms on geometrically-derived graphs in Section 3.

2.3 Python API

In this section, we now give an overview of the Python API in GeoGraph, which is illustrated in Figure 2. There are two main components of the API: the first component

```
from geograph import *
P = loadPoints(file)

edges = KNNGraph(P, k, epsilon = -1, weighted = False)
edges = DelaunayGraph(P, weighted = False)
edges = GabrielGraph(P, weighted = False)
edges = BetaSkeleton(P, beta, weighted = False)
```

edges

```
G = loadFromEdgeList(edges, symmetric = True, \
  weighted = False)

Output = G.HierarchicalAgglomerativeClustering(linkage)
Output = G.DeltaStepping(source, delta)
Output = G.MinimumSpanningTree()
Output = G.Components()
Output = G.BFS(source)
Output = G.PageRank()
Output = G.KCore()
...
```

Figure 2: The Python API of GeoGraph.

involves loading and processing a point data set to form a set of geometric graph edges, and the second component involves converting these edges into a graph in the CSR format, and running graph algorithms on this graph. We use the NumPy package [30] to keep intermediate data structures in memory and avoid disk I/Os.

The `loadPoints` function takes the file path of a point data set and returns a NumPy array `P` of shape `(#points, dimensionality)`. GeoGraph provides several functions to compute a geometric graph over `P` to form a graph in COO format (top of Figure 2). By default, the graph is unweighted, but a weighted graph using Euclidean distances can be generated by passing `True` for the `weighted` argument. For example, the user can construct an unweighted Delaunay graph on `P` by calling

```
>>> edges = DelaunayGraph(P)
```

This will return a NumPy array of shape `(#edges, 2)`, where each row contains the IDs of the two endpoints of an undirected edge. For weighted graphs, the edge list returned will be of shape `(#edges, 3)`, where the additional column contains the edge weights. The k-NN graph generator takes an additional parameter `epsilon`, and filters out edges with weight greater than `epsilon` when it is set to be a value greater than 0 (by default `epsilon` is set to -1, with no edges being filtered). All of the generators, except `KNNGraph`, generate symmetric graphs where each edge appears in both directions.

The edge list can be converted to a compressed graph representation by calling:

```
>>> G = loadFromEdgeList(edges)
```

This function takes additional parameters which enable a user to specify whether the edges should be made symmetric (`True` by default), and whether to take in edge weights (`False` by default). Then, numerous graph algorithms can be called from the compressed graph data structure `G` (bottom of Figure 2). For example, the following will compute the connected components of `G`, where the component IDs are stored in the list `C`:

```
>>> C = G.Components()
```

For `HierarchicalAgglomerativeClustering`, the user specifies a `linkage` parameter, such as `"single"`, `"average"`, or `"complete"`, as the linkage criteria [40].

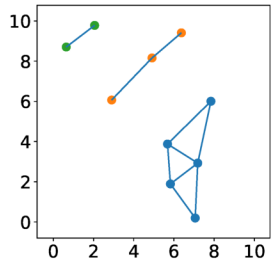

Figure 3: Example of running connected components on the 3-NN graph of a 2-dimensional point data set, where edges with weight greater than 3.2 are filtered out. The vertices of each color correspond to a connected component.

3 Examples of using GeoGraph

In this section, we illustrate some examples of using GeoGraph to run graph algorithms on graphs constructed from a geometric data set. We present some visualizations of the outputs on a small data set.

A good example is to consider applying graph clustering algorithms to geometric graphs. For example, consider computing the k-NN graph of a point set, generating the symmetrized (undirected) graph by making each directed edge bi-directional, and then applying a parallel connected components algorithm to this graph. To remove noise and produce more meaningful clusters, we can filter edges with weight larger than a certain value. The following code shows how to run connected components on a 3-NN graph that filters the edges with weight greater than 3.2. We construct a symmetrized graph data structure based on the k-NN edges, and then run the connected components algorithm, which returns the component ID of the vertices. A visualization of the components on a small data set is shown in Figure 3.

```
>>> from geograph import *
>>> P = loadPoints("data.csv")
>>> edges = KNNGraph(P, k=3, epsilon=3.2)
>>> G = loadFromEdgeList(edges, symmetric=True)
>>> C = G.Components()
```

