# Fast Parallel Algorithms and Library for Spatial Clustering and Computational Geometry

by

Yiqiu Wang

B.A., Rice University (2016)
S.M., Rice University (2018)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023



Authored by: Yiqiu Wang
Department of Electrical Engineering and Computer Science
May 19, 2023

Certified by: Julian Shun
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

# Fast Parallel Algorithms and Library for Spatial Clustering and Computational Geometry

by

Yiqiu Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

This thesis presents novel parallel shared-memory multi-core algorithms, implementations, and frameworks for efficiently solving large-scale spatial clustering and computational geometry problems. The primary focus is on designing theoretically-efficient and practical algorithms that can handle the increasing demand for faster processing speeds in spatial data sets.

In the first part of the thesis, we introduce new parallel algorithms and framework for spatial clustering. We design new parallel algorithms for exact and approximate DBSCAN, which match the work complexity of the best sequential algorithms while maintaining low depth. Extensive experiments demonstrate that our algorithms achieve massive speedup over existing algorithms and can efficiently process large-scale data sets. We also present new parallel algorithms for hierarchical DBSCAN (HDBSCAN) and Euclidean minimum spanning tree (EMST), including several theoretical results and practical optimizations. Furthermore, we propose a method to generate a dendrogram from the minimum spanning tree (MST) of the HDBSCAN or EMST problem. The EMST also solves single-linkage clustering. Lastly, we also design a framework for implementing parallel grid-based clustering algorithms.

The second part of the thesis introduces our contributions to parallel algorithms and a library for computational geometry. We contribute to three problems in computational geometry: a new parallel reservation-based algorithm that can express both randomized incremental convex hull and quickhull algorithms; a sampling-based algorithm to reduce work for the smallest enclosing ball problem; and a parallel batch-dynamic data structure for dynamic closest pair problem. We also introduce ParGeo, a library for parallel computational geometry that provides various parallel geometric algorithms, data structures, and graph generators. Our experimental evaluations show significant speedups achieved by our proposed algorithms across

different problems.

Overall, this thesis demonstrates that parallel shared-memory multi-core algorithms, implementations, and frameworks can efficiently solve large-scale spatial clustering and computational geometry problems both in theory and practice.

Thesis Supervisor: Julian Shun
Title: Associate Professor of Electrical Engineering and Computer Science

# Acknowledgments

I am humbled and grateful to express my deepest appreciation for the individuals who have been instrumental in my PhD journey.

First and foremost, I extend my heartfelt thanks to my thesis advisor, Julian Shun. He not only admitted me to MIT but also provided me with unwavering support, guidance, and patience throughout my entire PhD. His research philosophy and ideas have shaped my thesis, and together we have published research with real-world impact. Despite the challenges posed by the global pandemic of COVID-19, he created a supportive lab environment that kept us connected and thriving over the years.

I am also deeply indebted to my thesis committee members - Yan Gu and Jonathan Ragan-Kelly - for their invaluable insights, ideas, and direction that helped shape this thesis. Yan Gu has been an exceptional collaborator whose contributions have been indispensable in many of our papers. Jonathan Ragan-Kelly's thought-provoking questions during my Research Qualification Exam helped guide my research since then.

I am grateful for all of my collaborators who have contributed their time, energy, and ideas to this project: Laxman Dhulipala for his insightful ideas and visions; Shangdi Yu for her exchange of ideas and motivation; Rahul Yesantharao and Dev Chheda for their tireless work on the projects.

I would like to thank all current and past members and students of our lab who made this experience fun, exciting, and supportive: Changwan Hong, Siddhartha Jayanti, Junhong Lin, Quanquan C. Liu, Jessica Shi (who organized ParAlg Game Night), Tom Tseng. I especially remember the many Krunker game nights where we laughed and had fun together as a team. I would also like to give a special thank you to our administrative assistant Linda Lynch who has always being caring and helpful.

I am grateful to Srini Devadas for being my academic advisor and for his time and wisdom, which have always been motivating. I also want to acknowledge the MIT Global Language Program for providing classes that enabled me to minor in the Japanese language and eventually pass the JLPT N3. I would like to thank all my teachers, Takako Aikawa, Masami Ikeda, Wakana Maekawa, and Emiko O. Rafique. Throughout my learning journey, I would also like to specially thank Shummei Ekawa, Lisa Huang, Masato Oono, and Zi Song Yeoh for doing language exchange with me,

being my study buddy, and fostering memorable friendships.

My life at MIT has been greatly enriched by many other individuals whom I am grateful for their presence. I would like to thank the members of the MIT badminton club with whom I spent three years improving my skills. Additionally, I would like to express my appreciation for the 70 Amherst dorm community which has been a central part of my graduate student life. I am also deeply grateful to my peers at CSAIL who have provided invaluable camaraderie and inspiration throughout my time at MIT. Their insights and perspectives have challenged me to grow as a researcher and as a person.

I would like to acknowledge the immense impact that my girlfriend Cathy Chen has had on my life. Together we have navigated the challenges of the final years of graduate school and shared many conversations about our passions, beliefs, and aspirations. Our shared journey has been both challenging and rewarding, and I am grateful for her unwavering support.

Finally, I want to express my heartfelt gratitude to my parents for their unconditional love and support throughout my academic journey. Their encouragement and guidance have been instrumental in helping me pursue my dreams; without them, I could not have come this far.

*To my family.*

# Contents

# Part I

# Introduction

# 1 Introduction

## 1.1 Motivation

**Growing Data Sets and Demand** In recent years, there has been a growing demand for faster processing speeds for tasks related to spatial clustering and computational geometry. Data scientists and researchers use spatial clustering to recognize patterns, analyze images, retrieve information, and perform compression. As spatial data sets continue to grow in size and complexity with millions and billions of data points [92, 259, 153, 118, 80], it has become increasingly challenging to process them efficiently. Spatial clustering algorithms are commonly used for clustering GPS data from mobile devices or social network data based on geographical proximity, with social media like Facebook having close to 3 billion users as of 2023. Some of these algorithms can also be applied to detect anomalies in massive data sets such as credit card transactions or network traffic logs [200, 158].

In addition, researchers in the computational geometry and graphics community process spatial data to obtain nearest neighbors, search for neighbors defined by a range, generate graphs for further data mining, and detect collision among objects. Algorithms like the closest pair algorithm are widely used for identifying similarities between proteins or DNA sequences in computational biology. As another example, the well-separated pair decomposition (WSPD) algorithm is important for molecular dynamics simulations that identify pairs of particles that are far enough apart [130, 129]. With these large and growing data sets, efficient processing has become increasingly challenging.

With such rise of big data, the running time of programs that process these data sets can significantly impact the costs in terms of money and time. In both research and industry, reducing the time-to-completion of tasks has been shown to increase productivity and improve user experience.

**Large-Scale Computing** Moore's law is a widely known principle in the field of computer hardware that predicts the exponential growth of transistor density on a microchip [175], which has historically led to an increase in clock speeds of single core machines at a rate of approximately 30% per year since the mid-1970s [156]. This increase in processing power has allowed for significant advancements in computing technology. However, this trend has not continued indefinitely. Around the

mid-2000s, Dennard scaling, which refers to the ability to reduce transistor size while maintaining constant power density [87], stopped due to physical limitations of hardware. This means that even though we can still fit more transistors onto a microchip, we cannot continue to increase clock speeds at the same rate as before without running into issues with power consumption and heat dissipation.

The physical limitations of hardware that led to the end of Dennard scaling have had a significant impact on the development of modern computing technology. Meanwhile, the explosion of data set sizes calls for greater processing power, and has led to the emergence of parallel processing, which breaks down a complex task into smaller sub-tasks than can be executed simultaneously across multiple processors. This approach leads to parallel processors and machines that achieve faster speed without relying solely on the increase in clock speed. Examples include distributed clusters, multi-core central processing units (CPUs), graphics processing units (GPUs), field programmable gate arrays (FPGAs), and other parallel accelerators.

The shift in processor technology towards multi-core processing has often been referred to as the "multi-core revolution" [156]. We utilize shared-memory multi-core machines, where different cores have access to a shared global memory. The technology has become ubiquitous today with most personal computers and smart phones. Commodity shared-memory multi-core machines have also become prevalent, supporting up to terabytes of memory and providing efficient solutions for many problems in large-scale computing, and are available at a reasonable price [213]. For example, Amazon EC2 offers powerful machines like r5.24xlarge with 2 $\times$ Intel Xeon Platinum 8175M (2.50 GHz) CPUs for a total of 48 two-way hyper-threaded cores and 768 GB of RAM at just about \$6 per hour at the time of this writing.[1]

While general purpose GPUs have emerged as a viable option for parallel computing in recent years, they are designed for workloads that require massive amounts of parallel processing power. More specifically, they are optimized for workloads that involve a large amount of data that can be processed using simple operations independently. Unlike CPUs, which usually have a relatively small number of cores that are optimized for complicated tasks, GPUs have thousands of cores optimized for simpler calculations. The GPU architecture has made them more suitable for certain kind of tasks, such as graphics rendering, and multiplying matrices. However, there are a few disadvantages of using the GPUs for general purpose computing. Well-known challenges are that the GPUs are difficult to program, and when compared with commodity multi-core CPU machines, they also have a smaller main memory. At the time of this writing, the amount of memory carried by an Nvidia H100 GPU is 80 GB, which is the largest on the market, whereas shared-memory CPU machines

[1] `https://aws.amazon.com/ec2/pricing/on-demand/`

can have terabytes of memory. While it is possible to utilize GPU computing in a distributed setting, the communication overhead and data partitioning will lead to new challenges. Therefore, while GPUs offer advantages over CPUs on certain tasks, multi-core CPUs are more appealing due to their better programmability, and larger main memory.

**Our Approach** This thesis argues that shared-memory multi-core machines offer a sweet spot between programmability and efficiency, which makes them ideal for solving many problems in spatial data processing, especially those that deal with a large amount of data. With the growing size of data sets and an increasing demand for faster processing speeds, researchers in spatial data processing have been exploring new ways to efficiently process spatial data. One of the key motivations for this thesis is the need to develop efficient parallel shared-memory algorithms and programming frameworks that continue to deliver performance improvements even at the end of Moore's Law. With powerful commodity multi-core shared-memory machines readily available, this thesis aims to make researchers better equipped to tackle the computational challenges in spatial clustering and computational geometry. The following will be described in the rest of this section.

- In Section 1.2, we describe parallel shared-memory algorithms and programming framework for spatial clustering.
- In Section 1.3, we describe parallel shared-memory algorithms and library for computational geometry.
- In Section 1.4, we describe a summary of the thesis contribution, the thesis statement, and a list of included papers.

## 1.2 Parallel Spatial Clustering

Cluster analysis, commonly known as clustering, is the general task of grouping similar objects into groups which are known as clusters. It is a common method of data analysis, and is used in various fields, including pattern recognition, image analysis, information retrieval, bioinfomatics, data compression, computer graphics and machine learning. Density-based clustering is an important type of clustering algorithm, where clusters are identified such that each of them is a contiguous region of high-density points separated by low-density ones. In this thesis, we study parallel algorithms for density-based spatial clustering. We study one of the most

widely-used density-based spatial clustering methods, which is the *density-based spatial clustering of applications with noise* (*DBSCAN*) method by Ester et al. [95]. We also study *hierarchical density-based spatial clustering of applications with noise* (*HDBSCAN**) [61], which is later developed to enable users to explore clusters at different scales.

The thesis first studies new parallel algorithms for exact and approximate Euclidean DBSCAN that match the work complexity of the best sequential algorithms while maintaining low depth.[2] The algorithms are designed to bridge the gap between theory and practice. For exact 2D DBSCAN, we design several parallel algorithms that use either box or grid construction for partitioning points. Additionally, we use three parallel procedures to determine connectivity among core points: Delaunay triangulation, unit-spherical emptiness checking with line separation, and bichromatic closest pairs. For higher-dimensional exact DBSCAN, an algorithm based on solving the higher-dimensional bichromatic closest pairs problem is provided. Unlike many existing parallel algorithms, these exact algorithms produce the same results according to the standard definition of DBSCAN, without sacrificing clustering quality. For approximate DBSCAN, a new algorithm is designed that uses parallel quadtree construction and querying. This algorithm returns the same result as the sequential approximate algorithm by Gan and Tao [104].

This thesis also studies HDBSCAN*, and its related problem Euclidean minimum spanning tree (EMST). We present practical alorithms for these problems and prove that the theoretical work of the implementations matches their state-of-the-art counterparts, while having polylogarithmic depth. Several other theoretical results are presented. An EMST algorithm with subquadratic work and polylogarithmic depth is proposed based on a subquadratic-work sequential algorithm by Callahan and Kosaraju [58]. An HDBSCAN* algorithm for two dimensions with $O(\mathsf{minPts}^2 \cdot n \log n)$ work is also presented, matching the sequential algorithm by Berg et al. [85], along with $O(\mathsf{minPts} \cdot \log^2 n)$ depth. Finally, a work-efficient parallel algorithm for approximate OPTICS based on the sequential algorithm by Gan and Tao [105] is proposed.

To solve HDBSCAN*, the thesis also discusses an algorithm to generate a dendrogram from the MST of the HDBSCAN* or EMST problem. This solves the single-linkage clustering problem and gives a reachability plot for HDBSCAN*. The proposed method uses a work-efficient parallel divide-and-conquer approach that generates an Euler tour on the tree, splits it into multiple subtrees recursively gen-

[2]We use the work-depth model to analyze parallel algorithms' theoretical efficiency, where work refers to the number of operations used and depth is the length of the longest sequence of dependence. We explain the model further in later sections.

erates dendrograms for each subtree before gluing them back together. An in-order traversal of this dendrogram gives us the reachability plot. The parallel dendrogram algorithm is of independent interest as it can also be applied to generate dendrograms for other clustering problems. Finally, optimized parallel implementations of our EMST and HDBSCAN* algorithms are provided in this thesis. A memory optimization technique that avoids computing and materializing many well-separated decomposition (WSPD) pairs significantly improves our algorithms' performance (up to 8x faster and 10x less space).

We conducted a comprehensive experiments on synthetic and real-world datasets, comparing the performance of our implementations to optimized sequential and existing parallel algorithms for DBSCAN and HDBSCAN*. Our exact DBSCAN implementations achieved a self-relative speedup of 2-89x (24x on average) and 5-33x (16x on average) over the fastest sequential implementations, while our approximate DBSCAN implementations achieved a self-relative speedup of 14-44x (24x on average). Compared to existing parallel algorithms, our fastest exact implementations were faster by up to orders of magnitude (16-6102x). For HDBSCAN, our implementation achieved a speedup of 11.13-46.69x over the fastest sequential implementations and was at least an order of magnitude faster than existing sequential and parallel implementations. We also demonstrate that our clustering algorithms can process very large data sets efficiently just on a single mahcine. We compare the running times of our fastest exact DBSCAN implementation with the state-of-the-art distributed implementation rpdbscan on several large-scale data sets. The results in Figure 1 show that our parallel exact DBSCAN achieves several orders of magnitudes of speedup over rpdbscan using the same or fewer number of cores, which is attributed to lower communication costs in shared-memory and a better algorithm.

Last but not least, grid-based clustering algorithms [205, 238, 16, 258, 117, 19, 248, 253, 65] have emerged as a popular approach for handling large multi-dimensional point data sets. These algorithms partition the data space into a grid structure and form clusters directly from the cells within the grid. There has been little effort to generalize their implementations and enable simple and fast parallel programming for spatial data sets. In this thesis, we address this gap by developing a framework for parallel grid-based clustering that provides high-level programming constructs for implementing various parallel grid-based clustering algorithms efficiently. We show that we can implement multiple grid clustering algorithms with high parallel speedup using our framework.

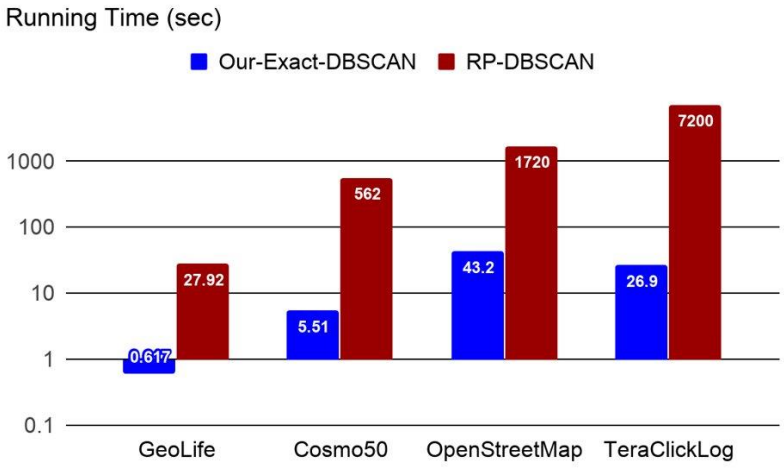



Figure 1: Comparing the running time (seconds) of our parallel implementation of DBSCAN with the state-of-the-art rpdbscan under varying $\epsilon$ parameter. The largest data set, *TeraClickLog*, has over 4 billion points and 13 dimensions.

.

## 1.3 Parallel Computational Geometry

Looking at a broader perspective beyond spatial clustering, computational geometry algorithms have significant applications in various domains such as computer graphics, robotics, computer vision, and geographic information systems [86, 193]. This thesis describes efficient libraries of computational geometry algorithms that the users can easily use in their high-level applications. While there are numerous libraries available for computational geometry, most of them do not support parallel processing [137, 96]. This thesis introduces the ParGeo library for parallel computational geometry that provides a rich set of parallel algorithms for geometric problems and data structures such as $k$d-trees, $k$-nearest neighbor ($k$-NN) search, range search, well-separated pair decomposition, Euclidean minimum spanning tree, spatial sorting and geometric clustering. ParGeo also includes a collection of geometric graph generators like $k$-nearest neighbor graphs and various spatial networks. The algorithms from ParGeo can either run sequentially or using parallel schedulers such as OpenMP or Cilk.

We also present our contributions to three different problems in computational geometry. Firstly, we introduce a new parallel reservation-based algorithm that can express both the randomized incremental convex hull algorithm and the quickhull

algorithm. Our algorithm adds multiple points in parallel per round and resolves conflicts using a reservation technique. Additionally, we propose a sampling-based algorithm to reduce the work for the smallest enclosing ball problem and provide the first parallel implementation of Welzl's classic algorithm. Thirdly, we design a theoretically-efficient and practical parallel batch-dynamic data structure for dynamic closest pair. Our solution is based on efficiently maintaining a sparse partition of points in parallel and takes a batch update of size $m$ to maintain the closest pair in $O(m(1+\log((n+m)/m)))$ expected work and $O(\log(n+m)\log^*(n+m))$ depth *whp* Finally, we introduce a new parallel batch-dynamic binary heap that we use in our data structure for dynamic closest pair, which itself may be of independent interest.

We test our proposed algorithms and library extensively on both synthetic and real-world data sets. We compared the performance of our parallel implementations against optimized sequential baselines. The results of our experiments show that our best convex hull implementation achieves up to 44.7x self-relative speedup and up to 559x speedup against the best existing sequential implementation for $\mathbb{R}^2$, and up to 24.9x self-relative speedup and up to 124x speedup against the best existing sequential implementation for $\mathbb{R}^3$. Our sampling-based smallest enclosing ball algorithm achieves up to 27.1x self-relative speedup and up to 178x speedup against the best existing sequential implementation for $\mathbb{R}^2$ and $\mathbb{R}^3$. Across all implementations in ParGeo, we achieve self-relative parallel speedups of 8.1–46.61x (on average 23.15x). We summarize the parallel speedups in Figure 2. The ParGeo library is publicly available,[3] and more detailed experimental evaluations are presented in subsequent sections of this thesis.

## 1.4 Summary of Contributions

We summarize the contribution of this thesis below with pointers to the respective sections:

1. We describe new parallel algorithms for exact and approximate DBSCAN. The algorithms match the work complexity of the best sequential algorithms while maintaining low depth. We perform an extensive experiments showing our algorithms achieve massive speedup over the existing algorithms, and is able to process large-scale data sets efficiently. We describe these algorithms in Section 4.

2. We describe new parallel algorithms for HDBSCAN* and EMST, including several theoretical results and pratical optimizations. We also proposes an al-

[3] `https://github.com/ParAlg/ParGeo`

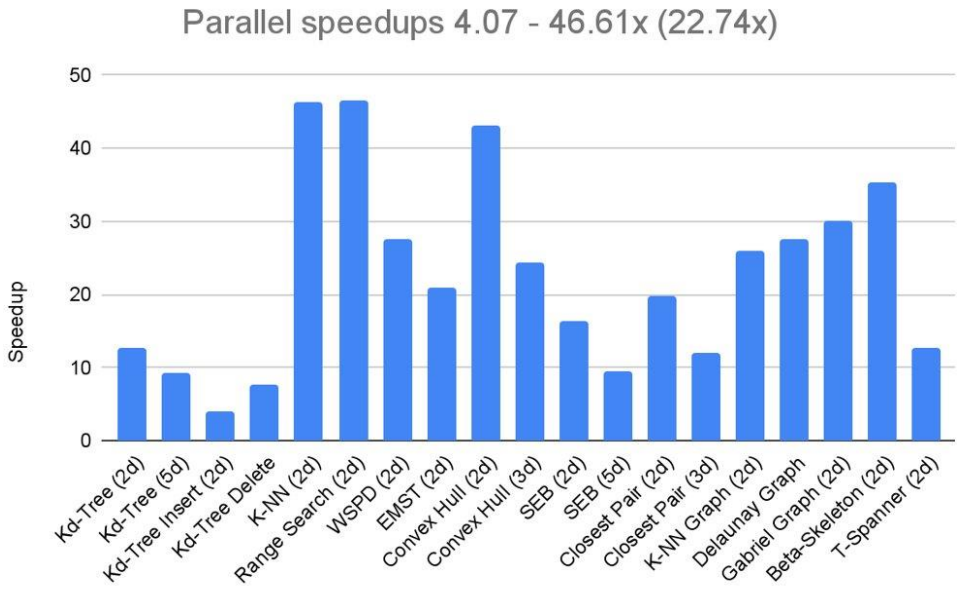



Figure 2: Parallel speedups on a machine with 36 cores for ParGeo implementations on uniform hypercube data sets of varying dimensions and 10 million points. We list the exact running times in Table 11

.

gorithm to generate a dendrogram from the MST of the HDBSCAN* or EMST problem, which solves the single-linkage clustering problem. We show our algorithms achieve massive parallel speedup over the state-of-the-art. We describe these algorithms in Section 5.

3. We describe a parallel grid clustering framework with high-level programming constructs for efficiently implementing various grid-based clustering algorithms. We describe the framework in Section 6.

4. We describe contributions to three problems in computational geometry. We introduce a new parallel algorithm that can express both the randomized incremental convex hull algorithm and the quickhull algorithm, propose a sampling-based algorithm to reduce work for the smallest enclosing ball problem, and design a parallel batch-dynamic data structure for dynamic closest pair. We also introduce a new parallel batch-dynamic binary heap that may be of independent interest. We describe these algorithms in Section 8.

5. We describe the ParGeo library for parallel computational geometry. This library provides a range of parallel algorithms for geometric problems and data structures, including $k$d-trees, $k$-NN search, range search, well-separated pair decomposition, Euclidean minimum spanning tree, spatial sorting, and geometric clustering. Additionally, ParGeo includes a collection of geometric graph generators. We describe ParGeo in Section 9.

**Thesis statement** Parallel shared-memory multi-core algorithms, implementations, and frameworks can efficiently process large-scale spatial clustering and computational geometry problems both in theory and practice.

This thesis is the result of a collaborative effort among the following papers:

- Yiqiu Wang, Yan Gu, and Julian Shun. Theoretically-efficient and practical parallel DBSCAN. In *ACM SIGMOD International Conference on Management of Data*, 2020. ([239], Section 4)

- 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. ([244], Section 5)

- Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. A parallel batch-dynamic data structure for the closest pair problem. In *37th International Symposium on Computational Geometry (SoCG 2021)*, 2021. ([243], Section 8)

- Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. GeoGraph: A framework for graph processing on geometric data. In *SIGOPS Oper. Syst. Rev*, 2021. ([241], Section 9),

- Yiqiu Wang, Rahul Yesantharao, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. Pargeo: a library for parallel computational geometry. In *30th Annual European Symposium on Algorithms (ESA 2022)*, 2022. ([240, 242], Section 8, 9)

# 2 Preliminaries and Notation

## 2.1 Computational Model

We use the work-depth model [138, 79] to analyze the theoretical efficiency of parallel algorithms. The ***work*** of an algorithm is the number of operations used, similar to the time complexity in the sequential RAM model. The ***depth*** is the length of the longest sequence of dependence. By Brent's scheduling theorem [55], an algorithm with work $W$ and depth $D$ has overall running time $W/P+D$, where $P$ is the number of processors available. In practice, the Cilk work-stealing scheduler [48] can be used to obtain an expected running time of $W/P + O(D)$. A parallel algorithm is ***work-efficient*** if its work asymptotically matches that of the best sequential algorithm for the problem, which is important since in practice the $W/P$ term in the running time often dominates.

## 2.2 Parallel Primitives

We give an overview of the primitives used in our new parallel algorithms in this thesis.

***Prefix sum*** takes as input an array $A$ of length $n$, and returns the array $(0, A[0], A[0]+A[1], \ldots, \sum_{i=0}^{n-2} A[i])$ as well as the overall sum, $\sum_{i=0}^{n-1} A[i]$. Prefix sum can be implemented by first adding the odd-indexed elements to the even-indexed elements in parallel, recursively computing the prefix sum for the even-indexed elements, and finally using the results on the even-indexed elements to update the odd-indexed elements in parallel. This algorithm takes $O(n)$ work and $O(\log n)$ depth [138].

***Filter*** takes an array $A$ of size $n$ and a predicate $f$, and returns a new array $A'$ containing elements $A[i]$ for which $f(A[i])$ is true, in the same order as in $A$. We first construct an array $P$ of size $n$ with $P[i] = 1$ if $f(A[i])$ is true and $P[i] = 0$ otherwise. Then we compute the prefix sum of $P$. Finally, for each element $A[i]$ where $f(A[i])$ is true, we write it to the output array $A'$ at index $P[i]$ (i.e., $A'[P[i]] = A[i]$). This algorithm also takes $O(n)$ work and $O(\log n)$ depth [138].

***Comparison sorting*** sorts $n$ elements based on a comparison function. Parallel comparison sorting can be done in $O(n \log n)$ work and $O(\log n)$ depth [138, 75]. We use a cache-efficient samplesort [44] from PBBS which samples $\sqrt{n}$ pivots on each

| | Work | Depth | Reference |
|---|---|---|---|
| Prefix sum, Filter | $O(n)$ | $O(\log n)$ | [138] |
| Comparison sort | $O(n \log n)$ | $O(\log n)$ | [138, 75] |
| Integer sort | $O(n)$ | $O(\log n)$ | [234] |
| Semisort | $O(n)^\dagger$ | $O(\log n)^*$ | [115] |
| Merge | $O(n)$ | $O(\log n)$ | [138] |
| Hash table | $O(n)^*$ | $O(\log n)^*$ | [109] |
| 2D Delaunay triangulation | $O(n \log n)^*$ | $O(\log n)^*$ | [198] |

Table 1: Work and depth bounds for parallel primitives. $^\dagger$ indicates an expected bound and $^*$ indicates a high-probability bound. The integer sort is for a polylogarithmic key range. The cost of the hash table is for $n$ insertions or queries.

level of recursion, partitions the keys based on the pivots, and recurses on each partition in parallel.

***Integer sorting*** sorts integer keys from a polylogarithmic range in $O(n)$ work and $O(\log n)$ depth [234]. The algorithm partitions the keys into sub-arrays and in parallel across all partitions, builds a histogram on each partition serially. It then uses a prefix sum on the counts of each key per partition to determine unique offsets into a global array for each partition. Finally, all partitions write their keys into unique positions in the global array in parallel.

***Semisort*** takes as input $n$ key-value pairs, and groups pairs with the same key together, but with no guarantee on the relative ordering among pairs with different keys. Semisort also returns the number of distinct groups. We use the implementation from [115], which is available in PBBS. The algorithm first hashes the keys, and then selects a sample of the keys to predict the frequency of each key. Based on the frequency of keys in the sample, we classify them into "heavy keys" and "light keys", and assign appropriately-sized arrays for each heavy key and each range of light keys. Finally, we insert all keys into random locations in the appropriate array and sort within the array. This algorithm takes $O(n)$ expected work and $O(\log n)$ depth with high probability.[4]

***Merge*** takes two sorted arrays, $A$ and $B$, and merges them into a single sorted array. If the sum of the lengths of the inputs is $n$, this can be done in $O(n)$ work and $O(\log n)$ depth [138]. The algorithm takes equally spaced pivots $A$ and does a binary search for each pivot in $B$. Each sub-array between pivots in $A$ has a

[4]We say that a bound holds ***with high probability (w.h.p.)*** on an input of size $n$ if it holds with probability at least $1 - 1/n^c$ for a constant $c > 0$.

corresponding sub-array between the binary search results in $B$. Then it repeats the above process for each pair, except that equally spaced pivots are taken from the sub-array from $B$ and binary searches are done in the sub-array from $A$. This creates small subproblems each of which can be solved using a serial merge, and the results are written to a unique range of indices in the final output. All subproblems can be processed in parallel. It can be implemented in $O(n)$ work and $O(\log n)$ depth [138].

***Split*** takes an array $A$ and a predicate function $f$, and moves all of the "true" elements before the "false" elements. Split can be implemented using filter. It can be implemented in $O(n)$ work and $O(\log n)$ depth [138].

***List ranking*** takes a linked list with values on each node and returns for each node the sum of values from the node to the end of the list. It can be implemented in $O(n)$ work and $O(\log n)$ depth [138].

The ***Euler tour*** of a tree takes as input an adjacency list representation of the tree and returns a directed circuit that traverses every edge of the tree exactly once. It can be implemented in $O(n)$ work and $O(\log n)$ depth [138].

The ***minimum*** algorithm computes the minimum of $n$ points in $O(n)$ expected work and $O(1)$ depth *whp* [234].

For ***parallel hash tables***, we can perform $n$ insertions or queries taking $O(n)$ work and $O(\log n)$ depth w.h.p. [109]. We use the non-deterministic concurrent linear probing hash table from [211], which uses an atomic update to insert an element to an empty location in its probe sequence, and continues probing if the update fails. For some of the results, we use ***parallel dictionaries***, which support $n$ insertions, deletions, or lookups in $O(n)$ work and $O(\log^* n)$ depth *whp* [109].

The ***Delaunay triangulation*** on a set of points in 2D contains triangles among every triple of points $p_1$, $p_2$, and $p_3$ such that there are no other points inside the circumcircle defined by $p_1$, $p_2$, and $p_3$ [84]. Delaunay triangulation can be computed in parallel in $O(n \log n)$ work and $O(\log n)$ depth w.h.p. [198]. We use the randomized incremental algorithm from PBBS, which inserts points in parallel into the triangulation in rounds, such that the updates to the triangulation in each round by different points do not conflict [43].

***WriteMin*** is a priority concurrent write that takes as input two arguments, where the first argument is the location to write to and the second argument is the value to write; on concurrent writes, the smallest value is written [214].

## 2.3 Relevant Techniques

**$k$-NN Query** A ***$k$-nearest neighbor ($k$-NN) query*** takes a point data set $\mathcal{P}$ and a distance function, and returns for each point in $\mathcal{P}$ its $k$ nearest neighbors

(including itself). Callahan and Kosaraju [57] show that $k$-NN queries in Euclidean space for all points can be solved in parallel in $O(kn \log n)$ work and $O(\log n)$ depth.

**$k$d-tree** A ***$k$d-tree*** is a commonly used data structure for $k$-NN queries [100]. It is a binary tree that is constructed recursively: each node in the tree represents a set of points, which are partitioned between its two children by splitting along one of the dimensions; this process is recursively applied on each of its two children until a leaf node is reached (a leaf node is one that contains at most $c$ points, for a predetermined constant $c$). It can be constructed in parallel by processing each child in parallel. A $k$-NN query can be answered by traversing nodes in the tree that are close to the input point, and pruning nodes further away that cannot possibly contain the $k$ nearest neighbors.

**BCCP and BCCP*** Existing algorithms, as well as some of our new algorithms, use subroutines for solving the ***bichromatic closest pair (BCCP)*** problem, which takes as input two sets of points, $A$ and $B$, and returns the pair of points $p_1$ and $p_2$ with minimum distance between them, where $p_1 \in A$ and $p_2 \in B$. We also define a variant, the ***BCCP**** problem, that finds the pair of points with the minimum mutual reachability distance, as defined for HDBSCAN*.

**Well-Separated Pair Decomposition** We use the same definitions and notations as in Callahan and Kosaraju [60]. Two sets of points, $A$ and $B$, are ***well-separated*** if $A$ and $B$ can each be contained in spheres of radius $r$, and the minimum distance between the two spheres is at least $sr$, for a ***separation constant*** $s$ (we use $s = 2$ throughout the thesis). An ***interaction product*** of point sets $A$ and $B$ is defined to be $A \otimes B = \{\{p, p'\} |\ p \in A,\ p' \in B,\ p \neq p'\}$. The set $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ is a ***well-separated realization*** of $A \otimes B$ if: **(1)** $A_i \subseteq A$ and $B_i \subseteq B$ for all $i = 1, ..., k$; **(2)** $A_i \cap B_i = \emptyset$ for all $i = 1, ..., k$; **(3)** $(A_i \otimes B_i) \bigcap (A_j \otimes B_j) = \emptyset$ for all $i, j$ where $1 \leq i < j \leq k$; **(4)** $A \otimes B = \bigcup_{i=1}^{k} A_i \otimes B_i$; **(5)** $A_i$ and $B_i$ are well-separated for all $i = 1, ..., k$.

For a point set $\mathcal{P}$, a ***well-separated pair decomposition (WSPD)*** is a well-separated realization of $\mathcal{P} \otimes \mathcal{P}$. We discuss how to construct a WSPD using a $k$d-tree in Section 5.2.

# Part II

# Parallel Spatial Clustering

# 3 Introduction

Cluster analysis, also known as clustering, is a common method of data analysis used in various fields such as pattern recognition, image analysis, bioinformatics, and machine learning. It involves grouping similar objects into clusters. One important type of clustering algorithm is density-based clustering, where clusters are identified based on contiguous regions of high-density points separated by low-density ones. One of the most widely-used density-based spatial clustering methods is DBSCAN, which was developed by Ester et al. [95]. It can find good clusters of different shapes in the presence of noise without requiring prior knowledge of the number of clusters. DBSCAN has been applied successfully in various domains such as transportation, biology, and astronomy. However, DBSCAN requires two parameters, $\epsilon$ and minPts, which determine what is considered "close" and "dense", respectively. In practice, many different values of $\epsilon$ need to be explored in order to find high-quality clusters.

To avoid repeatedly executing DBSCAN for different values of $\epsilon$, the OPTICS [22] and HDBSCAN* [61] algorithms have been proposed for constructing DBSCAN clustering hierarchies. These algorithms generate a minimum spanning tree on the input points based on a DBSCAN-specific metric known as the core distance, which is used to determine the density of the points. HDBSCAN* is a popular hierarchical clustering algorithm that is known to be robust to outliers in the data set.

Density-based clustering has numerous applications in data science, spatial databases and unsupervised learning. There have been numerous density-based clustering algorithms proposed over the years, such as DENCLUE [127], STING [238], and CHAMELEON [147]. In this thesis, we focus on DBSCAN and HDBSCAN*, describing our parallel algorithms for these problems in Sections 4 and 5 respectively, as well as for generating the dendrogram in the case of HDBSCAN*.

In this thesis, we introduce new parallel algorithms for exact and approximate DBSCAN, as well as for HDBSCAN* and EMST, with significant speedup over existing methods. These algorithms maintain low depth while matching the work complexity of the best sequential counterparts. Additionally, we propose a dendrogram generation algorithm to solve the single-linkage clustering problem. In Sections 4 and 5, we perform extensive experiments demonstrating the efficiency of these algorithms.

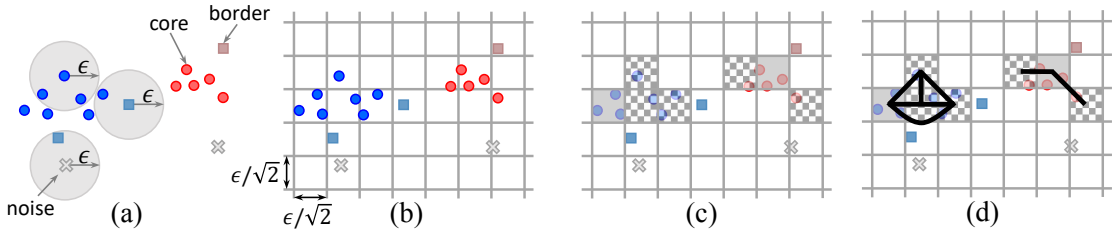



Figure 3: An example of DBSCAN and basic concepts in two dimensions. Here we set minPts $= 3$ and $\epsilon$ as drawn. In (a), the points are categorized into core points (circles) in two clusters (red and blue), border points (squares) that belong to the clusters, and noise points (crosses). Using the grid method for cell construction, the algorithm constructs cells with side length $\epsilon/\sqrt{2}$ (diagonal length $\epsilon$), as shown in (b). The cells with at least minPts points are marked as core cells (solid gray cells in (c)), while points in other cells try to check if they have minPts points within a distance of $\epsilon$. If so, the associated cells are marked as core cells as well (checkered cells in (c)). To construct the cell graph, we create an edge between two core cells if the closest pair of points from the two cells is within a distance of $\epsilon$ (shown in (d)). Each connected component in the cell graph is a unique cluster. Border points are assigned to clusters that they are within $\epsilon$ distance from.

## 3.1 Problem Definitions

### DBSCAN

The ***DBSCAN*** (density-based spatial clustering of applications with noise) problem takes as input $n$ points $\mathcal{P} = \{p_0, \ldots, p_{n-1}\}$, a distance function $d$, and two parameters $\epsilon$ and minPts [95]. A point $p$ is a ***core point*** if and only if $|\{p_i \mid p_i \in \mathcal{P}, d(p, p_i) \leq \epsilon\}| \geq$ minPts. We denote the set of core points as $\mathcal{C}$. DBSCAN computes and outputs subsets of $\mathcal{P}$, referred to as ***clusters***. Each point in $\mathcal{C}$ is in exactly one cluster, and two points $p, q \in \mathcal{C}$ are in the same cluster if and only if there exists a list of points $\bar{p}_1 = p, \bar{p}_2, \ldots, \bar{p}_{k-1}, \bar{p}_k = q$ in $\mathcal{C}$ such that $d(\bar{p}_{i-1}, \bar{p}_i) \leq \epsilon$. For all non-core points $p \in \mathcal{P} \setminus \mathcal{C}$, $p$ belongs to cluster $C_i$ if $d(p, q) \leq \epsilon$ for at least one point $q \in \mathcal{C} \cap C_i$. Note that a non-core point can belong to multiple clusters. A non-core point belonging to at least one cluster is called a ***border point*** and a non-core point belonging to no clusters is called a ***noise point***. For a given set of points and parameters $\epsilon$ and minPts, the clusters returned are unique. Similar to many previous papers on parallel DBSCAN, we focus on the Euclidean distance metric in this thesis. See Figure 3(a) for an illustration of the DBSCAN problem.

Gan and Tao [104] define the ***approximate DBSCAN*** problem, which in addition to the DBSCAN inputs, takes a parameter $\rho$. The definition is the same as DBSCAN, except for the connectivity rule among core points. In particular, core points within a distance of $\epsilon$ are still connected, but core points within a distance of $(\epsilon, \epsilon(1+\rho)]$ may or may not be connected. Core points with distance greater than $\epsilon(1+\rho)$ are still not connected. Due to this relaxation, multiple valid clusterings can be returned. The relaxation is what enables an asymptotically faster algorithm to be designed. A variation of this problem was described by Chen et al. [70].

Existing algorithms as well as some of our new algorithms use subroutines for solving the ***bichromatic closest pair (BCP)*** problem, which takes as input two sets of points $P_1$ and $P_2$, finds the closest pair of points $p_1$ and $p_2$ such that $p_1 \in P_1$ and $p_2 \in P_2$, and returns the pair and their distance.

### HDBSCAN*

**EMST** The ***Euclidean Minimum Spanning Tree (EMST)*** problem takes $n$ points $\mathcal{P} = \{p_1, \ldots, p_n\}$ and returns a minimum spanning tree (MST) of the complete undirected Euclidean graph of $\mathcal{P}$.

**DBSCAN*** The ***DBSCAN**** (density-based spatial clustering of applications with noise) problem takes as input $n$ points $\mathcal{P} = \{p_1, \ldots, p_n\}$, a distance function $d$, and two parameters $\epsilon$ and $\mathsf{minPts}$ [95, 61]. A point $p$ is a ***core point*** if and only if $|\{p_i \mid p_i \in \mathcal{P}, d(p, p_i) \leq \epsilon\}| \geq \mathsf{minPts}$. A point is called a ***noise point*** otherwise. We denote the set of core points as $\mathcal{P}_{core}$. DBSCAN* computes a partition of $\mathcal{P}_{core}$, where each subset is referred to as a ***cluster***, and also returns the remaining points as noise points. Two points $p, q \in \mathcal{P}_{core}$ are in the same cluster if and only if there exists a list of points $p = \bar{p}_1, \bar{p}_2, \ldots, \bar{p}_{k-1}, \bar{p}_k = q$ in $\mathcal{P}_{core}$ such that $d(\bar{p}_{i-1}, \bar{p}_i) \leq \epsilon$ for all $1 < i \leq k$. For a given set of points and two parameters $\epsilon$ and $\mathsf{minPts}$, the clusters returned are unique.[5]

**HDBSCAN*** The ***HDBSCAN**** (hierarchical DBSCAN*) problem [61] takes the same input as DBSCAN*, but without the $\epsilon$ parameter, and computes a hierarchy of DBSCAN* clusters for all possible values of $\epsilon$. The ***core distance*** of a point $p$, $\mathsf{cd}(p)$, is the distance from $p$ to its $\mathsf{minPts}$-nearest neighbor (including $p$ itself). The ***mutual reachability distance*** between two points $p$ and $q$ is defined to be

[5]The original DBSCAN definition includes the notion of border points, which are non-core points that are within a distance of $\epsilon$ to core points [95]. DBSCAN* chooses to omit this to be more consistent with a statistical interpretation of clusters [61].

$d_m(p,q) = max\{\mathsf{cd}(p), \mathsf{cd}(q), d(p,q)\}$. The ***mutual reachability graph*** $G_{MR}$ is a complete undirected graph, where the vertices are the points in $\mathcal{P}$, and the edges are weighted by the mutual reachability distances.[6]

The HDBSCAN*hierarchy is sequentially computed in two steps [61]. The first step computes an MST of $G_{MR}$ and then adds a self-edge to each vertex weighted by its core distance. An example MST is shown in Figure 4a. We note that the HDBSCAN*MST with $\mathsf{minPts} = 1$ is equivalent to the EMST, since the mutual reachability distance at $\mathsf{minPts} = 1$ is equivalent to the Euclidean distance. We further elaborate on the relationship between HDBSCAN*and EMST in Section 5.7. A dendrogram representing clusters at different values of $\epsilon$ is computed by removing edges from the MST plus self-edges graph in decreasing order of weight. The root of the dendrogram is a cluster containing all points. Each non-self-edge removal splits a cluster into two, which become the two children of the cluster in the dendrogram. The height of the split cluster in the dendrogram is equal to the weight of the removed edge. If the removed edge is a self-edge, we mark the component (point) as a noise point. An example of a dendrogram is shown in Figure 4b. If we want to return the clusters for a particular value of $\epsilon$, we can horizontally cut the dendrogram at that value of $\epsilon$ and return the resulting subtrees below the cut as the clusters or noise points. This is equivalent to removing edges from the MST of $G_{MR}$ with weight greater than $\epsilon$.