We also consider applying hierarchical graph clustering algorithms to an input weighted graph. The output of these

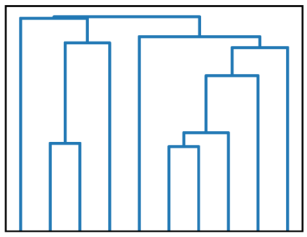

Figure 4: Example of running complete-linkage clustering on the 3-NN graph of a 2-dimensional point data set. We show the output, a corresponding dendrogram.

algorithms is usually a dendrogram representing the arrangement of clusters. Although efficient spatial hierarchical clustering algorithms exist, an important advantage of using a graph-based hierarchical clustering method is that the graph-based method can be run on a sparse geometric graph, like a k-NN graph with a small value of k or any of the spatial network graphs described in Section 2.1. By using an efficient graph-based hierarchical clustering method, like a hierarchical version of the SCAN algorithm [58], or a graph-based agglomerative clustering algorithm, we can potentially significantly outperform classic approaches that only use the input point set [34, 43]. The following code shows how to run complete-linkage clustering on a 3-NN graph. We construct a symmetric graph based on the k-NN edges, and then run the clustering algorithm on the graph, which returns a hierarchy corresponding to a dendrogram. A visualization of the dendrogram is shown in Figure 4.

```
>>> from geograph import *
>>> P = loadPoints("data.csv")
>>> edges = KnnGraph(P, k=3, weighted=True)
>>> G = loadFromEdgeList(edges, symmetric=True,
 ↪  weighted=True)
>>> CL =
 ↪  G.HierarchicalAgglomerativeClustering("complete")
```

A Euclidean minimum spanning tree (EMST) on a point data set has various applications, including being used in single-linkage clustering [28], network placement optimization [62], and approximating the Euclidean traveling salesman problem [59]. A well-known fact is that the EMST is a subset of the Delaunay triangulation of a graph [33]. We consider generating a graph containing edges of the Delaunay triangulation, and then passing the graph to a minimum spanning tree algorithm in GBBS, which is shown in the following code. A visualization of the minimum spanning tree on a small data set is shown in Figure 5.

```
>>> from geograph import *
>>> P = loadPoints("data.csv")
>>> edges = DelaunayGraph(P, weighted=True)
>>> G = loadFromEdgeList(edges, weighted=True)
>>> T = G.MinimumSpanningForest()
```

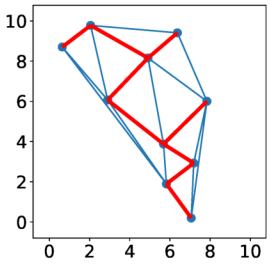

Figure 5: Example of running minimum spanning tree on the Delaunay triangulation graph of a 2-dimensional point data set. The edges of the minimum spanning tree are shown in red.

Finally, computing shortest paths on transportation and infrastructure networks is commonly used for planning [4]. These networks can be generated by constructing spatial networks on geometric data. We consider running the Δ-stepping single-source shortest paths algorithm [42] on the β-skeleton of a point set. The following code shows running the Δ-stepping algorithm with source vertex 0 and $\Delta = 0.01$ on a β-skeleton graph with $\beta = 2$ (the relative neighborhood graph).

```
>>> from geograph import *
>>> P = loadPoints("data.csv")
>>> edges = BetaSkeleton(P, beta=2, weighted=True)
>>> G = loadFromEdgeList(edges, weighted=True)
>>> C = G.DeltaStepping(source=0, delta=0.01)
```

4 Benchmarking

In this section, we benchmark the performance of GeoGraph on the four examples in Section 3. Our implementations are all parallel, except for complete-linkage clustering, whose parallelization is a work in progress. We compare with Higra [46] (version 0.6.4) for computing a hierarchical clustering on a k-NN graph and minimum spanning tree on the Delaunay graph (they do not support the other two examples). The Higra framework has a Python interface, and calls the SciPy [61] and scikit-learn [45] libraries serially to construct geometric graphs. In addition, to demonstrate the advantage of running graph generation and graph algorithms in memory without transferring intermediate data to and from disk, we compare with a version of GeoGraph where the edges generated are first written to disk and then loaded back into memory (GeoGraph-Disk).