For HDBSCAN*, the reachability plot (OPTICS sequence) [22] contains all points in $\mathcal{P}$ in some order $\{p_i |\ i = 1, \ldots, n\}$, where each point $p_i$ is represented as a bar with height $\min\{d_m(p_i, p_j) \mid j < i\}$. For HDBSCAN*, the order of the points is the order that they are visited in an execution of Prim's algorithm on the MST of $G_{MR}$ starting from an arbitrary point [22]. An example is shown in Figure 4c. Intuitively, the "valleys" of the reachability plot correspond to clusters [61].

**Relationship between HDBSCAN* and EMST** We note that the HDBSCAN* MST with $\mathsf{minPts} = 1$ is equivalent to the Euclidean MST (EMST), since the mutual reachability distance at $\mathsf{minPts} = 1$ is equivalent to the Euclidean distance. The EMST problems take as input a set of $n$ points in a $d$-dimensional space. EMST computes a minimum spanning tree on a complete graph formed among the $n$ points with edges between two points having the weight equal to their Euclidean distance. EMST has many applications, including in single-linkage clustering [113], network placement optimization [237], and approximating the Euclidean traveling salesman problem [231].

[6]The related OPTICS problem also generates a hierarchy of clusters but with a definition of reachability distance that is asymmetric, leading to a directed graph [22].

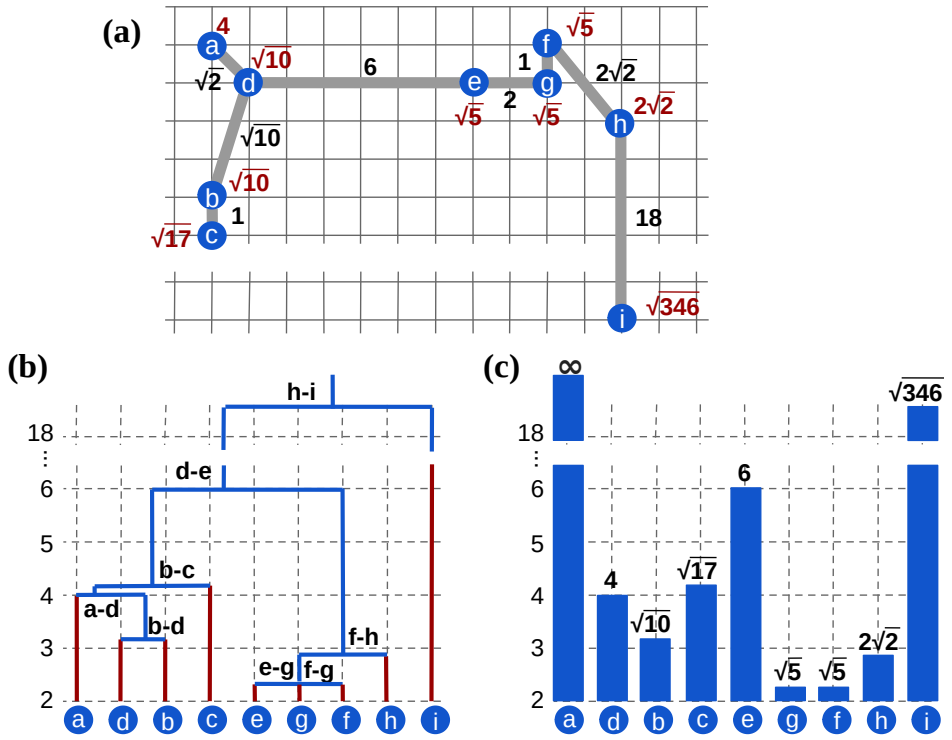



Figure 4: (a) An MST of the HDBSCAN*mutual reachability graph on an example data set in 2D. The red number next to each point is the core distance of the point for minPts $= 3$. The Euclidean distances between points are denoted by grey edges, whose values are marked in black. For example, $a$'s core distance is 4 because $b$ is $a$'s third nearest neighbor (including itself) and $d(a, b) = 4$. The edge weight of $(a, d)$ is $\max\{4, \sqrt{10}, \sqrt{2}\} = 4$. (b) An HDBSCAN*dendrogram for the data set. A point becomes a noise point when its vertical line becomes red. For example, if we cut the dendrogram at $\epsilon = 3.5$, then we have two clusters $\{d, b\}$ and $\{e, g, f, h\}$, while $a$, $c$ and $i$ are noise points. (c) A reachability plot for the data set starting at point $a$. The two "valleys", $\{a, b, c, d\}$ and $\{e, f, g, h\}$, are the two most obvious clusters.

| Notation | Definition |
|---|---|
| $d(p, q)$ | Euclidean distance between points $p$ and $q$. |
| $d_m(p, q)$ | Mutual reachability distance between points $p$ and $q$. |
| $d(A, B)$ | Minimum distance between the bounding spheres of points in tree node $A$ and points in tree node $B$. |
| $w(u, v)$ | Weight of edge $(u, v)$. |
| $A_{\mathsf{diam}}$ | Diameter of the bounding sphere of points in tree node $A$. |
| $\mathsf{cd}_{\min}(A)$ | Minimum core distance of points in tree node $A$. |
| $\mathsf{cd}_{\max}(A)$ | Maximum core distance of points in tree node $A$. |

Table 2: Summary of Notation

**Notation** Table 2 shows notation frequently used in this part of the thesis.

# 4 Theoretically Efficient and Practical Parallel DBSCAN

## 4.1 Introduction

The traditional DBSCAN algorithm [95] and their variants require work quadratic in the input size in the worst case, which can be prohibitive for the large data sets that need to be analyzed today. To address this computational bottleneck, there has been recent work on designing parallel algorithms for DBSCAN and its variants [54, 251, 24, 141, 140, 49, 250, 133, 77, 183, 184, 186, 185, 245, 161, 102, 83, 21, 121, 150, 125, 102, 69, 112, 78, 162, 257, 132, 23, 131, 218]. However, even though these solutions achieve scalability and speedup over sequential algorithms, in the worst-case their number of operations still scale quadratically with the input size. Therefore, a natural question is whether there exist DBSCAN algorithms that are faster both in theory and practice, and in both the sequential and parallel settings.

Given the ubiquity of datasets in Euclidean space, there has been work on faster sequential DBSCAN algorithms in this setting. Gunawan [117] and de Berg et al. [85] has shown that Euclidean DBSCAN in 2D can be solved sequentially in $O(n \log n)$ work. Gan and Tao [104] provide alternative Euclidean DBSCAN algorithms for two-dimensions that take $O(n \log n)$ work. For higher dimensions, Chen et al. [70] provide an algorithm that takes $O(n^{2(1-1/(d+2))}\text{polylog}(n))$ work for $d$ dimensions, and Gan and Tao [104] improve the result with an algorithm that takes $O(n^{2-(2/(\lceil d/2 \rceil+1))+\delta})$ work for any constant $\delta > 0$. To further reduce the work complexity, there have been approximate DBSCAN algorithms proposed. Chen et al. [70] provide an approximate DBSCAN algorithm that takes $O(n \log n)$ work for any constant number of dimensions, and Gan and Tao [104] provide a similar algorithm taking $O(n)$ expected work. However, none of the algorithms described above have been parallelized.

This thesis bridges the gap between theory and practice in parallel Euclidean DBSCAN by providing new parallel algorithms for exact and approximate DBSCAN with work complexity matching that of best sequential algorithms [117, 85, 104], and with low depth, which is the gold standard in parallel algorithm design. For exact 2D DBSCAN, we design several parallel algorithms that use either the box or the grid construction for partitioning points [117, 85] and one of the following three procedures for determining connectivity among core points: Delaunay triangulation [104], unit-spherical emptiness checking with line separation [104], and bichromatic closest

pairs. For higher-dimensional exact DBSCAN, we provide an algorithm based on solving the higher-dimensional bichromatic closest pairs problem in parallel. Unlike many existing parallel algorithms, our exact algorithms produce the same results according to the standard definition of DBSCAN, and so we do not sacrifice clustering quality. For approximate DBSCAN, we design an algorithm that uses parallel quadtree construction and querying. Our approximate algorithm returns the same result as the sequential approximate algorithm by Gan and Tao [104].

We perform a comprehensive set of experiments on synthetic and real-world datasets using varying parameters, and compare our performance to optimized sequential implementations as well as existing parallel DBSCAN algorithms. On a 36-core machine with two-way hyper-threading, our exact DBSCAN implementations achieve 2–89x (24x on average) self-relative speedup and 5–33x (16x on average) speedup over the fastest sequential implementations. Our approximate DBSCAN implementations achieve 14–44x (24x on average) self-relative speedup. Compared to existing parallel algorithms, which are scalable but have high overheads compared to serial implementations, our fastest exact algorithms are faster by up to orders of magnitude (16–6102x) under correctly chosen parameters. Our algorithms can process the largest dataset that has been used in the literature for exact DBSCAN, and outperform the state-of-the-art distributed RP-DBSCAN algorithm [218] by 18–577x. We have made our source code publicly available at https://github.com/wangyiqiu/dbscan.

### Related Work

Xu et al. [251] provide the first parallel exact DBSCAN algorithm, called PDBSCAN, based on a distributed $R^*$-tree. Arlia and Coppola [24] present a parallel DBSCAN implementation that replicates a sequential $R^*$-tree across machines to process points in parallel. Coppola and Vanneschi [77] design a parallel algorithm using a queue to store core points, where each core point is processed one at a time but their neighbors are checked in parallel to see whether they should be placed at the end of the queue. Januzaj et al. [140, 141] design an approximate DBSCAN algorithm based on determining representative points on different local processors, and then running a sequential DBSCAN on the representatives. Brecheisen et al. [54] parallelize a version of DBSCAN optimized for complex distance functions [53].

Patwary et al. [183] present PDSDBSCAN, a multicore and distributed algorithm for DBSCAN using a union-find data structure for connecting points. Their union-find data structure is lock-based whereas ours is lock-free. Patwary et al. [186, 185] also present distributed DBSCAN algorithms that are approximate but more scalable

than PDSDBSCAN. Hu et al. [132] design PS-DBSCAN, an implementation of DBSCAN using a parameter server framework. Gotz et al. [112] present HPDBSCAN, an algorithm for both shared-memory and distributed-memory based on partitioning the data among processors, running DBSCAN locally on each partition, and then merging the clusters together. Very recently, Sarma et al. [204] present a distributed algorithm, $\mu$DBSCAN, and report a running time of 41 minutes for clustering one billion 3-dimensional points using a cluster of 32 nodes. Our running times on the larger 13-dimensional TeraClickLog dataset are significantly faster (under 30 seconds on 48 cores).

Exact and approximate distributed DBSCAN algorithms have been designed using MapReduce [250, 83, 125, 102, 150, 257, 23, 131] and Spark [78, 121, 162, 133, 218, 161]. RP-DBSCAN [218], an approximate DBSCAN algorithm, has been shown to be the state-of-the-art for MapReduce and Spark. GPU implementations of DBSCAN have also been designed [49, 21, 245, 69].

In addition to parallel solutions, there have been optimizations proposed to speed up sequential DBSCAN [53, 152, 165]. DBSCAN has also been generalized to other definitions of neighborhoods [201]. Furthermore, there have been variants of DBSCAN proposed in the literature, which do not return the same result as the standard DBSCAN. IDBSCAN [50], FDBSCAN [159], GF-DBSCAN [227], I-DBSCAN [236], GNDBSCAN [134], Rough-DBSCAN [235], and DBSCAN++ [139] use sampling to reduce the number of range queries needed. El-Sonbaty et al. [93] presents a variation that partitions the dataset, runs DBSCAN within each partition, and merges together dense regions. GriDBSCAN [167] uses a similar idea with an improved scheme for partitioning and merging. Other partitioning based algorithms include PACA-DBSCAN [143], APSCAN [72], and AA-DBSCAN [149]. DBSCAN* and H-DBSCAN* are variants of DBSCAN where only core points are included in clusters [61]. Other variants use approximate neighbor queries to speed up DBSCAN [249, 124].

OPTICS [22], SUBCLU [144], and GRIDBSCAN [230], are hierarchical versions of DBSCAN that compute DBSCAN clusters on different parameters, enabling clusters of different densities to more easily be found. POPTICS [184] is a parallel version of OPTICS based on concurrent union-find.

## 4.2 DBSCAN Algorithm Overview

This section reviews the high-level structure of existing sequential DBSCAN algorithms [117, 104, 85] as well as our new parallel algorithms. The high-level structure is shown in Algorithm 1, and an illustration of the key concepts are shown in Fig-

**Algorithm 1** DBSCAN Algorithm

**Input**: A set $\mathcal{P}$ of points, $\epsilon$, and minPts
**Output**: An array *clusters* of sets of cluster IDs for each point

1: **procedure** DBSCAN($\mathcal{P}$, $\epsilon$, minPts)
2: $\mathcal{G}$ := Cells($\mathcal{P}$, $\epsilon$)
3: *coreFlags* := MarkCore($\mathcal{P}, \mathcal{G}, \epsilon$, minPts)
4: *clusters* := ClusterCore($\mathcal{P}, \mathcal{G}$, *coreFlags*, $\epsilon$, minPts)
5: ClusterBorder($\mathcal{P}, \mathcal{G}$, *coreFlags*, *clusters*, $\epsilon$, minPts)
6: **return** *clusters*

ure 3(b)-(d).

We place the points into disjoint $d$-dimensional ***cells*** with side-length $\epsilon/\sqrt{d}$ based on their coordinates (Line 2 and Figure 3(b)). The cells have the property that all points inside a cell are within a distance of $\epsilon$ from each other, and will belong to the same cluster in the end. Then on Line 3 and Figure 3(c), we mark the core points. On Line 4, we generate the clusters for core points as follows. We create a graph containing one vertex per ***core cell*** (a cell containing at least one core point), and connect two vertices if the closest pair of core points from the two cells is within a distance of $\epsilon$. We refer to this graph as the ***cell graph***. This step is illustrated in Figure 3(d). We then find the connected components of the cell graph to assign cluster IDs to points in core cells. On Line 5, we assign cluster IDs for border points. Finally, we return the cluster labels on Line 6.

All of our algorithms share this common structure. In Section 4.2, we introduce our 2D algorithms, and in Section 4.3, we introduce our algorithms for higher dimensions. We analyze the complexity of our algorithms in Section 4.5.

### 2D DBSCAN Algorithms

This section presents our parallel algorithms for implementing each line of Algorithm 1 in two dimensions. The cells can be constructed either using a grid-based method or a box-based method, which we describe in Sections 4.2 and 4.2, respectively. Section 4.2 presents our algorithm for marking core points. We present several methods for constructing the cell graph in Section 4.2. Finally, Section 4.2 describes our algorithm for clustering border points.

### Grid Computation

In the grid-based method, the points are placed into disjoint cells with side-length $\epsilon/\sqrt{2}$ based on their coordinates, as done in the sequential algorithms by Gunawan [117] and de Berg et al. [85]. A hash table is used to store only the non-empty cells, and a serial algorithm simply inserts each point into the cell corresponding to its coordinates.

**Parallelization.** The challenge in parallelization is in distributing the points to the cells in parallel while maintaining work-efficiency. While a comparison sort could be used to sort points by their cell IDs, this approach requires $O(n \log n)$ work and is not work-efficient. We observe that semisort (see Section 2) can be used to solve this problem work-efficiently. The key insight here is that we only need to group together points in the same cell, and do not care about the relative ordering of points within a cell or between different cells. We apply a semisort on an array of length $n$ of key-value pairs, where each key is the cell ID of a point and the value is the ID of the point. This also returns the number of distinct groups (non-empty cells).

We then create a parallel hash table of size equal to the number of non-empty cells, where each entry stores the bounding box of a cell as the key, and the number of points in the cell and a pointer to the start of its points in the semisorted array as the value. We can determine neighboring cells of a cell $g$ with arithmetic computation based on $g$'s bounding box, and then look up each neighboring cell in the hash table, which returns the information for that cell if it is non-empty.

### Box Computation

In the box-based method, we place the points into disjoint 2-dimensional bounding boxes with side-length at most $\epsilon/\sqrt{d}$, which are the cells.

Existing sequential solutions [117, 85] first sort all points by $x$-coordinate, then scan through the points, grouping them into strips of width $\epsilon/\sqrt{2}$ and starting a new strip when a scanned point is further than distance $\epsilon/\sqrt{2}$ from the beginning of the strip. It then repeats this process per strip in the $y$-dimension to create cells of side-length at most $\epsilon/\sqrt{2}$. This step is shown in Figure 5(a). Pointers to neighboring cells are stored per cell. This is computed for all cells in each $x$-dimensional strip $s$ by merging $s$ with each of strips $s-2$, $s-1$, $s+1$, and $s+2$, as these are the only strips that can contain cells with points within distance $\epsilon$. For each merge, we compare the bounding boxes of the cells in increasing $y$-coordinate, linking any two cells that may possibly have points within $\epsilon$ distance.

**Parallelization.** We now describe the method for assigning points to strips, which

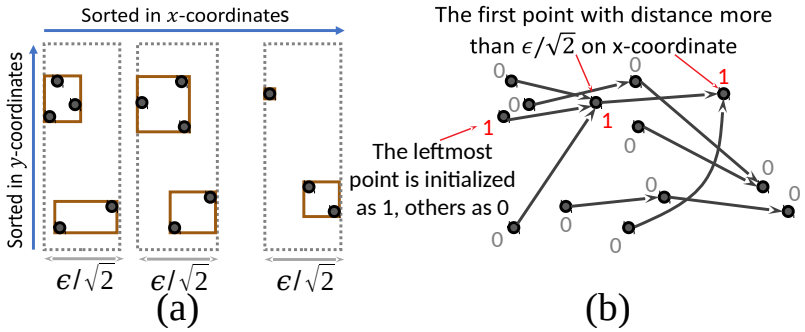



Figure 5: Parallel box method construction. In (a), the gray dashed rectangles correspond to strips and the brown solid rectangles correspond to box cells. To compute the strips, we create a pointer from each point to the first point with an $x$-coordinate that is more than $\epsilon/\sqrt{2}$ larger. We initialize the leftmost point with a value of 1 and all other points with a value of 0. As shown in (b), after running pointer jumping, the points at the beginning of strips have values of 1 and all other points have values of 0. We apply the same procedure in each strip on the $y$-coordinates to obtain the boxes.

is illustrated in Figure 5(b). Let $p_x$ be the $x$-coordinate of point $p$. We create a linked list where each point is a node. The node for point $p$ stores a pointer to the node for point $q$ (we call $q$ the ***parent*** of $p$), where $q$ is the point with the smallest $x$-coordinate such that $p_x + \epsilon/\sqrt{2} < q_x$. Each point can determine its parent in $O(\log n)$ work and depth by binary searching the sorted list of points.

We then assign a value of 1 to the node with the smallest $x$-coordinate, and 0 to all other nodes. We run a pointer jumping routine on the linked list where on each round, every node passes its value to its parent and updates its pointer to point to the parent of its parent [138]. The procedure terminates when no more pointers change in a round. In the end, every node with a value of 1 will correspond to the point at the beginning of a strip, and all nodes with a value of 0 will belong to the strip for the closest node to the left with a value of 1. This gives the same strips as the sequential algorithm, since all nodes marked 1 will correspond to the closest point farther than $\epsilon/\sqrt{2}$ from the point of the previously marked node. For merging to determine cells within distance $\epsilon$, we use the parallel merging algorithm described in Section 2.

**Algorithm 2** Parallel MarkCore

```
1: procedure MarkCore(P, G, ε, minPts)
2:   coreFlags := {0, ..., 0}                          ▷ Length |P| array
3:   par-for each g ∈ G do
4:     if |g| ≥ minPts then                  ▷ |g| is the number of points in g
5:       par-for each p in cell g do
6:         coreFlags[p] := 1
7:     else
8:       par-for each p in cell g do
9:         count := |g|
10:        for each h ∈ g.NeighborCells(ε) do
11:          count := count + RangeCount(p, ε, h)
12:        if count ≥ minPts then
13:          coreFlags[p] := 1
14:  return coreFlags
```

### Mark Core

Illustrated in Figure 3(c), the high-level idea in marking the core points is as follows: first, if a cell contains at least minPts points then all points in the cell are core points, as it is guaranteed that all the points inside a cell will be within $\epsilon$ to any other point in the same cell; otherwise, each point $p$ computes the number of points within its $\epsilon$-radius by checking its distance to points in all ***neighboring cells*** (defined as cells that could possibly contain points within a distance of $\epsilon$ to the current cell), and marking $p$ as a core point if the number of such points is at least minPts. For a constant dimension, only a constant number of neighboring cells need to be checked.

**Parallelization.** Our parallel algorithm for marking core points is shown in Algorithm 2. We create an array *coreFlags* of length $n$ that marks which points are core points. The array is initialized to all 0's (Line 2). We then loop through all cells in parallel (Line 3). If a cell contains at least minPts points, we mark all points in the cell as core points in parallel (Line 4–6). Otherwise, we loop through all points $p$ in the cell in parallel, and for each neighboring cell $h$ we count the number of points within a distance of $\epsilon$ to $p$, obtained using a RangeCount($p$, $\epsilon$, $h$) query (Lines 8–11) that reports the number of points in $h$ that are no more than $\epsilon$ distance from $p$. The RangeCount($p$, $\epsilon$, $h$) query can be implemented by comparing $p$ to all points in each neighboring cell $h$ in parallel, followed by a parallel prefix sum to obtain the number of points in the $\epsilon$-radius. If the total count is at least minPts, then $p$ is marked as a core point (Lines 12–13).

### Cluster Core

The next step of the algorithm is to generate the cell graph (illustrated in Figure 3(d)). We present three approaches for determining the connectivity between cells in the cell graph. After obtaining the cell graph, we run a parallel connected components algorithm to cluster the core points. For the BCP-based approach, we describe an optimization that merges the BCP computation with the connected components computation using a lock-free union-find data structure.

**BCP-based Cell Graph.** The problem of determining cell connectivity can be solved by computing the BCP of core points between two cells (recall the definition in Section 2), and checking whether the distance is at most $\epsilon$.

Each cell runs a BCP computation with each of its neighboring cells to check if they should be connected in the cell graph. We execute all BCP calls in parallel, and furthermore each BCP call can be implemented naively in parallel by computing all pairwise distances in parallel, writing them into an array containing point pairs and their distances, and applying a prefix sum on the array to obtain the BCP. We apply two optimizations to speed up individual BCP calls: (1) we first filter out points further than $\epsilon$ from the other cell beforehand as done by Gan and Tao [104], and (2) we iterate only until finding a pair of points with distance at most $\epsilon$, at which point we abort the rest of the BCP computation, and connect the two cells. Filtering points can be done using a parallel filter. To parallelize the early termination optimization, it is not efficient to simply parallelize across all the point comparisons as this will lead to a significant amount of wasted work. Instead, we divide the points in each cell into fixed-sized blocks, and iterate over all pairs of blocks. For each pair of blocks, we compute the distances of all pairs of points between the two blocks in parallel by writing their distances into an array. We then take the minimum distance in the array using a prefix sum, and return if the minimum is at most $\epsilon$. This approach reduces the wasted work over the naive parallelization, while still providing ample parallelism within each pair of blocks.

**Triangulation-based Cell Graph.** In two dimensions, Gunawan [117] describes a special approach using Voronoi diagrams. In particular, we can efficiently determine whether a core cell should be connected to a neighboring cell by finding the nearest core point from the neighboring cell to each of the core cell's core points. Gan and Tao [104] and de Berg et al. [85] show that a Delaunay triangulation can also be used to determine connectivity in the cell graph. In particular, if there is an edge in the Delaunay triangulation between two core cells with distance at most $\epsilon$, then those two cells are connected. This process is illustrated in Figure 6. The proof of correctness is described in [104, 85].

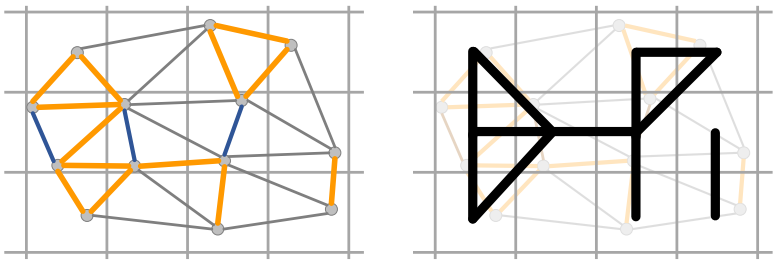


Figure 6: Using Delaunay triangulation (DT) to construct the cell graph in 2D. **(Left)** We construct the DT for all core points, and an edge in the DT can either be inside a cell (dark blue), or across cells with length no more than $\epsilon$ (orange), or with length more than $\epsilon$ (gray). **(Right)** An orange edge will add the associated edge in the cell graph, and in this example, there are two clusters.

To compute Delaunay triangulation or Voronoi diagram in parallel, Reif and Sen present a parallel algorithm for constructing Voronoi diagrams and Delaunay triangulations in two dimensions. We use the parallel Delaunay triangulation implementation from PBBS [43, 215], as described in Section 2.

**Unit-spherical emptiness checking-based (USEC) Cell Graph.** Gan and Tao [104] (who attribute the idea to Bose et al. [51]) describe an algorithm for solving the unit-spherical emptiness checking (USEC) with line separation problem when comparing two core cells to determine whether they should be connected in the cell graph.

In the USEC with line separation problem, we are given a horizontal or vertical line $\ell$ and would like to check if any of the $\epsilon$-radius circles of points on one side of $\ell$ contain any points on the other side of $\ell$. The problem assumes that the points on each side of $\ell$ are sorted by both $x$-coordinate and $y$-coordinate. This is illustrated in Figure 7.

To apply the USEC with line separation problem to DBSCAN, the points in each core cell are first sorted both by $x$-coordinate and by $y$-coordinate (two copies are stored). For each cell, we consider its top and left boundaries as $\ell$. For each $\ell$, we generate the union of $\epsilon$-radius circles centered around sorted points in the cell (sorted by $x$-coordinate for a horizontal boundary and $y$-coordinate for a vertical boundary) and keep the outermost arcs of the union of circles lying on the other side of $\ell$, which is called the wavefront. This is illustrated in Figure 7(b). For a cell connectivity query, we choose $\ell$ to be one of the boundaries of the two cells that separates the two cells. Then we scan the points of one cell in sorted order and check if any of them

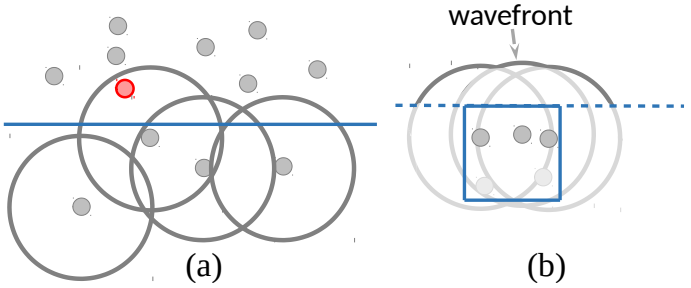



Figure 7: An example of the USEC with line separation problem. In (a), the points are above the horizontal line while the circles are centered below the line. In this case, the answer is "yes" since there is a point inside one of the circles. In (b), we show how this problem relates to DBSCAN. We generate the wavefront of the circles on the left and top borders of each cell, and check if core points in nearby cells are within the wavefront. In this example, we show the top wavefront.

are contained in the wavefront of the other cell.

Our algorithm assumes that all points are distinct. Without loss of generality, assume that we are generating a wavefront above a horizontal line. To generate the wavefront in parallel, we use divide-and-conquer by constructing the wavefront for the left half of the points and the right half of the points (in sorted order) recursively in parallel. Merging two wavefronts is more challenging. The wavefronts are represented as balanced binary trees supporting split and join operations [17]. We merge two wavefronts by taking the top part of each wavefront and joining them together. The top part of each wavefront can be obtained by checking where the left and right wavefronts intersect, and then merging them.

We first prove that the left and right wavefronts intersect at a unique point, assuming the points are distinct. Consider any three points $a$, $b$, and $c$ from the same cell, in order of $x$-coordinate. We consider the arcs formed by their $\epsilon$-radius circles, forming a wavefront on the top border of the cell. We prove that the arc of $c$ can intersect at most one location in the union of arcs formed by $a$ and $b$. Suppose that $c$'s arc intersects at least two locations in $b$'s arc. Take any two of these locations. They are part of the circle that forms $c$'s arc. This is a contradiction because it implies the circle forming $c$'s arc has larger radius than the circle forming $b$'s arc, as shown in Figure 8(a). Therefore $c$'s arc can intersect at most one location in $b$'s arc. A similar argument shows that $c$'s arc can intersect at most one location in $a$'s arc.

Now we need to prove that $c$'s arc cannot intersect both $a$'s arc and $b$'s arc.

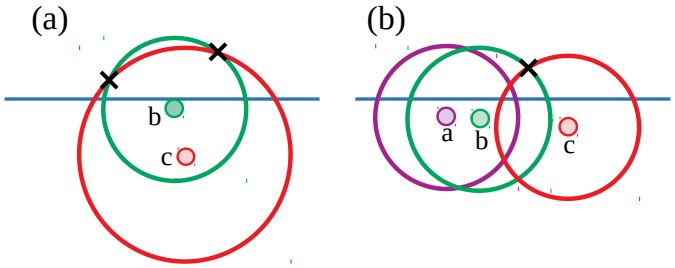



Figure 8: (a) shows that, in order for the circle centered at $c$ to intersect with the $\epsilon$-radius circle of $b$ at two points, the former must has a radius larger than $\epsilon$. (b) shows the circle of $c$'s intersection with that of $b$, and that the left half of $c$'s circle is below the union of the arcs formed by the circles of $a$ and $b$ above the horizontal line.

Without loss of generality, suppose that $c$'s arc intersects with $b$'s arc. Then, once $c$'s arc intersects with $b$'s arc, the rest of $c$'s arc must be in the interior of $b$'s arc, and thus below the union of $a$'s and $b$'s arcs above the horizontal line. The rest of $c$'s arc must be in the interior of $b$'s arc because $c$ is to the right of $b$, and their arcs are formed by circles of the same radius. This is shown in Figure 8(b).

Applying the above argument to all possible choices of $a$ and $b$ in the left wavefront, and all possible choices of $c$ in the right wavefront implies that there can only be one intersection between the two wavefronts.

Now denote the unique arc in the left wavefront that intersects with the right wavefront as $A$. All the arcs to the right of $A$ in the left wavefront lie under the right wavefront, so they will not form part of the combined wavefront. On the other hand, all arcs to the left of $A$ in the left wavefront forms the left half of the combined wavefront. We find arc $A$ by doing an exponential search in the left wavefront starting from the rightmost arc. For each arc $A'$ visited, we perform a binary search for $A'$ in the right wavefront to find its intersection with the right wavefront. There are three possible results: (a) $A'$ lies completely under the right wavefront (no intersection); (b) $A'$ intersects with the right wavefront; and (c) $A'$ lies completely outside the right wavefront (no intersection). If the result is (a), we continue the exponential search; if the result is (b), we terminate the search and join the two wavefronts at the intersection; and if the result is (c), arc $A$ must lie between $A'$ and the previous arc visited in the exponential search, and so we continue with a binary search inside that interval to find arc $A$. After the entire wavefront is generated, we write it out to an array by traversing the binary tree in parallel.

**Algorithm 3** Parallel ClusterCore

```
1: procedure ClusterCore(P, G, coreFlags, ε, minPts)
2:   uf := UnionFind()                                  ▷ Initialize union-find structure
3:   SortBySize(G)                                      ▷ Sort by non-increasing order of size
4:   par-for each {g ∈ G : g is core} do
5:     for each {h ∈ g.NeighborCells(ε) : h is core} do
6:       if g > h and uf.Find(g) ≠ uf.Find(h) then
7:         if Connected(g, h) then                      ▷ On core points only
8:           uf.Link(g, h)
9:   clusters := {−1, ..., −1}                          ▷ Length |P| array
10:  par-for each {g ∈ G : g is core} do
11:    par-for each {p in cell g : coreFlags[p] = 1} do
12:      clusters[p] := uf.Find(g)
13:  return clusters
```

We can perform a cell connectivity query in parallel by creating sub-problems using pivots, similar to how parallel merge is implemented (see Section 2). Recall that we are comparing the sorted points of one cell with the wavefront of the other cell. We take equally spaced arc intersections as pivots from the wavefront, and use binary search to find where the pivot fits in the sorted point set. Every set of arcs between two pivots corresponds to a set of sorted points between two binary search results. Then we repeat the procedure on each pair by taking equally spaced pivots in the sorted point set and doing binary search of the pivot in the set of arcs (which may split an arc into multiple pieces). This gives subproblems containing a contiguous subset of the sorted points and a contiguous subset of the (possibly split) arcs of the wavefront. Each subproblem is solved using the sequential USEC with line separation algorithm of [104]. If any of the subproblems return "yes", then the answer to the original USEC with line separation problem is "yes", and otherwise it is "no".

**Reducing Cell Connectivity Queries.** We now present an optimization that merges the cell graph construction with the connected components computation using a parallel lock-free union-find data structure to maintain the connected components on-the-fly. This technique is used in both the BCP approach and USEC approach for cell graph construction. The pseudocode is shown in Algorithm 3. The idea is to only run a cell connectivity query between two cells if they are not yet in the same component (Line 6), which can reduce the total number of connectivity queries. For example, assume that cells $a$, $b$, and $c$ belong to the same component. After connecting $a$ with $b$ and $b$ with $c$, we can avoid the connectivity check between $a$ and

**Algorithm 4** Parallel ClusterBorder

```
1: procedure ClusterBorder(P,G,coreFlags,clusters,ϵ,minPts)
2:   par-for each {g ∈ G : |g| < minPts} do
3:     par-for each {p in cell g : coreFlags[p] = 0} do
4:       for each h ∈ g ∪ g.NeighborCells(ϵ) do
5:         par-for each {q in cell h : coreFlags[q] = 1} do
6:           if d(p, q) ≤ ϵ then
7:             clusters[p] := clusters[p] ∪ clusters[q]          ▷ In parallel
```

$c$ by checking their respective components in the union-find structure beforehand. This optimization was used by Gan and Tao [104] in the sequential setting, and we extend it to the parallel setting. We also only check connectivity between two cells at most once by having the cell with higher ID responsible for checking connectivity with the cell with a lower ID (Line 6).

When constructing the cell graph and checking connectivity, we use a heuristic to prioritize the cells based on the number of core points in the cells, and start from the cells with more points, as shown on Line 3. This is because cells with more points are more likely to have higher connectivity, hence connecting the nearby cells together and pruning their connectivity queries. This optimization can be less efficient in parallel, since a connectivity query could be executed before the corresponding query that would have pruned it in the sequential execution. To overcome this, we group the cells into batches, and process each batch in parallel before moving to the next batch. We refer to this new approach as ***bucketing***, and show experimental results for it in Section 4.6.

### Cluster Border

To assign cluster IDs for border points. We check all points not yet assigned a cluster ID, and for each point $p$, we check all of its neighboring cells and add it to the clusters of all neighboring cells with a core point within distance $\epsilon$ to $p$.

**Parallelization.** Our algorithm is shown in Algorithm 4. We loop through all cells with fewer than minPts points in parallel, and for each such cell we loop over all of its non-core points $p$ in parallel (Lines 2–3). On Lines 4–7, we check all core points in the current cell $g$ and all neighboring cells, and if any are within distance $\epsilon$ to $p$, we add their clusters to $p$'s set of clusters (recall that border points can belong to multiple clusters).

## 4.3 Higher-dimensional Exact and Approximate DBSCAN

The efficient exact and approximate algorithms for higher-dimensional DBSCAN are also based on the high-level structure of Algorithm 1, and are extensions of some of the techniques for two-dimensional DBSCAN described in Section 4.2. They use the grid-based method for assigning points to cells (Section 4.2). Algorithms 2, 3, and 4 are used for marking core points, clustering core points, and clustering border points, respectively. However, we use two major optimizations on top of the 2D algorithms: a $k$-d tree for finding neighboring cells and a quadtree for answering range counting queries.

### Finding Neighboring Cells

The number of possible neighboring cells grows exponentially with the dimension $d$, and so enumerating all possible neighboring cells can be inefficient in practice for higher dimensions (although still constant work in theory). Therefore, instead of implementing NeighborCells by enumerating all possible neighboring cells, we first insert all cells into a $k$-d tree [38], which enables us to perform range queries to obtain just the non-empty neighboring cells. The construction of our $k$-d tree is done recursively, and all recursive calls for children nodes are executed in parallel. We also sort the points at each level in parallel and pass them to the appropriate child. Queries do not modify the $k$-d tree, and can all be performed in parallel. Since a cell needs to find its neighboring cells multiple times throughout the algorithm, we cache the result on its first query to avoid repeated computation.

## 4.4 Range Counting

While RangeCount queries can be implemented theoretically-efficiently in DBSCAN by checking all points in the target cell, there is a large overhead for doing so in practice. In higher-dimensional DBSCAN, we construct a quadtree data structure for each cell to answer RangeCount queries. The structure of a quadtree is illustrated in Figure 9. A cell of side-length $\epsilon/\sqrt{d}$ is recursively divided into $2^d$ sub-cells of the same size until the sub-cell becomes empty. This forms a tree where each sub-cell is a node and its children are the up to $2^d$ non-empty sub-cells that it divides into. Each node of the tree stores the number of points contained in its corresponding sub-cell. Queries do not modify the quadtrees and are therefore all executed in parallel. We now describe how to construct the quadtrees in parallel.

**Parallel Quadtree Construction.** The construction procedure recursively divides each cell into sub-cells. Each node of the tree has access to the points contained in

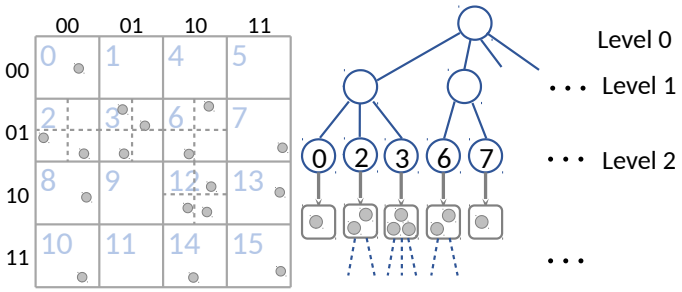



Figure 9: A cell (left) and its corresponding quadtree data structure (right).

its sub-cell in a contiguous subarray that is part of a global array (e.g., by storing a pointer to the start of its points in the global array as well as the number of points that it represents). We use an integer sort on keys from the range $[0, \ldots, 2^d - 1]$ to sort the points in the subarray based on which of the $2^d$ sub-cells it belongs to. Now the points belonging to each of the child nodes are contiguous, and we can recursively construct the up to $2^d$ non-empty child nodes independently in parallel by passing in the appropriate subarray.

To reduce construction time, we set a threshold for the number of points in a sub-cell, below which the node becomes a leaf node. This reduces the height of the tree but makes leaf nodes larger. In addition, we avoid unnecessary tree node traversal by ensuring that each tree node has at least two non-empty children: when processing a cell, we repeatedly divide the points until they fall into at least two different sub-cells.

**Range Counting in MarkCore.** RangeCount queries are used in marking core points in Algorithm 2. For each cell, a quadtree containing all of its points is constructed in parallel. Then the RangeCount($p$, $\epsilon$, $h$) query reports the number of points in cell $h$ that are no more than $\epsilon$ distance from point $p$. Instead of naively looping through all points in $h$, we initiate a traversal of the quadtree starting from cell $h$, and recursively search all children whose sub-cell intersects with the $\epsilon$-radius of $p$. When reaching a leaf node on a query, we explicitly count the number of points contained in the $\epsilon$-radius of the query point.

**Exact DBSCAN.** For higher-dimensional exact DBSCAN, one of our implementations uses RangeCount queries when computing BCPs in Algorithm 3. For each core cell, we build a quadtree on its core points in parallel. Then for each core point $p$ in each core cell $g$, we issue a RangeCount query to each of its neighboring core

cells $h$ and connect $g$ and $h$ in the cell graph if the range query returns a non-zero count of core points. Since we do not need to know the actual count, but only whether or not it is non-zero, our range query is optimized to terminate once such a result can be determined. We combine this with the optimization of reducing cell connectivity queries described in Section 4.2

**Approximate DBSCAN.** For approximate DBSCAN, the sequential algorithm of Gan and Tao [104] follows the high-level structure of Algorithm 1 using the grid-based cell structure. The only difference is in the cell graph construction, which is done using approximate RangeCount queries.

In the quadtree for approximate RangeCount, each cell of side-length $\epsilon/\sqrt{d}$ is still recursively divided into $2^d$ sub-cells of the same size, but until either the sub-cell becomes empty or has side-length at most $\epsilon\rho/\sqrt{d}$. The tree has maximum depth $l = 1 + \lceil \log_2 1/\rho \rceil$. We use a modified version of our parallel quadtree construction method to parallelize approximate DBSCAN.

An approximate RangeCount($p$, $\epsilon$, $h$, $\rho$) query takes as input a point $p$, and returns an integer that is between the number of points in the $\epsilon$-radius and the number of points in the $\epsilon(1+\rho)$-radius of $p$ that are in $h$, (when using approximate RangeCount, all relevant methods takes an additional parameter $\rho$). If the answer is non-zero, then the core cell containing $p$ is connected to core cell $h$. Our query implementation starts a traversal of the quadtree from $h$, and recursively searches all children whose sub-cell intersects with the $\epsilon$-radius of $p$. As done in exact DBSCAN, our query is optimized to terminate once a zero count or a non-zero count can be determined. Once either a leaf node is reached or a node's sub-cell is completely contained in the $\epsilon(1+\rho)$-radius of $p$, the search on that path terminates. Queries do not modify the quadtree and can all be executed in parallel.

## 4.5 Analysis

This section analyzes the theoretical complexity of our algorithms, showing that they are work-efficient and have polylogarithmic depth.

### 2D Algorithms

**Grid Computation.** In our parallel algorithm presented in Section 4.2, creating $n$ key-value pairs can be done in $O(n)$ work and $O(1)$ depth in a data-parallel fashion. Semisorting takes $O(n)$ expected work and $O(\log n)$ depth w.h.p. Constructing the hash table and inserting non-empty cells into it takes $O(n)$ work and $O(\log n)$ depth

w.h.p. The overall cost of the parallel grid computation is therefore $O(n)$ work in expectation and $O(\log n)$ depth w.h.p.

**Box Computation.** The serial algorithm [117, 85] uses $O(n \log n)$ work, including sorting, scanning the points to assign them to strips and cells, and merging strips. However, the span is $O(n)$ since in the worst case there can be $O(n)$ strips.

Parallel comparison sorting takes $O(n \log n)$ work and $O(\log n)$ depth. Therefore, sorting the points by $x$-coordinate, and each strip by $y$-coordinate can be done in $O(n \log n)$ work and $O(\log n)$ depth overall. Parent finding using binary search for all points takes $O(n \log n)$ work and $O(1)$ depth. For pointer jumping, the longest path in the linked list halves on each round, and so the algorithm terminates after $O(\log n)$ rounds. We do $O(n)$ work per round, leading to an overall work of $O(n \log n)$. The depth is $O(1)$ per round, for a total of $O(\log n)$ overall. We repeat this process for the points in each strip, but in the $y$-direction, and the work and depth bounds are the same. For assigning pointers to neighboring cells for each cell, we use a parallel merging algorithm, which takes $O(n)$ work and $O(\log n)$ depth. The pointers are stored in an array, accessible in constant work and depth.

**MarkCore.** For cells with at least minPts points, we spend $O(n)$ work overall marking their points as core points (Lines 4–6 of Algorithm 2). All cells are processed in parallel, and all points can be marked in parallel, giving $O(1)$ depth.