We perform all of our experiments on a `c5.18xlarge` instance on Amazon EC2. The instance has 2 $\times$ Intel Xeon Platinum 8124M (3.00GHz) CPUs for a total of 36 cores with two-way hyper-threading, and 144 GB of RAM. The storage uses Amazon EBS with a General Purpose SSD. We use two synthetic 2-dimensional data sets, each with 10 million points. We generate the *blobs* data set using scikit-learn's [45] generator, which produces samples from isotropic Gaussian

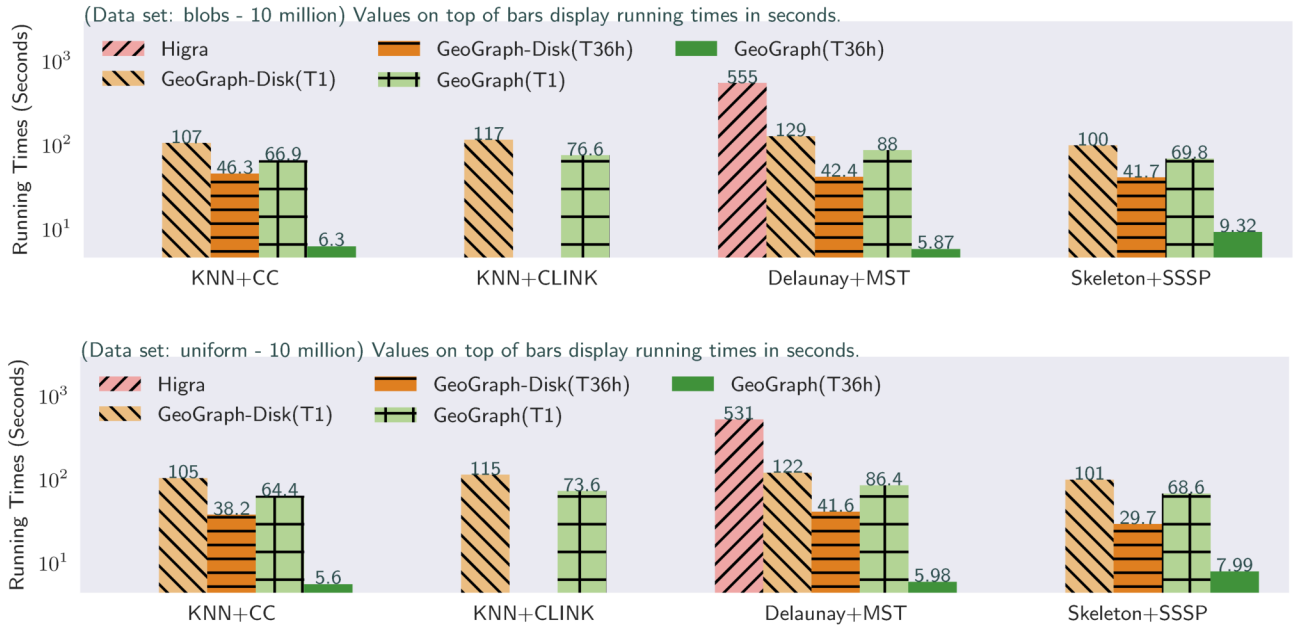

Figure 6: Comparison between GeoGraph, GeoGraph with disk I/O, and Higra. T1 corresponds to the serial time and T36h corresponds to the parallel time on 36 cores with hyper-threading. KNN+CC is connected components on the 3-NN graph; KNN+CLINK is complete-linkage clustering on the 3-NN graph; Delaunay+MST is minimum spanning tree on the Delaunay graph; and Skeleton+SSSP is Delta-stepping on the 2-skeleton with Delta set to 0.01. The running time in seconds is displayed at the top of each bar.

blobs with varying variances. We also use a *uniform* data set consisting of data points generated uniformly at random in a square of side length 10.