For all cells with fewer than minPts points, each point only needs to execute a range count query on a constant number of neighboring cells [117, 104]. RangeCount($p$, $\epsilon$, $h$) compares $p$ to all points in neighboring cell $h$ in parallel. Across all queries, each cell will only be checked by $O(\mathsf{minPts})$ many points, and so the overall work for range counting is $O(n \cdot \mathsf{minPts})$. Therefore, Lines 8–13 of Algorithm 2 takes $O(n \cdot \mathsf{minPts})$ work. All points are processed in parallel, and there are a constant number of RangeCount calls per point, each of which takes $O(\log n)$ depth for a parallel prefix sum to obtain the number of points in the $\epsilon$-radius. Therefore, the depth for range counting is $O(\log n)$.

The work for looking up the neighbor cells is $O(n)$ and depth is $O(\log n)$ w.h.p. using the parallel hash table that stores the non-empty cells. Therefore, parallel MarkCore takes $O(n \cdot \mathsf{minPts})$ work and $O(\log n)$ depth w.h.p.

**Cell Graph Construction.** Reif and Sen present a parallel algorithm for constructing Voronoi diagrams and Delaunay triangulations in two dimensions in $O(n \log n)$ work and $O(\log n)$ depth w.h.p. [198]. For the Voronoi diagram approach, each nearest neighbor query can be answered in $O(\log n)$ work, which is used to check whether two cells should be connected and can be applied in parallel. Each cell will only execute a constant number of queries, and so the overall complexity is $O(n \log n)$ work

and $O(\log n)$ depth w.h.p. For the Delaunay triangulation approach, we can simply apply a parallel filter over all of the edges in the triangulation, keeping the edges between different cells with distance at most $\epsilon$. The cost of the filter is dominated by the cost of constructing the Delaunay triangulation.

For the USEC with line separation method, the sorted order of points in each dimension for each cell can be generated in $O(n \log n)$ work and $O(\log n)$ depth overall. To generate a wavefront on $n$ points, the exponential search has $O(\log n)$ steps per level, and each step involves a binary search which takes $O(\log n)$ work and depth. Therefore, the the work and depth for the searches is $O(\log^2 n)$ per level. Splitting and joining the binary trees to generate the new wavefront on each level takes $O(\log n)$ work and depth [17]. Thus, for the work, we obtain the recurrence $W(n) = 2W(n/2) + O(\log^2 n)$, which solves to $O(n)$. Since we can solve the recursive subproblems in parallel, for the depth, we obtain the recurrence $D(n) = D(n/2) + O(\log^2 n)$, which solves to $O(\log^3 n)$.

Checking whether the sorted set of points from one cell intersects with a wavefront from another cell can be done using an algorithm similar to parallel merging, as described in Section 4.2. In particular, for a wavefront of size $m$ and sorted point set of size $s$, we pick $k = m/\log(m+s)$ equally spaced pivots from the wavefront, and use binary search to split the sorted point set into subsets of size $s_1, \ldots, s_{k+1}$ where $\sum_{i=1}^{k+1} s_i = s$. The binary searches take a total of $O(k \log s) = O(m)$ work and $O(\log s)$ depth. Then, for the $i$'th pair forming a subproblem, we pick $j_i = s_i/\log(m+s)$ equally spaced pivots from the $i$'th subset of points and perform a binary search into the $i$'th subset of arcs of the wavefront to create more subproblems. This takes a total of $\sum_{i=1}^{k+1} O(j_i \log m) = O(s)$ work and $O(\log m)$ depth. The subset of points and subset of arcs for each subproblem are now all guaranteed to be of size $O(\log(m+s))$. In parallel across all subproblems, we run the serial USEC with line separation algorithm of [104], which takes linear work in the input size. Therefore, the work for this particular instance of USEC with line separation is $O(m+s)$ and the depth is $O(\log(m+s))$. All cell connectivity queries can be performed in parallel, and so the total work is $O(n)$ and and depth is $O(\log n)$. Since the sequential algorithms for wavefront generation and determining cell connectivity take linear work, our algorithm is work-efficient. After generating each wavefront, we write it out to an array by traversing the binary tree in parallel, which takes linear work and logarithmic depth [17]. Including the preprocessing step of sorting, our parallel USEC with line separation problem for determining the connectivity of core cells takes $O(n \log n)$ work and $O(\log^3 n)$ depth.

**Connected Components.** After the cell graph that contains $O(n)$ points and edges are constructed, we run connected components on the cell graph. This step can be

done in parallel in $O(n)$ work and $O(\log n)$ depth w.h.p. using parallel connectivity algorithms [108, 119, 76, 120, 189].

**ClusterBorder.** Using a similar analysis as done for marking core points, it can be shown that assigning cluster IDs to border points takes $O(n \cdot \mathsf{minPts})$ work sequentially [117, 85]. In parallel, since there are a constant number of neighboring cells for each non-core point, and all points in neighboring cells as well as all non-core points are checked in parallel, the depth is $O(1)$ for the distance comparisons. Looking up the neighboring cells can be done in $O(n)$ work and $O(\log n)$ depth w.h.p. using our parallel hash table. Adding cluster IDs to border point's set of clusters, while removing duplicates at the same time, can be done using parallel hashing in linear work and $O(\log n)$ depth w.h.p. The work is $O(n \cdot \mathsf{minPts})$ since we do not introduce any asymptotic work overhead compared to the sequential algorithm.

Overall, we have the following theorem.

**Theorem 1.** *For a constant value of* $\mathsf{minPts}$*, 2D Euclidean DBSCAN can be computed in* $O(n \log n)$ *work and* $O(\log n)$ *depth w.h.p.*

### Higher-dimensional Algorithm

For $d \geq 3$ dimensions, the BCP problem can be solved either using brute-force checking, which takes quadratic work, or using more theoretically-efficient algorithms that take sub-quadratic work [13, 68, 73]. This leads to a DBSCAN algorithm that takes $O((n \log n)^{4/3})$ expected work for $d = 3$ and $O(n^{2-(2/(\lceil d/2 \rceil + 1))+\delta})$ expected work for $d \geq 4$ where $\delta > 0$ is any constant [104]. The theoretically-efficient BCP algorithms seem too complicated to be practical (we are not aware of any implementations of these algorithms), and the actual implementation of [104] does not use them. However, we believe that it is still theoretically interesting to design a sub-quadratic work parallel BCP algorithm to use in DBSCAN, which is the focus of this section.

The sub-quadratic work BCP algorithms are based on constructing Delaunay triangulations (DT) in $d$ dimensions, which can be used for nearest neighbor search. However, we cannot afford to construct a DT on all the points, since a $d$-dimensional DT contains up to $O(n^{\lceil d/2 \rceil})$ simplices, which is at least quadratic in the worst-case for $d \geq 3$.

The idea in the algorithm by Agarwal et al. [13] is to construct multiple DTs, each for a subset of the points, and a nearest neighbor query then takes the closest neighbor among queries to all of the DTs. The data structure for nearest neighbor queries used by Aggarwal et al. is based on the RPO (Randomized Post Office) tree by Clarkson [73]. The RPO tree contains $O(\log n)$ levels where each node in the RPO tree corresponds to the DT of a random subset of the points. Parallel DT

for a constant dimension $d$ can be computed work-efficiently in expectation and in $O(\log n \log^* n)$ depth w.h.p. [45]. The children of each node can be determined by traversing the history graph of the DT, which takes $O(\log n)$ work and depth. The RPO tree is constructed recursively for $O(\log n)$ levels, and so the overall depth is $O(\log^2 n \log^* n)$ w.h.p. A query traverses down a path in the RPO tree, querying each DT along the path, which takes $O(\log^2 n)$ work and depth overall. Using this data structure to solve BCP gives a DBSCAN algorithm with $O(n^{2-(2/(\lceil d/2 \rceil+1))+\delta})$ expected work and $O(\log^2 n \log^* n)$ depth w.h.p. For $d = 3$, an improved data structure by Agarwal et al. [12] can be used to improve the expected work to $O((n \log n)^{4/3})$. The data structure is also based on DT, and so similar to before, we can parallelize the DT construction and obtain the same depth bound.

The overall bounds are summarized in the following theorem.

**Theorem 2.** *For a constant value of* minPts*, Euclidean DBSCAN can be solved in* $O((n \log n)^{4/3})$ *expected work for* $d = 3$ *and* $O(n^{2-(2/(\lceil d/2 \rceil+1))+\delta})$ *expected work for any constant* $\delta > 0$ *for* $d > 3$*, and polylogarithmic depth with high probability.*

### Approximate Algorithm

The algorithms for grid construction, marking core points, connected components, and clustering border points are the same as the exact algorithms, and so we only analyze approximate cell graph construction in the approximate algorithm based on the quadtree introduced in Section 4.4. The quadtree has $l = 1 + \lceil \log_2 1/\rho \rceil$ levels and can be constructed in $O(n'l)$ work sequentially for a cell with $n'$ points. A hash table is used to map non-empty cells to their quadtrees, which takes $O(n)$ work w.h.p. to construct. Using a fact from [25], Gan and Tao show that the number of nodes visited by a query is $O(1 + (1/\rho)^{d-1})$. Therefore, for constant $\rho$ and $d$, all of the quadtrees can be constructed in a total of $O(n)$ work w.h.p., and queries can be answered in $O(1)$ expected work.

All of the quadtrees can be constructed in parallel. To parallelize the construction of a quadtree for a cell with $n'$ points, we sort the points on each level in $O(n')$ work and $O(\log n')$ depth using parallel integer sorting [234], since the keys are integers in a constant range. In total, this gives $O(n'l)$ work and $O(l \log n')$ depth per quadtree. We use a parallel hash table to map non-empty cells to their quadtrees, which takes $O(n)$ work and $O(\log n)$ depth w.h.p. to construct. To construct the cell graph, all core points issue a constant number of queries to neighboring cells in parallel. The $O(n)$ hash table queries can be done in $O(n)$ work and $O(\log n)$ depth w.h.p. and thus cell graph construction has the same complexity. This gives the following theorem.

**Theorem 3.** *For constant values of* minPts *and* $\rho$*, our approximate Euclidean DBSCAN algorithm takes* $O(n)$ *work and* $O(\log n)$ *depth with high probability.*

## 4.6 Experiments

This section presents experiments comparing our exact and approximate algorithms as well as existing algorithms.

**Datasets.** We use the synthetic seed spreader (SS) datasets produced by Gan and Tao's generator [104]. The generator produces points generated by a random walk in a local neighborhood, but jumping to a random location with some probability. ***SS-simden*** and ***SS-varden*** refer to the datasets with similar-density and variable-density clusters, respectively. We also use a synthetic dataset called ***UniformFill*** that contains points distributed uniformly at random inside a bounding hypergrid with side length $\sqrt{n}$, where $n$ is the total number of points. The points have double-precision floating point values, but we scaled them to integers when testing Gan and Tao's implementation, which requires integer coordinates. We generated the synthetic datasets with 10 million points (unless specified otherwise) for dimensions $d = 2, 3, 5, 7$.

In addition, we use the following real-world datasets, which contain points with double-precision floating point values.

1. ***Household*** [92] is a 7-dimensional dataset with $2,049,280$ points excluding the date-time information.

2. ***GeoLife*** [259] is a 3-dimensional dataset with $24,876,978$ points. This dataset contains user location data (longitude, latitude, altitude), and its distribution is extremely skewed.

3. ***Cosmo50*** [153] is a 3-dimensional dataset with $321,065,547$ points. We extracted the $x$, $y$, and $z$ coordinate information to construct the 3-dimensional dataset.

4. ***OpenStreetMap*** [118] is a 2-dimensional dataset with $2,770,238,904$ points, containing GPS location data.

5. ***TeraClickLog*** [80] is a 13-dimensional dataset with $4,373,472,329$ points containing feature values and click feedback of online advertisements. As far as we know, *TeraClickLog* is the largest dataset used in the literature for *exact* DBSCAN.

We performed a search on $\epsilon$ and minPts for the synthetic datasets and chose the default parameters to be those that output a correct clustering. For the *SS* datasets, the default parameters that we use are similar to those found by Gan and Tao [104]. For ease of comparison, the default parameters for *Household* are the same as Gan and Tao [104] and the default parameters for *GeoLife*, *Cosmo50*, *OpenStreetMap*, and *TeraClickLog* are same as RP-DBSCAN [218]. For approximate DBSCAN, we set $\rho = 0.01$, unless specified otherwise.

**Testing Environment.** We perform all of our experiments on Amazon EC2 machines. We use a c5.18xlarge machine for testing of all datasets other than *Cosmo50*, *OpenStreetMap*, and *TeraClickLog*. The c5.18xlarge machine has 2 × Intel Xeon Platinum 8124M (3.00GHz) CPUs for a total for a total of 36 two-way hyper-threaded cores, and 144 GB of RAM. We use a r5.24xlarge machine for the three larger datasets just mentioned. The r5.24xlarge machine has 2 × Intel Xeon Platinum 8175M (2.50 GHz) CPUs for a total of 48 two-way hyper-threaded cores, and 768 GB of RAM. By default, we use all of the cores with hyper-threading on each machine. We compile our programs with the `g++` compiler (version 7.4) with the `-O3` flag, and use Cilk Plus for parallelism [136].

### Algorithms Tested

We implement the different methods for marking core points and BCP computation in exact and approximate DBSCAN for $d \geq 3$, and present results for the fastest versions, which are described below.

- ***our-exact***: This exact implementation implements the RangeCount query in marking core points by scanning through all points in the neighboring cell in parallel described in Section 4.2. For determining connectivity in the cell graph, it uses the BCP method described in Section 4.2.

- ***our-exact-qt***: This exact implementation implements the RangeCount query supported by the quadtree described in Section 4.4. For determining connectivity in the cell graph, it uses the BCP method described in Section 4.2.

- ***our-approx***: This approximate implementation implements the RangeCount query in marking core points by scanning through all points in the neighboring cell in parallel, and uses the quadtree for approximate RangeCount queries in cell graph construction described in Section 4.4.

- ***our-approx-qt***: This approximate implementation is the same as *our-approx* except that it uses the RangeCount query supported by the quadtree described in Section 4.4 for marking core points.

We append the ***-bucketing*** suffix to the names of these implementations when using the bucketing optimization described in Section 4.2.

For $d = 2$, we have six implementations that differ in whether they use the grid or the box method to construct cells and whether they use BCP, Delaunay triangulation, or USEC with line separation to construct the cell graph. We refer to these as ***our-2d-grid-bcp***, ***our-2d-grid-usec***, ***our-2d-grid-delaunay***, ***our-2d-box-bcp***, ***our-2d-box-usec***, and ***our-2d-box-delaunay***.

We note that our exact algorithms return the same answer as the standard DBSCAN definition, and our approximate algorithms return answers that satisfy Gan and Tao's approximate DBSCAN definition (see Section 2).

We compare with the following implementations:

- ***Gan&Tao-v2*** [104] is the state-of-the-art serial implementation for both exact and approximate DBSCAN. *Gan&Tao-v2* only accepts integer values between 0 and $100,000$, and so when running their code we scaled the datasets up into this integer range and scaled up the $\epsilon$ value accordingly to achieve a consistent clustering output with other methods.

- ***pdsdbscan*** [184] is the implementation of the parallel disjoint-set exact DBSCAN by Patwary et al. compiled with OpenMP.

- ***hpdbscan*** [112] is the implementation of parallel exact DBSCAN by Gotz et al. compiled with OpenMP. We modified the source code to remove the file output code.

- ***rpdbscan*** [218] is the state-of-the-art distributed implementation for DBSCAN using Apache Spark. We note that their variant does not return the same result as DBSCAN. We tested *rpdbscan* on the same machine that we used, and also report the timings in [218], which were obtained using at least as many cores as our largest machine.

### Experiments for $d \geq 3$

We first evaluate the performance of the different algorithms for $d \geq 3$. In the following plots, data points that did not finish within an hour are not shown.

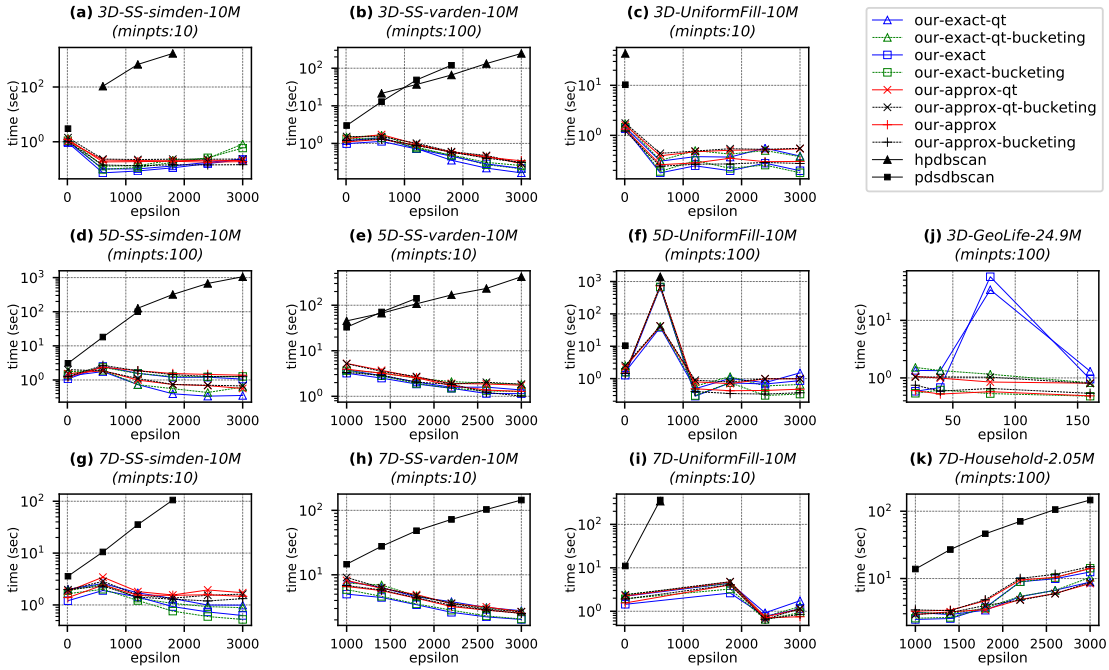



Figure 10: Running time vs. $\epsilon$ on 36 cores with hyper-threading. The $y$-axes are in log-scale.

**Influence of $\epsilon$ on Parallel Running Time.** In this experiment, we fix the default value of minPts corresponding to the correct clustering, and vary $\epsilon$ within a range centered around the default $\epsilon$ value. Figure 10 shows the parallel running time vs. $\epsilon$ for the different implementations. In general, both *pdsdbscan* and *hpdbscan* becomes slower with increasing $\epsilon$. This is because they use pointwise range queries, which get more expensive with larger $\epsilon$. Our methods tend to improve with increasing $\epsilon$ because there are fewer cells leading to a smaller cell graph, which speeds up computations on the graph. Our implementations significantly outperform *pdsdbscan* and *hpdbscan* on all of the data points.

We observe a spike in plot Figure 10(f) when $\epsilon = 608$. The implementations that mark core points by scanning through all points in neighboring cells spend a significant amount of time in that phase; in comparison, the quadtree versions perform better because of their more optimized range counting. There is also a spike in Figure 10(j) when $\epsilon = 80$. Our exact implementation spends a significant amount of time in cell graph construction. This is because the *GeoLife* dataset is heavily skewed, certain cells could contain significantly more points. When many

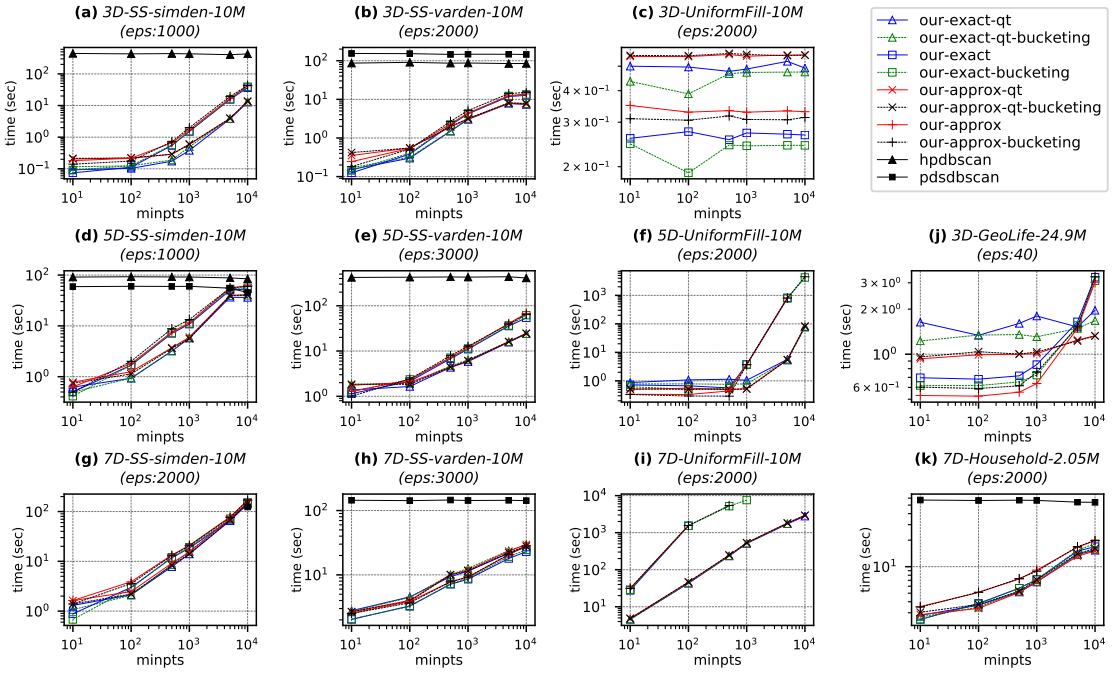



Figure 11: Running time vs. minPts on 36 cores with hyper-threading. The $y$-axes are in log-scale.

cell connectivity queries involve these cells, the quadratic nature using the BCP approach in *our-exact* makes the cost of queries expensive. On the contrary, methods using the quadtree for cell graph construction (*our-exact-qt*, *our-approx-qt*, and *our-approx*) tend to have consistent performance across the $\epsilon$ values. For the spike in Figure 10(j), it is interesting to see that the bucketing implementations, *our-exact-qt-bucketing* and *our-exact-bucketing*, are significantly faster than *our-exact-qt* and *our-exact* because many of the expensive connectivity queries are pruned.

**Influence of minPts on Parallel Running Time.** In this experiment, we fix the default value of $\epsilon$ for a dataset and vary minPts over a range from 10 to 10,000. Figure 11 shows that our implementations have an increasing trend in running time as minPts increases in most cases. This is consistent with our analysis in Section 4.5 that the overall work for marking core points is $O(n \cdot \text{minPts})$. In contrast, minPts does not have much impact on the performance of *hpdbscan* and *pdsdbscan* because their range queries, which dominate the total running times, do not depend on minPts. Our implementations outperform *hpdbscan* and *pdsdbscan* for almost all values of

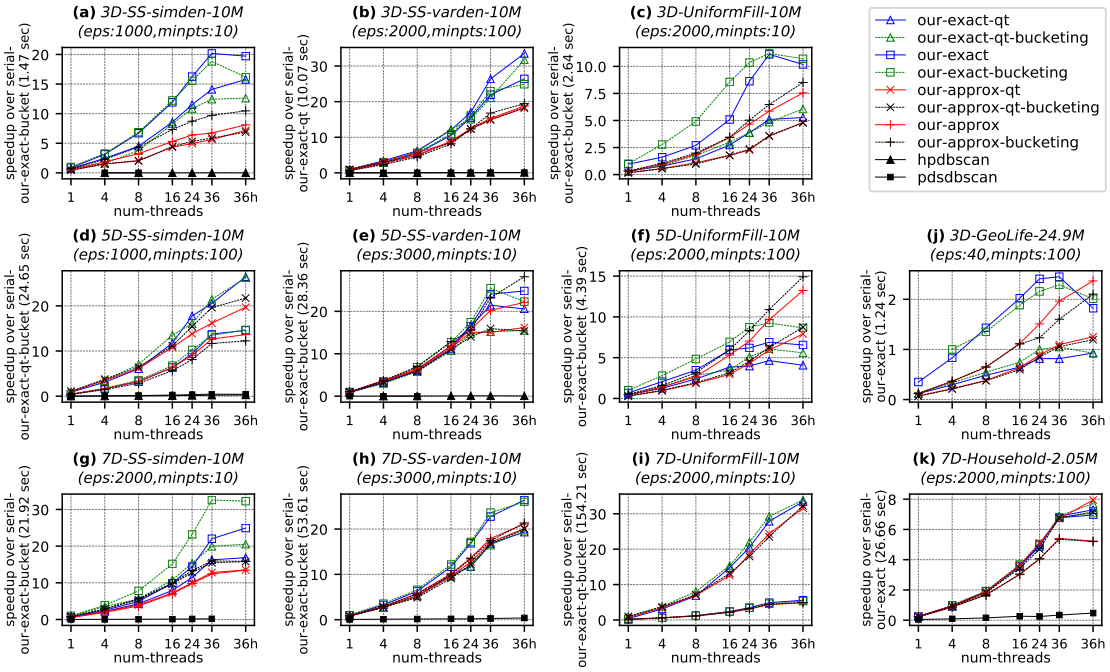



Figure 12: Speedup of implementations over the *best* serial baselines vs. thread count. The best serial baseline and its running time for each dataset is shown on the $y$-axis label. "36h" on the $x$-axes refers to 36 cores with hyper-threading.

minPts. Figures 11(d) and 11(g) suggests that *hpdbscan* can surpass our performance for certain datasets when minPts $= 10,000$. However, as suggested by Schubert et al. [206], the minPts value used in practice is usually much smaller, and based on our observation, a minPts value of at most 100 usually gives the correct clusters.

**Parallel Speedup.** To the best of our knowledge, *Gan&Tao-v2* is the fastest existing serial implementation both for exact and approximate DBSCAN. However, we find that across all of our datasets, our serial implementations are faster than theirs by an average of 5.18x and 1.52x for exact DBSCAN and approximate DBSCAN, respectively. In Figure 12, we compare the speedup of the parallel implementations under different thread counts over the best serial baselines for each dataset and choice of parameters. We also show the self-relative speedups for one dataset in Figure 13 and note that the trends are similar on other datasets. For these experiments, we use parameters that generate the correct clusters. Our implementations obtain very good speedups on most datasets, achieving speedups of 5–33x (16x on average)

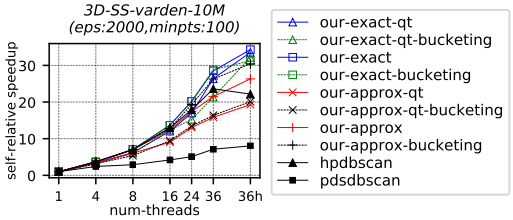



Figure 13: Self-relative speedup of implementations vs. thread count. "36h" on the $x$-axis refers to 36 cores with hyper-threading.

over the best serial baselines. Additionally, the self-relative speedups of our exact and approximate methods are 2–89x (24x on average) and 14-44x (24x on average), respectively. Although *hpdbscan* and *pdsdbscan* achieve good self-relative speedup (22–31x and 7–20x, respectively), they fail to outperform the serial implementation on most of the datasets. Compared to *hpdbscan* and *pdsdbscan*, we are faster by up to orders of magnitude (16–6102x).

Our speedup on the GeoLife dataset (Figure 12(j)) is low due to the high skewness of cell connectivity queries caused by the skewed point distribution, however the parallel running time is reasonable (less than 1 second). In contrast, *hpdbscan* and *pdsdbscan* did not terminate within an hour.

The bucketing heuristic achieved the best parallel performance for several of the datasets (Figures 10(f), (g), and (j); Figures 11(c) and (j); and Figures 12(c), (f), (g), and (j)). In general, the bucketing heuristic greatly reduces the number of connectivity queries during cell graph construction, but in some cases it can reduce parallelism and/or increase overhead due to sorting. We also observe a similar trend on all methods where bucketing is applied.

We also implemented our own parallel baseline based on the original DBSCAN algorithm [95]. We use a parallel $k$-d tree, and all points perform queries in parallel to find all neighbors in their $\epsilon$-radius to check if they should be a core point. However, the baseline was over 10x slower than our fastest parallel implementation for datasets with the correct parameters, and hence we do not show it in the plots.

**Influence of $\rho$ on Parallel Running Time.** Figure 14 shows the effect of varying $\rho$ for our two approximate DBSCAN implementations. We also show our best exact method as a baseline. We only show plots for two datasets as the trend was similar in other datasets. We observe a small decrease in running time as $\rho$ increases, but find that the approximate methods are still mostly slower than the best exact method. On average, for the parameters corresponding to correct clustering, we find

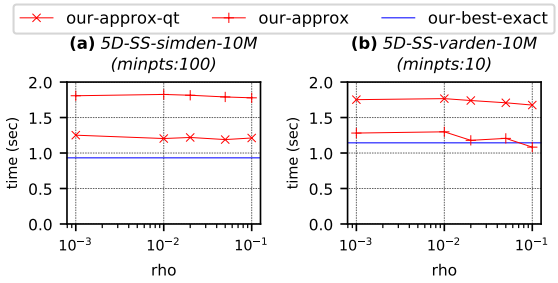



Figure 14: Running time vs. $\rho$ on 36 cores with hyper-threading.

that our best exact method is 1.24x and 1.53x faster than our best approximate method when running in parallel and serially, respectively; this can also be seen in Figure 12. Schubert et al. [206] also found exact DBSCAN to be faster than approximate DBSCAN for appropriately-chosen parameters, which is consistent with our observation.

**Large-scale Datasets.** In Table 3, we show the running times of *our-exact* on large-scale datasets. We compare with the reported numbers for the state-of-the-art distributed implementation *rpdbscan*, which use 48 cores distributed across 12 machines [218], as well as numbers for *rpdbscan* on our machines. The purpose of this experiment is to show that we are able to efficiently process large datasets using just a multicore machine. *GeoLife* was run on the 36 core machine whereas others were run on the 48 core machine due to their larger memory footprint. We see that *our-exact* achieves a 18–577x speedup over *rpdbscan* using the same or a fewer number of cores. We believe that this speedup is due to lower communication costs in shared-memory as well as a better algorithm. Even though *TeraClickLog* is significantly larger than the other datasets, our running times are not proportionally larger. This is because for the parameters chosen by [218], all points fall into one cell. Therefore, in our implementation all points are core points and are trivially placed into the only cluster. In contrast, *rpdbscan* incurs communication costs in partitioning the points across machines and merging the clusters from different machines together.

### Experiments for $d = 2$

In Figure 15, we show the performance of our six 2D algorithms as well as *hpdbscan* and *pdsdbscan* on the synthetic datasets. We show the running time while varying $\epsilon$, minPts, number of points, or number of threads. We first note that all of our implementations are significantly faster than *hpdbscan* and *pdsdbscan*. In general, we found

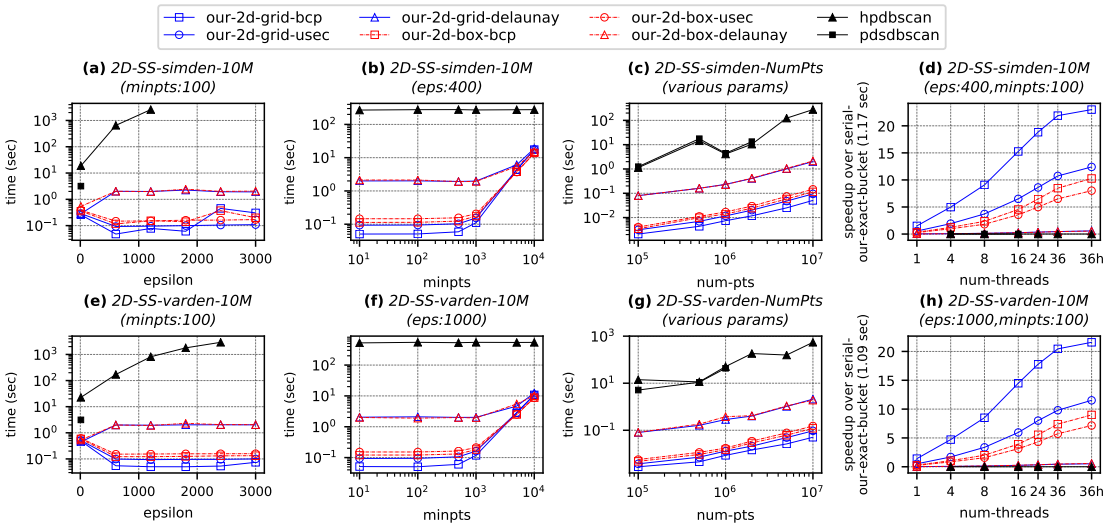



Figure 15: Running time vs. $\epsilon$, minPts, number of points, or thread count for the 2D implementations. In (c) and (g), the parameters are chosen for each input size such that the algorithm outputs the correct clustering. In (d) and (h), "36h" on the $x$–axis refers to 36 cores with hyper-threading. The $y$-axes in (a)–(c) and (e)–(g) are in log-scale.

| | GeoLife | | | | Cosmo50 | | | |
|---|---|---|---|---|---|---|---|---|
| $\epsilon$ | 20 | 40 | 80 | 160 | 0.01 | 0.02 | 0.04 | 0.08 |
| *our-exact* | 0.541 | 0.617 | 0.535 | 0.482 | 41.8 | 5.51 | 4.69 | 3.03 |
| *rpdbscan (our machine)* | 29.13 | 27.92 | 32.04 | 27.81 | 3750 | 562.0 | 576.9 | 672.6 |
| *rpdbscan ([218])* | 36 | 33 | 28 | 27 | 960 | 504 | 438 | 432 |
| | OpenStreetMap | | | | TeraClickLog | | | |
| $\epsilon$ | 0.01 | 0.02 | 0.04 | 0.08 | 1500 | 3000 | 6000 | 12000 |
| *our-exact* | 41.4 | 43.2 | 40 | 44.5 | 26.8 | 26.9 | 27.0 | 27.6 |
| *rpdbscan (our machine)* | – | – | – | – | – | – | – | – |
| *rpdbscan ([218])* | 3000 | 1720 | 1200 | 840 | 15480 | 7200 | 3540 | 1680 |

Table 3: Parallel running times (seconds) for *our-exact* and *rpdbscan*. The value of minPts is set to 100. *GeoLife* was run on the 36 core machine and the other datasets were run on the 48 core machine. For *rpdbscan*, we omit timings for experiments that encountered exceptions or did not complete within 1 hour. We also include the distributed running times reported in [218] that used as many cores as our machines.

the grid-based implementations to be faster than the box-based implementations due to the higher cell construction time of the boxed-based implementations. We also found the Delaunay triangulation-based implementations to be significantly slower than the BCP and USEC-based methods due to the high overhead of computing the Delaunay triangulation. The fastest implementation overall was *our-2d-grid-bcp*.

**Comparison with GPU Implementation**

While there exists GPU implementation of DBSCAN, we believe our implementations are much faster than the existing GPU implementations. We first compare with *G-DBSCAN* [21], and their implementation takes 74.4 seconds to process a data set with 0.7 million points on an Nvidia Tesla M2050 with 448 cores. *G-DBSCAN* first creates a graph on the data set, where an edge is created between two vertices within $\epsilon$ to each other, after which a breadth-first search is computed on the graph. The algorithm may not be work-efficient since the size of the graph can be quadratic in the size of the input. We also compare with *CUDA-DClust+* [191], which takes 15.62 seconds to process a data set with 10 million points using an Nvidia Quadro GP100 with 3584 cores. In comparison, our implementation takes much less than 1 second in most cases to process data sets with 10 million points as shown by our experiments described earlier. *CUDA-DClust+* reduces the quadratic cost of $\epsilon$-neighborhood searches using an index structure, which is a similar strategy to our optimized implementations.

# 5 Fast Parallel Algorithms for Euclidean Minimum Spanning Tree and Hierarchical Spatial Clustering

## 5.1 Introduction

There has been a significant amount of theoretical work on designing fast sequential EMST algorithms (e.g., [13, 254, 210, 26, 58]). There have also been some practical implementations of EMST [170, 66, 35, 178], although most of them are sequential (part of the algorithm by Chatterjee et al. [66] is parallel). The state-of-the-art EMST implementations are either based on generating a well-separated pair decomposition (WSPD) [60] and applying Kruskal's minimum spanning tree (MST) algorithm on edges produced by the WSPD [66, 178], or dual-tree traversals on $k$-d trees integrated into Boruvka's MST algorithm [170]. Much less work has been proposed for parallel HDBSCAN* and OPTICS [184, 203]. In this thesis, we design new algorithms for EMST, which can also be leveraged to design a fast parallel HDBSCAN* algorithm.

This thesis presents practical parallel in-memory algorithms for HDBSCAN* and EMST, and proves that the theoretical work of our implementations matches their state-of-the-art counterparts, while having polylogarithmic depth. Our algorithms are based on finding a WSPD and then running Kruskal's algorithm on edges between pairs in the WSPD. For our HDBSCAN* algorithm, we propose a new notion of well-separation to include the notion of core distances, which enables us to improve the space usage and work of our algorithm.

Given the MST from the HDBSCAN* or the EMST problem, we provide an algorithm to generate a dendrogram, which represents the hierarchy of clusters in our data set. For EMST, this solves the single-linkage clustering problem [113], and for HDBSCAN*, this gives us a dendrogram as well as a reachability plot [61]. We introduce a work-efficient parallel divide-and-conquer algorithm that first generates an Euler tour on the tree, splits the tree into multiple subtrees, recursively generates the dendrogram for each subtree, and glues the results back together. An in-order traversal of the dendrogram gives the reachability plot. Our algorithm takes $O(n \log n)$ work and $O(\log^2 n \log \log n)$ depth. Our parallel dendrogram algorithm is of independent interest, as it can also be applied to generate dendrograms for other clustering problems.

We also present several additional theoretical results: (1) an EMST algorithm

with subquadratic work and polylogarithmic depth based on a subquadratic-work sequential algorithm by Callahan and Kosaraju [58]; (2) an HDBSCAN*algorithm for two dimensions with $O(\mathsf{minPts}^2 \cdot n \log n)$ work, matching the sequential algorithm by Berg et al. [85], and $O(\mathsf{minPts} \cdot \log^2 n)$ depth; and (3) a work-efficient parallel algorithm for approximate OPTICS based on the sequential algorithm by Gan and Tao [105].

We provide optimized parallel implementations of our EMST and HDBSCAN* algorithms. We introduce a memory optimization that avoids computing and materializing many of the WSPD pairs, which significantly improves our algorithm's performance (up to 8x faster and 10x less space). We also provide optimized implementations of $k$-d trees, which our algorithms use for spatial queries.

We perform a comprehensive set of experiments on both synthetic and real-world data sets using varying parameters, and compare the performance of our implementations to optimized sequential implementations as well as existing parallel implementations. Compared to existing EMST sequential implementations [170, 171], our fastest sequential implementation is 0.89–4.17x faster (2.44x on average). On a 48-core machine with hyper-threading, our EMST implementation achieves 14.61–55.89x speedup over the fastest sequential implementations. Our HDBSCAN* implementation achieves 11.13–46.69x speedup over the fastest sequential implementations. Compared to existing sequential and parallel implementations for HDBSCAN* [105, 171, 203, 184], our implementation is at least an order of magnitude faster. Our source code is publicly available at `https://github.com/wangyiqiu/hdbscan`.

## 5.2 Parallel EMST and HDBSCAN*

In this section, we present our new parallel algorithms for EMST and HDBSCAN*. We also introduce our new memory optimization to improve space usage and performance in practice.

### EMST

To solve EMST, Callahan and Kosaraju present an algorithm for constructing a WSPD that creates an edge between the BCCP of each pair in the WSPD with weight equal to their distance, and then runs an MST algorithm on these edges. They show that their algorithm takes $O(T_d(n, n) \log n)$ work [58], where $T_d(n, n)$ refers to the work of computing BCCP on two sets each of size $n$.

For our parallel EMST algorithm, we parallelize WSPD construction algorithm, and then develop a parallel variant of Kruskal's MST algorithm that runs on the

**Algorithm 5** Well-Separated Pair Decomposition

```
1: procedure WSPD(A)
2:   if |A| > 1 then
3:     do in parallel
4:       WSPD(A_left)                          ▷ parallel call on the left child of A
5:       WSPD(A_right)                         ▷ parallel call on the right child of A
6:     FINDPAIR(A_left, A_right)
7: procedure FINDPAIR(P, P')
8:   if P_diam < P'_diam then
9:     SWAP(P, P')
10:  if WELLSEPARATED(P, P') then RECORD(P, P')
11:  else
12:    do in parallel
13:      FINDPAIR(P_left, P')                  ▷ P_left is the left child of P
14:      FINDPAIR(P_right, P')                 ▷ P_right is the right child of P
```

edges formed by the pairs in the WSPD. We also propose a non-trivial optimization to make the implementation fast and memory-efficient.

**Constructing a WSPD in Parallel.** We introduce the basic parallel WSPD in Algorithm 5. Prior to calling WSPD, we construct a spatial median $k$d-tree $T$ in parallel with each leaf containing one point. Then, we call the procedure WSPD on line 1 and make the root node of $T$ its input. In WSPD, we make parallel calls to FINDPAIR on the two children of all non-leaf nodes by recursively calling WSPD. The procedure FINDPAIR on line 7 takes as input a pair $(P, P')$ of nodes in $T$, and checks whether $P$ and $P'$ are well-separated. If they are well-separated, then the algorithm records them as a well-separated pair on line 10; otherwise, the algorithm splits the set with the larger bounding sphere into its two children and makes two recursive calls in parallel (Lines 13–14). This process is applied recursively until the input pairs are well-separated. The major difference of Algorithm 1 from the serial version is the parallel thread-spawning on Lines 3–5 and 12–14. This procedure generates a WSPD with $O(n)$ pairs [58].

**Parallel GFK Algorithm for EMST.** The original algorithm by Callahan and Kosaraju [58] computes the BCCP between each pair in the WSPD to generate a graph from which an MST can be computed to obtain the EMST. However, it is not necessary to compute the BCCP for all pairs, as observed by Chatterjee et al. [66]. Our implementation only computes the BCCP between a pair if their points are not

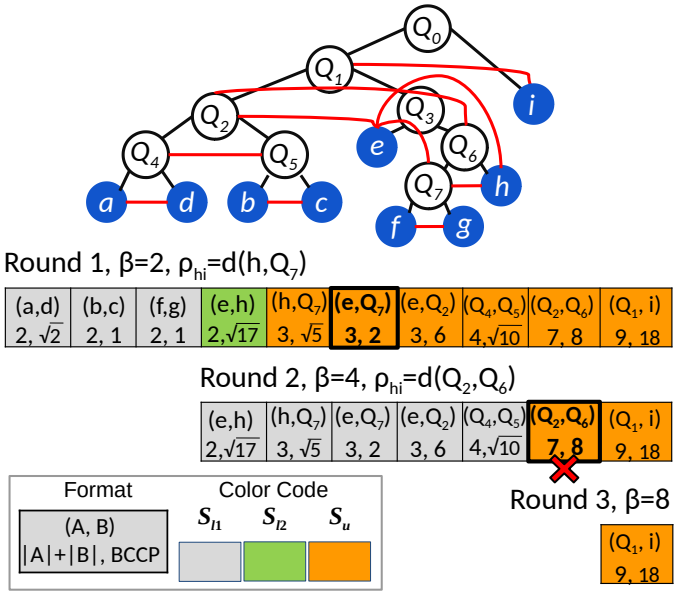



Figure 16: The is an example for both GFK (Algorithm 6) and MemoGFK (Algorithm 7) for EMST corresponding to the data set shown in Figure 4. The red lines linking tree nodes and the boxes drawn below represent well-separated pairs. The boxes also show the cardinality and BCCP value of the pair. Their correspondence with the symbols $S_{l1}$, $S_{l2}$, and $S_u$ from the pseudocode are color-coded. The pairs that generate $\rho_{hi}$ are in bold squares, and the pairs filtered out have a red cross. Using our MemoGFK optimization, only the pairs in $S_{l1}$ needs to be materialized, in contrast to needing to materialize all of the pairs in GFK.