In Figure 6, we show the running times of the methods on the four examples. Using 36-cores with hyper-threading, GeoGraph achieves 7.49x–14.99x self-relative speedup. Compared with the baseline that writes the graph to disk and loads it back into memory, GeoGraph achieves 3.72–7.35x speedup.[1]

Compared to Higra, our minimum spanning tree computation on the Delaunay graph is 104–112x faster. This is due to GeoGraph supporting faster graph generation and a more optimized minimum spanning tree algorithm. We encountered internal errors in Higra when computing the k-NN graph and complete-linkage clustering on the data sets with 10 million points. Therefore, we also tested Higra on smaller data sets with 100 thousand points, drawn from the same distributions. On 36 cores with hyper-threading, Higra takes 1.53 and 1.58 seconds for the blobs and uniform data sets, respectively, while GeoGraph takes 0.204 and 0.207 seconds. Overall, our graph generation is 11.4–112.9x faster than Higra while our graph algorithms are 6.63–13.69x faster. While Higra generates graphs by calling the Python libraries SciPy for the Delaunay graph and scikit-learn for the k-NN graph serially, we use optimized parallel C++ implementations to convert geometric data sets to graphs. Overall, GeoGraph is 7.5–94.57x faster than Higra.

[1]The disk I/O times varied across runs, likely due to the nondeterminism of Amazon EBS.

5 Conclusion and Vision

Geometric data processing and graph processing have been the subjects of intense theoretical and empirical studies over the past few decades, but unfortunately these subjects have often been considered in isolation, especially from a systems perspective. In this paper, we have presented our ongoing work on GeoGraph, a shared-memory multicore framework that enables users to run graph algorithms on graphs constructed with computational geometry primitives within the same interface. Using GeoGraph, we have shown how users can easily perform tasks on graphs generated from on geometric data sets, including clustering, finding the minimum spanning tree, and computing shortest paths, which require high-performance geometry and graph processing primitives.

For future work, we are interested in using GeoGraph to study algorithms and applications in geometry that can benefit from using efficient graph algorithms internally. For example, computing the shortest path on a visibility graph is an essential building block of motion planning algorithms [18]. Geometric clustering algorithms, such as DBSCAN [23, 63] and hierarchical spatial clustering [12, 64], also rely on underlying connected components or minimum spanning tree algorithms. These applications provide another context for

the interaction between geometric data processing and graph processing, and further stems the need for a unified framework.

The graph construction methods that are currently supported in GeoGraph work well for low-dimensional data sets. We believe that there is a significant potential for future work on designing efficient graph construction algorithms for high-dimensional data sets, such as approximate k-NN graph construction, which has applications to data mining and information retrieval [10, 15, 22, 27, 39, 55]. Studying how different graph construction methods affect the quality of the downstream tasks is an important research direction.

Another interesting challenge is to design efficient visualization techniques which present both the input point set and geometric graph realizations of it, and illustrate algorithm outputs on both. We envision future systems to support visualization techniques that are parallel and scale to large data sets.

Due to the rapid changes in real-world data, future systems should also consider the setting where the input data set receives batches of updates (point insertions, deletions, or modifications). These systems would then update the associated graph, which could be dynamically represented using an efficient parallel batch-dynamic graph data structure (e.g., [20]). Finally, due to the large variety of computing resources available today with different performance characteristics, it is crucial for future systems to support efficient processing on different types of hardware, including multicore CPUs, GPUs, distributed clusters, disks, and domain-specific accelerators.

We envision a future with portable high-performance systems that can seamlessly bridge geometric data processing and graph processing on both static and dynamic data. Such systems will provide novel, interpretable, and high-quality insights into the structure of geometric data sets using graph processing, while using parallel algorithms that run in near-linear work in the sparsity of the input graph, thus potentially achieving significant speedups over existing quadratic-work point set clustering and analysis methods.

Acknowledgments

This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Pargeo, an open source library for parallel algorithms in computational geometry. https://github.com/wangyiqiu/pargeo, 2021.

[2] David J. Aldous and Julian Shun. Connected Spatial Networks over Random Points and a Route-Length Statistic. *Statistical Science*, 25(3):275–288, 2010.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

[4] Marc Barthelemy. Spatial networks. *Physics Reports*, 499(1-3):1–101, Feb 2011.

[5] Marc Barthelemy. *Morphogenesis of Spatial Networks*. Jan 2018.

[6] Maciej Besta, Dimitri Stanojevic, Johannes de Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph processing on FPGAs: Taxonomy, survey, challenges. *CoRR*, abs/1903.06697, 2019.

[7] Siddharth Bhatia and Rajiv Kumar. Review of graph processing frameworks. In *IEEE International Conference on Data Mining Workshops*, pages 998–1005, 2018.

[8] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures*, page 507–509, 2020.

[9] Angela Bonifati, George Fletcher, Jan Hidders, and Alexandru Iosup. *A Survey of Benchmarks for Graph-Processing Systems*, pages 163–186. 2018.

[10] Antoine Boutet, Anne-Marie Kermarrec, Nupur Mittal, and François Taïani. Being prepared in a sparse world: the case of KNN graph construction. In *IEEE International Conference on Data Engineering*, pages 241–252, 2016.

[11] Maria R. Brito, Edgar L. Chávez, Adolfo J. Quiroz, and Joseph E. Yukich. Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35(1):33–42, 1997.

[12] Ricardo Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Transactions on Knowledge Discovery from Data*, pages 5:1–5:51, 2015.

[13] Edgar Chávez and Eric Sadit Tellez. Navigating k-nearest neighbor graphs to solve nearest neighbor searches. In *Advances in Pattern Recognition*, pages 270–280, 2010.

[14] Daniel Chemla, Frédéric Meunier, and Roberto Wolfler Calvo. Bike sharing systems: Solving the static rebalancing problem. *Discrete Optimization*, 10(2):120–146, 2013.

[15] Jie Chen, Haw-ren Fang, and Yousef Saad. Fast approximate kNN graph construction for high dimensional data via recursive Lanczos bisection. *Journal of Machine Learning Research*, 10(9), 2009.

[16] Miguel E. Coimbra, Alexandre P. Francisco, and Luís Veiga. An analysis of the graph processing landscape.

Journal of Big Data, 8(1):55, 2021.

[17] Ryan R. Curtin, Marcus Edel, Mikhail Lozhnikov, Yannis Mentekidis, Sumedh Ghaisas, and Shangtong Zhang. mlpack 3: a fast, flexible machine learning library. *Journal of Open Source Software*, 3:726, 2018.

[18] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[19] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 393–404, 2018.

[20] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 918–934, 2019.

[21] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. The graph based benchmark suite (GBBS). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems and Network Data Analytics*, 2020.

[22] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *International Conference on World Wide Web*, page 577–586, 2011.

[23] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[24] Efi Fogel and Monique Teillaud. The computational geometry algorithms library CGAL. *ACM Commun. Comput. Algebra*, 49(1):10–12, June 2015.

[25] Pasi Franti, Olli Virmajoki, and Ville Hautamaki. Fast agglomerative clustering using a k-nearest neighbor graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1875–1881, 2006.

[26] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 7 1976.

[27] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, January 2019.

[28] John C. Gower and Gavin J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 18(1):54–64, 1969.

[29] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology*, 34(2):339–371, 2019.

[30] Charles R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[31] Ville Hautamaki, Ismo Karkkainen, and Pasi Franti. Outlier detection using k-nearest neighbour graph. In *International Conference on Pattern Recognition*, volume 3, pages 430–433, 2004.

[32] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Comput. Surv.*, 51(3), June 2018.

[33] Jerzy W. Jaromczyk and Godfried T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80(9):1502–1517, 1992.

[34] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[35] David G. Kirkpatrick and John D. Radke. A framework for computational morphology. In *Computational Geometry*, volume 2 of *Machine Intelligence and Pattern Recognition*, pages 217–248. 1985.