**Algorithm 6** Parallel GeoFilterKruskal

1: **procedure** ParallelGFK(WSPD: $S$, Edges: $E_{out}$, UnionFind: $UF$)
2: $\beta = 2$
3: **while** $|E_{out}| < (n-1)$ **do**
4: $(S_l, S_u) = \textsc{Split}(S,\ f_\beta)$ ▷ For a pair $(A, B)$, $f_\beta$ checks if $|A| + |B| \leq \beta$
5: $\rho_{hi} = \min_{(A,B)\in S_u} d(A, B)$
6: $(S_{l1}, S_{l2}) = \textsc{Split}(S_l,\ f_{\rho_{hi}})$ ▷ For a pair $(A, B)$, $f_{\rho_{hi}}$ checks if $\mathsf{BCCP}(A, B) \leq \rho_{hi}$
7: $E_{l1} = \textsc{GetEdges}(S_{l1})$ ▷ Retrieves edges associated with pairs in $S_{l1}$
8: $\textsc{ParallelKruskal}(E_{l1},\ E_{out},\ UF)$
9: $S = \textsc{Filter}(S_{l2} \cup S_u,\ f_{diff})$ ▷ For a pair $(A, B)$, $f_{diff}$ checks points in $A$ are in different component from $B$ in $UF$
10: $\beta = \beta \times 2$

yet connected in the spanning forest generated so far. This optimization reduces the total number of BCCP calls. Furthermore, we propose a memory optimization that avoids materializing all of the pairs in the WSPD. We will first describe how we obtain the EMST from the WSPD, and then give details of our memory optimization.

The original Kruskal's algorithm is an MST algorithm that takes input edges sorted by non-decreasing weight, and processes the edges in order, using a union-find data structure to join components for edges with endpoints in different components. Our implementation is inspired by a variant of Kruskal's algorithm, GeoFilterKruskal (GFK). This algorithm was used for sequential EMST by Chatterjee et al. [66], and for MST in general graphs by Osipov et al. [180]. It improves Kruskal's algorithm by avoiding the BCCP computation between pairs unless needed, and prioritizing BCCPs between pairs with smaller cardinalities, which are cheaper, with the goal of pruning more expensive BCCP computations.

We propose a parallel GFK algorithm as shown in Algorithm 6. It uses Kruskal's MST algorithm as a subroutine by passing it batches of edges, where each batch has edges with weights no less than those of edges in previous batches, and the union-find structure is shared across multiple invocations of Kruskal's algorithm. ParallelGFK takes as input the WSPD pairs $S$, an array $E_{out}$ to store the MST edges, and a union-find structure $UF$. On each round, given a constant $\beta$, we only consider node pairs in the WSPD with cardinality (sum of sizes) at most $\beta$ because it is cheaper to compute their BCCPs. To do so, the set of pairs $S$ is partitioned into $S_l$, containing pairs with cardinality at most $\beta$, and $S_u$, containing the remaining pairs (line 4). However, it is only correct to consider pairs in $S_l$ that produce edges

lighter than any of the pairs in $S_u$. On line 5, we compute an upper bound $\rho_{hi}$ for the edges in $S_l$ by setting $\rho_{hi}$ equal to the minimum $d(A, B)$ for all $(A, B) \in S_u$ (this is a lower bound on the edges weights formed by these pairs). In the example shown in Figure 16, in the first round, with $\beta = 2$, the set $S_l$ contains $(a, d)$, $(b, c)$, $(f, g)$, and $(e, h)$, and the set $S_u$ contains $(h, Q_7)$, $(e, Q_7)$, $(e, Q_2)$, $(Q_4, Q_5)$, $(Q_2, Q_6)$, and $(Q_1, i)$. $\rho_{hi}$ corresponds to $(e, Q_7)$ on Line 5. Then, we compute the BCCP of all elements of set $S_l$, and split it into $S_{l1}$ and $S_{l2}$, where $S_{l1}$ has edges with weight at most $\rho_{hi}$ (line 6). On Line 6, $S_{l1}$ contains $(a, d)$, $(b, c)$ and $(f, g)$, as their BCCP distances are smaller than $\rho_{hi} = d(e, Q_7)$, and $S_{l2}$ contains $(e, h)$ . After that, $E_{l1}$, the edges corresponding to $S_{l1}$, are passed to Kruskal's algorithm (Lines 7–8). The remaining pairs $S_{l2} \cup S_u$ are then filtered based on the result of Kruskal's algorithm (Line 9)—in particular, pairs that are connected in the union-find structure of Kruskal's algorithm can be discarded, and for many of these pairs we never have to compute their BCCP. In Figure 16, the second round processes $(e, h)$, $(h, Q_7)$, $(e, Q_7)$, $(e, Q_2)$, $(Q_4, Q_5)$, $(Q_2, Q_6)$, and $(Q_1, i)$, and works similarly to Round 1. However, $(Q_2, Q_6)$ gets filtered out during the second round, and we never have to compute its BCCP, leading to less work compared to a naive algorithm. Finally, the subsequent rounds process a single pair $(Q_1, i)$. At the end of each round, we double the value of $\beta$ to ensure that there are logarithmic number of rounds and hence better depth (in contrast, the sequential algorithm of Chatterjee et al. [66] increases $\beta$ by 1 every round). Throughout the algorithm, we cache the BCCP results of pairs to avoid repeated computations. Overall, the main difference between Algorithm 2 and sequential algorithm is the use of parallel primitives on nearly every line of the pseudocode, and the exponentially increasing value of $\beta$ on Line 11, which is crucial for achieving a low depth bound.

The following theorem summarizes the bounds of our algorithm.

**Theorem 4.** *We can compute the EMST on a set of $n$ points in constant dimensions in $O(n^2)$ work and $O(\log^2 n)$ depth.*

*Proof.* Callahan [57] shows that a WSPD with $O(n)$ well-separated pairs can be computed in $O(n \log n)$ work and $O(\log n)$ depth, which we use for our analysis. Our parallel GeoFilterKruskal algorithm for EMST proceeds in rounds, and processes the well-separated pairs in an increasing order of cardinality. Since $\beta$ doubles on each round, there can be at most $O(\log n)$ rounds since the largest pair can contain $n$ points. Within each round, the Split on Line 4 and Filter on Line 9 both take $O(n)$ work and $O(\log n)$ depth. We can compute the BCCP for each pair on Line 6 by computing all possible point distances between the pair, and using WriteMin to obtain the minimum distance. Since the BCCP of each pair will only be computed

once and is cached, the total work of BCCP on Line 6 is $\sum_{A,B \in S} |A||B| = O(n^2)$ work since the WSPD is an exact set cover for all distinct pairs. Therefore, Line 6 takes $O(n^2)$ work across all rounds and $O(1)$ depth for each round. Given $n$ edges, the MST computation on Line 8 can be done in $O(n \log n)$ work and $O(\log n)$ depth using existing parallel algorithms [138]. Therefore, the overall work is $O(n^2)$. Since each round takes $O(\log n)$ depth, and there are $O(\log n)$ rounds, the overall depth is $O(\log^2 n)$. □

We note that there exist subquadratic work BCCP algorithms [13], which result in a subquadratic work EMST algorithm. Although the algorithm is impractical and no implementations exist, for theoretical interest we give a work-efficient parallel algorithm with polylogarithmic depth in Section 5.5 of the Appendix.

We implemented our own sequential and parallel versions of the GFK algorithm as a baseline based on Algorithm 6, which we found to be faster than the implementation of Chatterjee et al. [66] in our experiments. In addition, because the original GFK algorithm requires materializing the full WSPD, its memory consumption can be excessive, limiting the algorithm's practicality. This issue worsens as the dimensionality of the points increases, as the number of pairs in the WSPD increases exponentially with the dimension. While Chatterjee et al. [66] show that their GFK algorithm is efficient, they consider much smaller data sets than the ones in this thesis.

**The MemoGFK Optimization.** To tackle the memory consumption issue, we propose an optimization to the GFK algorithm, which reduces its space usage and improves its running time in practice. We call the resulting algorithm ***MemoGFK*** (memory-optimized GFK). The basic idea is that, rather than materializing the full WSPD at the beginning, we partially traverse the $k$d-tree on each round and retrieve only the pairs that are needed. The pseudocode for our algorithm is shown in Algorithm 7, where ParallelMemoGFK takes in the root $R$ of a $k$d-tree, an array $E_{out}$ to store the MST edges, and a union-find structure $UF$.

The algorithm proceeds in rounds similar to parallel GeoFilterKruskal, and maintains lower and upper bounds ($\rho_{lo}$ and $\rho_{hi}$) on the weight of edges to be considered each round. On each round, it first computes $\rho_{hi}$ based on $\beta$ by a single $k$d-tree traversal, which will be elaborated below (line 4). Then, together with $\rho_{lo}$ from the previous round ($\rho_{lo} = 0$ on the first round), the algorithm retrieves pairs with BCCP distance in the range $[\rho_{lo}, \rho_{hi})$ via a second $k$d-tree traversal on line 5. The edges corresponding to these pairs are then passed to Kruskal's algorithm on line 7. An example of the first round of the algorithm with MemoGFK is illustrated in Figure 16. Without the optimization, the GFK algorithm needs to first materialize all

**Algorithm 7** Parallel MemoGFK

1: **procedure** ParallelMemoGFK($k$d-tree root: $R$, Edges: $E_{out}$, UnionFind: $UF$)
2: $\beta = 2$, $\rho_{lo} = 0$
3: **while** $|E_{out}| < (n-1)$ **do**
4: $\rho_{hi} =$ GetRho($R$, $\beta$)
5: $S_{l1} =$ GetPairs($R$, $\beta$, $\rho_{lo}$, $\rho_{hi}$, $UF$)
6: $E_{l1} =$ GetEdges($S_{l1}$) ▷ Retrieves edges associated with pairs in $S_{l1}$
7: ParallelKruskal($E_{l1}$, $E_{out}$, $UF$)
8: $\beta = \beta \times 2$, $\rho_{lo} = \rho_{hi}$

of the pairs in Round 1. With MemoGFK, $\rho_{hi} = d(e, Q_7)$ is computed via a tree traversal on Line 4, after which only the pairs in the set $S_{l1} = \{(a,d),(b,c),(f,g)\}$ are retrieved and materialized on Line 5 via a second tree traversal. Retrieving pairs only as needed reduces memory usage and improves performance. The correctness of the algorithm follows from the fact that each round considers non-overlapping ranges of edge weights in increasing order until all edges are considered, or when MST is completed.

Now we discuss the implementation details of the two-pass tree traversal on Line 4–5. The GetRho subroutine, which computes $\rho_{hi}$, does so by finding the lower bound on the minimum separation of pairs whose cardinality is greater than $\beta$ and are not yet connected in the MST. We traverse the $k$d-tree starting at the root, in a similar way as when computing the WSPD in algorithm 5. During the process, we update a global copy of $\rho_{hi}$ using WriteMin whenever we encounter a well-separated pair in FindPair, with cardinality greater than $\beta$. We can prune the traversal once $|A| + |B| \leq \beta$, as all pairs that originate from $(A, B)$ will have cardinality at most $\beta$. We also prune the traversal when the two children of a tree node are already connected in the union-find structure, as these edges will not need to be considered by Kruskal's algorithm. In addition, we prune the traversal when the distance between the bounding spheres of $A$ and $B$, $d(A, B)$, is larger than $\rho_{hi}$, as its descendants cannot produce a smaller distance.

The GetPairs subroutine then retrieves all pairs whose points are not yet connected in the union-find structure and have BCCP distances in the range $[\rho_{lo}, \rho_{hi})$. It does so also via a pruned traversal on the $k$d-tree starting from the root, similarly to algorithm 5, but only retrieves the useful pairs. For a pair of nodes encountered in the FindPair subroutine, we estimate the minimum and maximum possible BCCP between the pair using bounding sphere calculations, an example of which is shown

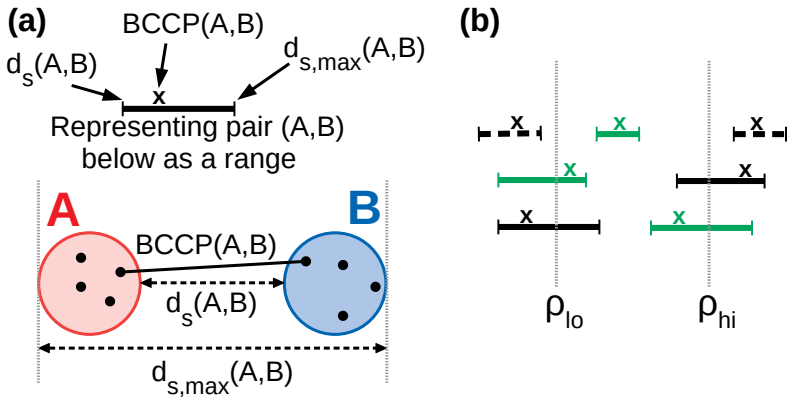



Figure 17: (a) shows a representation of a well-separated pair $(A, B)$ as a line segment, based on the values of its $d(A, B)$ and $d_{\max}(A, B)$, which serve as the lower and upper bounds, respectively, for their BCCP and the BCCP of their descendants. The "x"'s on the line marks the value of the BCCP. (b) shows an example of tree node pairs encountered during a pruned tree traversal on Line 5 of Algorithm 7, where the pairs are represented the same way as in (a). The pairs in solid green lines, if well-separated, will be retrieved and materialized because their BCCPs are within the $[\rho_{lo}, \rho_{hi})$ range, whereas those in solid black lines will not as their BCCPs are out of range (although their BCCPs will still be computed, since their lower and upper bounds do not immediately put them out of range). The traversal will be pruned when encountering a pair represented by dotted lines as their BCCP and the BCCP of their descendants will be out of range.

in Figure 17a. We prune the traversal when $d_{\max}(A, B) < \rho_{lo}$, or when $d(A, B) \geq \rho_{hi}$, in which case BCCP$(A, B)$ (as well as those of its recursive calls on descendant nodes) will be outside of the range. An example is shown in Figure 17b. In addition, we also prune the traversal if $A$ and $B$ are already connected in the MST, as an edge between $A$ and $B$ will not be part of the MST.

We evaluate MemoGFK in Section 5.8. We also use the memory optimization for HDBSCAN*, which will be described next.

### HDBSCAN*

**Baseline.** Inspired by a sequential approximate algorithm to solve the OPTICS problem by Gan and Tao [105], we modified and parallelized their algorithm to compute the exact HDBSCAN*as our baseline. First, we perform $k$-NN queries using Euclidean distance with $k =$ minPts to compute the core distances. Gan and Tao's

original algorithm creates a mutual reachability graph of size $O(n \cdot \mathsf{minPts}^2)$, using an approximate notion of BCCP between each WSPD pair, and then computes its MST using Prim's algorithm. Our exact algorithm parallelizes their algorithm, and instead uses the exact BCCP* computations based on the mutual reachability distance to form the mutual reachability graph. In addition, we also compute the MST on the generated edges using the MemoGFK optimization described in Section 5.2. Summed across all well-separated pairs, the BCCP computations take quadratic work and constant depth. Therefore, our baseline algorithm takes $O(n^2)$ work and $O(\log^2 n)$ depth, and computes the exact HDBSCAN*. In Section 5.6 of the Appendix, we also describe a work-efficient parallel approximate algorithm based on [105].

**Improved Algorithm.** We present a more space-efficient algorithm that is faster in practice by using a new definition of well-separation for the WSPD for HDBSCAN*. We denote the maximum and minimum core distances of the points in node $A$ as $\mathsf{cd}_{\max}(A)$ and $\mathsf{cd}_{\min}(A)$, respectively. Consider a pair $(A, B)$ in the WSPD. We define $A$ and $B$ to be ***geometrically-separated*** if $d(A, B) \geq \max\{A_{\mathsf{diam}}, B_{\mathsf{diam}}\}$ and ***mutually-unreachable*** if $\max\{d(A, B), \mathsf{cd}_{\min}(A), \mathsf{cd}_{\min}(B)\} \geq \max\{A_{\mathsf{diam}}, B_{\mathsf{diam}}, \mathsf{cd}_{\max}(A), \mathsf{cd}_{\max}(B)\}$. We consider $A$ and $B$ to be well-separated if they are geometrically-separated, mutually-unreachable, or both. The original definition of well-separation only includes the first condition.

This leads to space savings because in algorithm 5, recursive calls to procedure FindPair$(A, B)$ on line 7 will not terminate until $A$ and $B$ are well-separated. Since our new definition is a disjunction between mutual-unreachability and geometric-separation, the calls to FindPair can terminate earlier, leading to fewer pairs generated. When constructing the mutual reachability subgraph to pass to MST, we add only a single edge between the BCCP* (BCCP with respect to mutual reachability distance) of each well-separated pair. With our new definition, the total number of edges generated is upper bounded by the size of the WSPD, which is $O(n)$ [60]. In contrast, Gan and Tao's approach generates $O(n \cdot \mathsf{minPts}^2)$ edges.

**Theorem 5.** *Under the new definition of well-separation, our algorithm computes an MST of the mutual reachability graph.*

*Proof.* Under our new definition, well-separation is defined as the disjunction between being geometrically-separated and mutually-unreachable. We connect an edge between each well-separated pair $(A, B)$ with the mutual-reachability distance $\max\{d(u^*, v^*), \mathsf{cd}(u^*), \mathsf{cd}(v^*)\}$ as the edge weight, where $u^* \in A$, $v^* \in B$, and $(u^*, v^*)$ is the BCCP* of $(A, B)$. We overload the notation BCCP*$(A, B)$ to also denote the mutual-reachability distance of $(u^*, v^*)$.

Consider the point set $P_{\mathsf{root}}$, which is contained in the root node of the tree associated with its WSPD. Let $T$ be the MST of the full mutual reachability graph $G_{MR}$. Let $T'$ be the MST of the mutual reachability subgraph $G'_{MR}$, computed by connecting the BCCP* of each well-separated pair. To ensure that $T'$ produces the correct HDBSCAN*clustering, we prove that it has the same weight as $T$—in other words, $T'$ is a valid MST of $G_{MR}$.

We prove the optimality of $T'$ by induction on each tree node $P$. Since the WSPD is hierarchical, each node $P$ also has a valid WSPD consisting of a subset of pairs of the WSPD of $P_{\mathsf{root}}$. Let $(u, v)$ be an edge in $T$. There exists an edge $(u', v') \in T'$ that connects the same two components as in $T$ if we were to remove $(u, v)$. We call $(u', v')$ the ***replacement*** of $(u, v)$, which is ***optimal*** if $w(u', v') = w(u, v)$. Let $T_P$ and $T'_P$ be subgraphs of $T$ and $T'$, respectively, containing points in $P$, but not necessarily spanning $P$. We inductively hypothesize that all edges of $T'_P$ are optimal. In the base case, a singleton tree node $P$ satisfies the hypothesis by having no edges.

Now consider any node $P$ and edge $(u, v) \in T_P$. The children of $P$ are optimal by our inductive hypothesis. We prove that the edges connecting the children of $P$ are optimal. Points $u$ and $v$ must be from a well-separated pair $(A, B)$, where $A$ and $B$ are children of $P$ in the WSPD hierarchy. Let $U$ and $V$ be a partition of $P$ formed by a cut in $T_P$ that separates point pair $(u, v)$, where $u \in U$ and $v \in V$. We want to prove that the replacement of $(u, v)$ in $T'_P$ is optimal.

We now discuss the first scenario of the proof, shown in Figure 18a, where the replacement edge between $U$ and $V$ is $(u', v') = \mathsf{BCCP}^*(A, B) = (u^*, v^*)$, and we assume without loss of generality that $u' \in A \cap U$ and $v' \in B \cap V$. Since $(u, v)$ is the closest pair of points connecting $U$ and $V$ by the cut property, then $(u', v')$, the BCCP* of $(A, B)$, must be optimal; otherwise, $(u, v)$ has smaller weight than $\mathsf{BCCP}^*(A, B)$, which is a contradiction. This scenario easily generalizes to the case where $A$ and $B$ happen to be completely within $U$ and $V$, respectively.

We now discuss the second scenario, shown in Figure 18b, where $\mathsf{BCCP}^*(A, B) = (u^*, v^*)$ is internal to either $U$ or $V$. We assume without loss of generality that $u^* \in A \cap V$ and $v^* \in B \cap V$, and that $U$ and $V$ are connected by some intra-node edge $(u', v')$ of $A$ in $T'_P$. We want to prove that $(u', v')$ is an optimal replacement edge. We consider two cases based on the relationship between $A$ and $B$ under our new definition of well-separation.

**Case 1.** Nodes $A$ and $B$ are mutually-unreachable, and may or may not be geometrically-separated. The weight of $(u', v')$ is $\max\{d(u', v'), \mathsf{cd}(u'), \mathsf{cd}(v')\} \leq \max\{A_{\mathsf{diam}}, \mathsf{cd}_{\max}(A)\}$. Consider the BCCP* pair $(u^*, v^*)$ between $A$ and $B$. Based on the fact that $A$ and

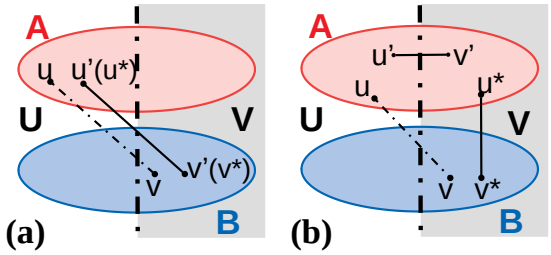



Figure 18: In this figure, we show the two proof cases for HDBSCAN*. We use an oval to represent each node in the WSPD, and solid black dots to represent data points. We represent the partition of the space to $U$ and $V$ using a cut represented by a dotted line.