[36] Ning Liu, Dong-sheng Li, Yi-ming Zhang, and Xiong-lve Li. Large-scale graph processing systems: a survey. *Frontiers of Information Technology & Electronic Engineering*, 21(3):384–404, 2020.

[37] Małgorzata Lucińska and Sławomir T. Wierzchoń. Spectral clustering based on k-nearest neighbor graph. In *Computer Information Systems and Industrial Management*, pages 254–265, 2012.

[38] Markus Maier, Matthias Hein, and Ulrike von Luxburg. Optimal construction of k-nearest-neighbor graphs for identifying noisy clusters. *Theoretical Computer Science*, 410(19):1749–1764, 2009.

[39] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[40] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[41] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[42] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[43] Nicholas Monath, Avinava Dubey, Guru Guruganesh, Manzil Zaheer, Amr Ahmed, Andrew McCallum, Gokhan Mergen, Marc Najork, Mert Terzihan, Bryon Tjanaka, Yuan Wang, and Yuchen Wu. Scalable

bottom-up hierarchical clustering. *arXiv preprint arXiv:2010.11821*, 2020.

[44] Rodrigo Paredes and Edgar Chávez. Using the k-nearest neighbor graph for proximity searching in metric spaces. In *String Processing and Information Retrieval*, pages 127–138, 2005.

[45] Fabian Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[46] Benjamin Perret, Giovanni Chierchia, Jean Cousty, Silvio J. Guimaraes, Yukiko Kenmochi, and Laurent Najman. Higra: Hierarchical graph analysis. *SoftwareX*, 10:100335, 2019.

[47] Franco P. Preparata and Michael I. Shamos. *Computational Geometry*. Springer, 1990.

[48] John Radke and Anders Flodmark. The use of spatial decompositions for constructing street centerlines. *Geographic Information Sciences*, 5(1):15–23, 1999.

[49] Thomas B. Sebastian and Benjamin B. Kimia. Metric-based shape retrieval in large databases. In *Proceedings of the International Conference on Pattern Recognition (ICPR)*, 2002.

[50] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. *arXiv preprint arXiv:2002.10047*, 2020.

[51] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on GPUs: A survey. *ACM Comput. Surv.*, 50(6):81:1–81:35, January 2018.

[52] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 68–70, 2012.

[53] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *IEEE Data Compression Conference*, pages 403–412, 2015.

[54] Amarnag Subramanya and Partha Pratim Talukdar. *Graph-Based Semi-Supervised Learning*. Morgan & Claypool Publishers, 2014.

[55] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Rand-NSG: Fast accurate billion-point nearest neighbor search on a single node. In *Conference on Neural Information Processing Systems*, pages 13748–13758, 2019.

[56] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[57] Godfried T. Toussaint and Constantin Berzan. Proximity-graph instance-based learning, support vector machines, and high dimensionality: An empirical comparison. In *Machine Learning and Data Mining in Pattern Recognition*, pages 222–236, 2012.

[58] Tom Tseng, Laxman Dhulipala, and Julian Shun. Parallel index-based structural graph clustering and its approximation. In *ACM SIGMOD International Conference on Management of Data*, 2021.

[59] Vijay V. Vazirani. *Approximation Algorithms*. Springer Publishing Company, Incorporated, 2010.

[60] Remco C. Veltkamp. The γ-neighborhood graph. *Computational Geometry*, 1(4):227–246, 1992.

[61] Pauli Virtanen et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, 2020.

[62] Peng-Jun Wan, Grucia Călinescu, Xiang-Yang Li, and Ophir Frieder. Minimum-energy broadcasting in static ad hoc wireless networks. *Wireless Networks*, 8(6):607–617, 2002.

[63] Yiqiu Wang, Yan Gu, and Julian Shun. Theoretically-efficient and practical parallel DBSCAN. In *ACM SIGMOD International Conference on Management of Data*, page 2555–2571, 2020.

[64] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In *ACM SIGMOD International Conference on Management of Data*, 2021.

[65] Peter Willett. Recent trends in hierarchic document clustering: A critical review. *Information Processing & Management*, 24(5):577–597, 1988.

[66] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.