$B$ are mutually-unreachable, we have

$$\begin{aligned}\mathsf{BCCP}^*(A,B) &= \max\{d(u^*,v^*),\mathsf{cd}(u^*),\mathsf{cd}(v^*)\}\\ &\geq \max\{d(A,B),\mathsf{cd}_{\min}(A),\mathsf{cd}_{\min}(B)\}\\ &\geq \max\{A_{\mathsf{diam}},B_{\mathsf{diam}},\mathsf{cd}_{\max}(A),\mathsf{cd}_{\max}(B)\}\\ &\geq \max\{A_{\mathsf{diam}},\mathsf{cd}_{\max}(A)\},\end{aligned}$$

where the inequality from the second to the third line above comes from the definition of mutual-unreachability. Therefore, $w(u',v')$ is not larger than $\mathsf{BCCP}^*(A,B) = w(u^*,v^*)$, and by definition of $\mathsf{BCCP}^*$, $w(u^*,v^*)$ is not larger than $w(u,v)$. Hence, $w(u',v')$ is not larger than $w(u,v)$. On the other hand, $w(u',v')$ is not smaller than $w(u,v)$, since otherwise we could form a spanning tree with a smaller weight than $T_P$, contradicting the fact that it is an MST. Thus, $(u',v')$ is optimal.

**Case 2.** Nodes $A$ and $B$ are geometrically-separated and not mutually-unreachable. By the definition of $\mathsf{BCCP}^*$, we know that $w(u^*,v^*) \leq w(u,v)$, which implies

$$\begin{aligned}\max\{\mathsf{cd}(u^*),\mathsf{cd}(v^*),d(u^*,v^*)\} &\leq \max\{\mathsf{cd}(u),\mathsf{cd}(v),d(u,v)\}\\ \max\{\mathsf{cd}(u^*),\mathsf{cd}(u),d(u,u^*)\} &\leq \max\{\mathsf{cd}(u),\mathsf{cd}(v),d(u,v)\}.\end{aligned}$$

To obtain the second inequality above from the first, we replace $\mathsf{cd}(v^*)$ on the left-hand side with $\mathsf{cd}(u)$, since $\mathsf{cd}(u)$ is also on the right-hand side; we also replace $d(u^*,v^*)$ with $d(u,u^*)$ because of the geometric separation of $A$ and $B$. Since $(u',v')$ is the lightest $\mathsf{BCCP}^*$ edge of some well-separated pair in $A$, $\max\{\mathsf{cd}(u'),\mathsf{cd}(v'),$

$d(u', v')\} \leq \max\{\mathsf{cd}(u), \mathsf{cd}(u^*), d(u, u^*)\}$. We then have

$$\max\{\mathsf{cd}(u'), \mathsf{cd}(v'), d(u', v')\} \ \leq \max\{\mathsf{cd}(u), \mathsf{cd}(v), d(u, v)\}.$$

This implies that $w(u', v')$ is not larger than $w(u, v)$. Since $(u, v)$ is an edge of MST $T_P$, the weight of the replacement edge $w(u', v')$ is also not smaller than $w(u, v)$, and hence $(u', v')$ is optimal.

Case 1 and 2 combined prove the optimality of replacement edges in the second scenario. Considering both scenarios, we have shown that each replacement edge in $T'_p$ connecting the children of $P$ is optimal, which proves the inductive hypothesis. Applying the inductive hypothesis to $P_{\mathsf{root}}$ completes the proof. □

Our algorithm achieves the following bounds.

**Theorem 6.** *Given a set of $n$ points, we can compute the MST on the mutual reachability graph in $O(n^2)$ work, $O(\log^2 n)$ depth, and $O(n \cdot \mathsf{minPts})$ space.*

*Proof.* Compared to the cost of GFK for EMST, GFK for HDBSCAN*has the additional cost of computing the core distances, which takes $O(\mathsf{minPts} \cdot n \log n)$ work and $O(\log n)$ depth using $k$-NN [57]. With our new definition of well-separation, the WSPD computation will only terminate earlier than in the original definition, and so the bounds that we showed for EMST above still hold. The new WSPD definition also gives an $O(n)$ space bound for the well-separated pairs. The space usage of the $k$-NN computation is $O(n \cdot \mathsf{minPts})$, which dominates the space usage. Overall, this gives $O(n^2)$ work, $O(\log^2 n)$ depth, and $O(n \cdot \mathsf{minPts})$ space. □

Our algorithm gives a clear improvement in space usage over the naive approach of computing an MST from the mutual reachability graph, which takes $O(n^2)$ space, and our parallelization of the exact version of Gan and Tao's algorithm, which takes $O(n \cdot \mathsf{minPts}^2)$ space. We will also see that the smaller memory footprint of this algorithm leads to better performance in practice.

**Implementation.** We implement three algorithms for HDBSCAN*: a parallel version of the approximate algorithm based on Gan and Tao [105], a parallel exact algorithm based on Gan and Tao, and our space-efficient algorithm from Section 5.2. Our implementations all use Kruskal's algorithm for MST and use the memory optimization introduced for MemoGFK in Section 5.2. For our space-efficient algorithm, we modify the WSPD and MemoGFK algorithm to use our new definition of well-separation.

## 5.3 Dendrogram and Reachability Plot

We present a new parallel algorithm for generating a dendrogram and reachability plot, given an unrooted tree with edge weights. Our algorithm can be used for single-linkage clustering [113] by passing the EMST as input, as well as for generating the HDBSCAN*dendrogram and reachability plot (refer to Section 3.1 for definitions). In addition, our dendrogram algorithm can be used in efficiently generating hierarchical clusters using other linkage criteria (e.g., [256, 252, 179]).

Sequentially, the dendrogram can be generated in a bottom-up (agglomerative) fashion by sorting the edges by weight and processing the edges in increasing order of weight [171, 85, 176, 113, 126]. Initially, all points are assigned their own clusters. Each edge merges the clusters of its two endpoints, if they are in different clusters, using a union-find data structure. The order of the merges forms a tree structure, which is the dendrogram. This takes $O(n \log n)$ work, but has little parallelism since the edges need to be processed one at a time. For HDBSCAN*, we can generate the reachability plot directly from the input tree by running Prim's algorithm on the tree edges starting from an arbitrary vertex [22]. This approach takes $O(n \log n)$ work and is also hard to parallelize efficiently, since Prim's algorithm is inherently sequential.

Our new parallel algorithm uses a top-down approach to generate the dendrogram and reachability plot given a weighted tree. Our algorithm takes $O(n \log n)$ expected work and $O(\log^2 n \log \log n)$ depth with high probability, and hence is work-efficient.

### Ordered Dendrogram

We discuss the relationship between the dendrogram and reachability plot, which are both used in HDBSCAN*. It is known [202] that a reachability plot can be converted into a dendrogram using a linear-work algorithm for Cartesian tree construction [103], which can be parallelized [212]. However, converting in the other direction, which is what we need, is more challenging because the children in dendrogram nodes are unordered, and can correspond to many possible sequences, only one of which corresponds to the traversal order in Prim's algorithm that defines the reachability plot.

Therefore, for a specific starting point $s$, we define the ***ordered dendrogram*** of $s$, which is a dendrogram where its in-order traversal corresponds to the reachability plot starting at point $s$. With this definition, there is a one-to-one correspondence between a ordered dendrogram and a reachability plot, and there are a total of $n$ possible ordered dendrograms and reachability plots for an input of size $n$. Then, a reachability plot is just the in-order traversal of the leaves of an ordered dendrogram,

and an ordered dendrogram is the corresponding Cartesian tree for the reachability plot.

### A Novel Top-Down Algorithm

We introduce a novel work-efficient parallel algorithm to compute a dendrogram, which can be modified to compute an ordered dendrogram and its corresponding reachability plot.

**Warm-up.** We first propose a simple top-down algorithm for constructing the dendrogram, which does not quite give us the desired work and depth bounds. We first generate an Euler tour on the input tree [138]. Then, we delete the heaviest edge, which can be found in linear work and $O(1)$ depth by checking all edges. By definition, this edge will be the root of the dendrogram, and removing this edge partitions the tree into two subtrees corresponding to the two children of the root. We then convert our original Euler tour into two Euler tours, one for each subtree, which can be done in constant work and depth by updating a few pointers. Next, we partition our list of edges into two lists, one for each subproblem. This can be done by applying list ranking on each Euler tour to determine appropriate offsets for each edge in a new array associated with its subproblem. This step takes linear work and has $O(\log n)$ depth [138]. Finally, we solve the two subproblems recursively.

Although the algorithm is simple, there is no guarantee that the subproblems are of equal size. In the worst case, one of the subproblems could contain all but one edges (e.g., if the tree is a path with edge weights in increasing order), and the algorithm would require $O(n)$ levels of recursion. The total work would then be $O(n^2)$ and depth would be $O(n \log n)$, which is clearly undesirable.

**An algorithm with $O(\log n)$ levels of recursion.** We now describe a top-down approach that guarantees $O(\log n)$ levels of recursion. We define the ***heavy edges*** of a tree with $n$ edges to be the $n/2$ (or any constant fraction of $n$) heaviest edges and the ***light edges*** of a tree to be the remaining edges. Rather than using a single edge to partition the tree, we use the $n/2$ heaviest edges to partition the tree. The heavy edges correspond to the part of the dendrogram closer to the root, which we refer to as the *top* part of the dendrogram, and the light edges correspond to subtrees of the top part of the dendrogram. Therefore, we can recursively construct the dendrogram on the heavy edges and the dendrograms on the light edges in parallel. Then, we insert the roots of the dendrograms for the light edges into the leaf nodes of the heavy-edge dendrogram. The base case is when there is a single edge, from which we can trivially generate a dendrogram.

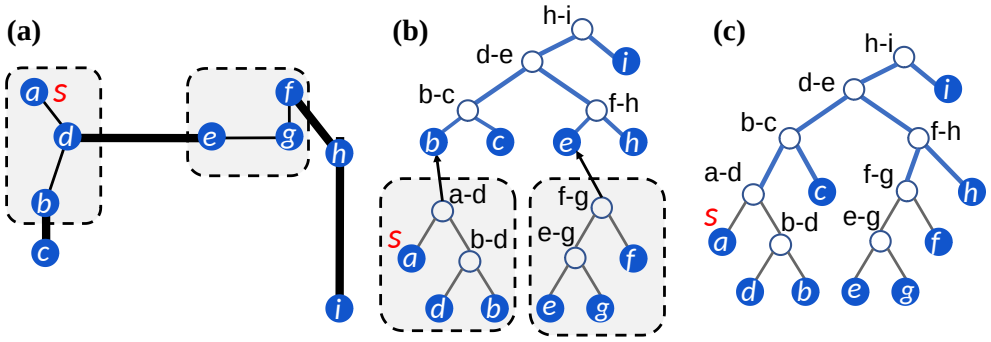



Figure 19: An example of the dendrogram construction algorithm on the tree from fig. 4. The input tree is shown in (a). The 4 heavy edges are in bold. We have three subproblems—one for the heavy edges and two for the light edges. The dendrograms for the subproblems are generated recursively, as shown in (b). The edge labeled on an internal node is the edge whose removal splits a cluster into the two clusters represented by its children. As shown in (c), we insert the roots of the dendrograms for the light edges at the corresponding leaf nodes of the heavy-edge dendrogram. For the ordered dendrogram, the in-order traversal of the leaves corresponds to the reachability plot shown in fig. 4 when the starting point $s = a$.

An example is shown in Figure 19. We first construct the Euler tour of the input tree (Figure 19a). Then, we find the median edge based on edge weight, separate the heavy and light edges and compact them into a heavy-edge subproblem and multiple light-edge subproblems. For the subproblems, we construct their Euler tours by adjusting pointers, and mark the position of each light-edge subproblem in the heavy-edge subproblem where it is detached. Then, recursively and in parallel, we compute the dendrograms for each subproblem (Figure 19b). After that, we insert the roots of the light-edge dendrograms to the appropriate leaf nodes in the heavy-edge dendrogram, as marked earlier (Figure 19c).

fig. 19 shows how this algorithm applies to the input in fig. 4 with source vertex $a$. The four heaviest edges $(b, c)$, $(d, e)$, $(f, h)$, and $(h, i)$ divide the tree into two light subproblems, consisting of $\{(a, d), (d, b)\}$ and $\{(e, g), (g, f)\}$. The heavy edges form another subproblem. We mark vertices $b$ and $e$, where the light subproblems are detached. After constructing the dendrogram for the three subproblems, we insert the light dendrograms at leaf nodes $b$ and $e$, as shown in Figure 19b. It forms the correct dendrogram in Figure 19c.

We now describe the details of the steps to separate the subproblems and re-insert them into the final dendrogram.

**Subproblem Finding.** To find the position in the heavy-edge dendrogram to insert

a light-edge dendrogram at, every light-edge subproblem will be associated with a unique heavy edge. The dendrogram of the light-edge subproblem will eventually connect to the corresponding leaf node in the heavy-edge dendrogram associated with it. We first explain how to separate the heavy-edge subproblem and the light-edge subproblems.

First, we compute the unweighted distance from every point to the starting point $s$ in the tree, and we refer to them as the ***vertex distances***. For the ordered dendrogram, $s$ is the starting point of the reachability plot, whereas $s$ can be an arbitrary vertex if the ordering property is not needed. We compute the vertex distances by performing list ranking on the tree's Euler tour rooted at $s$. These distances can be computed by labeling each downward edge (away from $s$) in the tree with a value of 1 and each upward edge (towards $s$) in the tree with a value of $-1$, and running list ranking on the edges. The vertex distances are computed only once.

We then identify the light-edge subproblems in parallel by using the vertex distances. For each light edge $(u, v)$, we find an adjacent edge $(w, u)$ such that $w$ has smaller vertex distance than both $u$ and $v$. We call $(w, u)$ the ***predecessor edge*** of $(u, v)$. Each edge can only have one predecessor edge (an edge adjacent to $s$ will choose itself as the predecessor). In a light-edge subproblem not containing the starting vertex $s$, the predecessor of each light edge will either be a light edge in the same light-edge subproblem, or a heavy edge. The edges in each light-edge subproblem will form a subtree based on the pointers to predecessor edges. We can obtain the Euler tour of each light-edge subproblem by adjusting pointers of the original Euler tour. The next step is to run list ranking to propagate a unique label (the root's label of the subproblem subtree) of each light-edge subproblem to all edges in the same subproblem. To create the Euler tour for the heavy subproblem, we contract the subtrees for the light-edge subproblems: for each light-edge subproblem, we map its leaves to its root using a parallel hash table. Now each heavy edge adjacent to a light-edge subproblem leaf can connect to the heavy edge adjacent to the light-edge subproblem root by looking it up in the hash table. The Euler tour for the heavy-edge subproblem can now be constructed by adjusting pointers. We assign the label of the heavy-edge subproblem root to all of the heavy edges in parallel. Then, we semisort the labeled edges to group edges of the same light-edge subproblems and the heavy-edge subproblem. Finally, we recursively compute the dendrograms on the light-edge subproblems and the heavy-edge subproblem. In the end, we connect the light-edge dendrogram for each subproblem to the heavy-edge dendrogram leaf node corresponding to the shared endpoint between the light-edge subproblem and its unique heavy predecessor edge. For the light-edge subproblem containing the

starting point $s$, we simply insert its light-edge dendrogram into the left-most leaf node of the heavy-edge dendrogram.

Consider Figure 19a. The heavy-edge subproblem contains edges $\{(b,c),(d,e),(f,h),(h,i)\}$, and its dendrogram is shown in Figure 19b. For the light-edge subproblem $\{(e,g),(g,f)\}$, $(e,g)$ has heavy predecessor edge $(d,e)$, and $(g,f)$ has light predecessor edge $(e,g)$. The unique heavy edge associated with the light-edge subproblem is hence $(d,e)$, with which it shares vertex $e$. Hence, we insert the light-edge dendrogram for the subproblem into leaf node $e$ in the heavy-edge dendrogram, as shown in Figure 19b. The light-edge subproblem containing $\{(a,d),(d,b)\}$ contains the starting point $s = a$, and so we insert its dendrogram into the leftmost leaf node $b$ of the heavy-edge dendrogram, as shown in Figure 19b.

We first show that our algorithm correctly computes a dendrogram, and analyze its cost bounds (Theorem 7). Then, we describe and analyze additional steps needed to generate an ordered dendrogram and obtain a reachability plot from it (Theorem 8).

**Theorem 7.** *Given a weighted spanning tree with $n$ vertices, we can compute a dendrogram in $O(n \log n)$ expected work and $O(\log^2 n \log\log n)$ depth with high probability.*

*Proof.* We first prove that our algorithm correctly produces a dendrogram. In the base case, we have one edge $(u,v)$, and the algorithm produces a tree with a root representing $(u,v)$, and with $u$ and $v$ as children of the root, which is trivially a dendrogram. We now inductively hypothesize that recursive calls to our algorithm correctly produce dendrograms. The heavy subproblem recursively computes a top dendrogram consisting of all of the heavy edges, and the light subproblems form dendrograms consisting of light edges. We replace the leaf vertices in the top dendrogram associated with light subproblems by the roots of the dendrograms on light edges. Since the edges in the heavy subproblem are heavier than all edges in light subproblems, and are also ancestors of the light edges in the resulting tree, this gives a valid dendrogram.

We now analyze the cost of the algorithm. To generate the Euler tour at the beginning, we first sort the edges and create an adjacency list representation, which takes $O(n \log n)$ work and $O(\log n)$ depth [75]. Next, we root the tree, which can be done by list ranking on the Euler tour of the tree. Then, we compute the vertex distances to $s$ using another round of list ranking based on the rooted tree.

There are $O(\log n)$ recursive levels since the subproblem sizes are at most half of the original problem. We now show that each recursive level takes linear expected work and polylogarithmic depth with high probability. Note that we cannot afford

to sort the edges on every recursive level, since that would take $O(n \log n)$ work per level. However, we only need to know which edges are heavy and which are light, and so we can use parallel selection [138] to find the median and partition the edges into two sets. This takes $O(n)$ work and $O(\log n \log\log n)$ depth. Identifying predecessor edges takes a total of $O(n)$ work and $O(1)$ depth: we find and record for each vertex its edge where the other endpoint has a smaller vertex distance than it (using WriteMin); then, the predecessor of each edge is found by checking the recorded edge for its endpoint with smaller vertex distance. We then use list ranking to assign labels to each subproblem, which takes $O(n)$ work and $O(\log n)$ depth [138]. The hash table operations to contract and look up the light-edge subproblems cost $O(n)$ work and $O(\log n)$ depth with high probability. The semisort to group the subproblems takes $O(n)$ expected work and $O(\log n)$ depth with high probability. Attaching the light-edge dendrograms to the heavy-edge dendrogram takes $O(n)$ work and $O(1)$ depth across all subproblems. Multiplying the bounds by the number of levels of recursion proves the theorem. □

**Theorem 8.** *Given a starting vertex $s$, we can generate an ordered dendrogram and reachability plot in the same cost bounds as in Theorem 7.*

*Proof.* We have computed the vertex distances of all vertices from $s$. When generating the ordered dendrogram and constructing each internal node of the dendrogram corresponding to an edge $(u, v)$, and without loss of generality let $u$ have a smaller vertex distance than $v$, our algorithm puts the result of the subproblem attached to $u$ in the left subtree, and that of $v$ in the right subtree. This additional comparison does not increase the work and depth of our algorithm.

Our algorithm recursively builds ordered dendrograms on the heavy-edge subproblem and on each of the light-edge subproblems, which we assume to be correct by induction. The base case is a single edge $(u, v)$, and without loss of generality let $u$ have a smaller vertex distance than $v$. Then, the dendrogram will contain a root node representing edge $(u, v)$, with $u$ as its left child and $v$ as its right child. Prim's algorithm would visit $u$ before $v$, and so is the in-order traversal of the dendrogram, so it is an ordered dendrogram.

We now argue that the way that light-edge dendrograms are attached to the leaves of the heavy-edge dendrogram correctly produces an ordered dendrogram. First, consider a light-edge subproblem that contains the source vertex $s$. In this case, its dendrogram is attached as the leftmost leaf of the heavy-edge dendrogram, and will be the first to be traversed in the in-order traversal. The vertices in the light-edge subproblem form a connected component $A$. They will be traversed before

any other vertices in Prim's algorithm because all incident edges that leave $A$ are heavy edges, and thus are heavier than any edge in $A$. Therefore, vertices outside of $A$ can only be visited after all vertices in $A$ have been visited, which correctly corresponds to the in-order traversal.

Next, we consider the case where the light-edge subproblem does not contain $s$. Let $(u, v)$ be the predecessor edge of the light-edge subproblem, and let $A$ be the component containing the edges in the light-edge subproblem ($v$ is a vertex in $A$). Now, consider a different light-edge subproblem that does not contain $s$, whose predecessor edge is $(x, y)$, and let $B$ be the component containing the edges in this subproblem ($y$ is a vertex in $B$). By construction, we know that $A$ is in the right subtree of the dendrogram node corresponding to edge $(u, v)$ and $B$ is in the right subtree of node corresponding to $(x, y)$. The ordering between $A$ and $B$ is correct as long as they are on different sides of either node $(u, v)$ or node $(x, y)$. For example, if $B$ is in the left subtree of node $(u, v)$, then its vertices appear before $A$ in the in-order traversal of the dendrogram. By the inductive hypothesis on the heavy-edge subproblem, in Prim's order, $B$ will be traversed before $(u, v)$, and $(u, v)$ is traversed before $A$. We can apply a similar argument to all other cases where $A$ and $B$ are on different sides of either node $(u, v)$ or node $(x, y)$.

We are concerned with the case where $A$ and $B$ are both in the right subtrees of the nodes representing their predecessor edges. We prove by contradiction that this cannot happen. Without loss of generality, suppose node $(x, y)$ is in the right subtree of node $(u, v)$, and let both $A$ and $B$ be in the right subtree of $(x, y)$. There exists a lowest common ancestor (LCA) node $(x', y')$ of $A$ and $B$. $(x', y')$ must be a heavy edge in the right subtree of $(x, y)$. By properties of the LCA, $A$ and $B$ are in different subtrees of node $(x', y')$. Without loss of generality, let $A$ be in the left subtree. Now consider edge $(x', y')$ in the tree. By the inductive hypothesis on the heavy-edge dendrogram, in Prim's traversal order, we must first visit the leaf that $A$ attaches to (and hence $A$) before visiting $(x', y')$, which must be visited before the leaf that $B$ attaches to (and hence $B$). On the other hand, edge $(x, y)$ is also along the same path since it is the predecessor of $B$. Thus, we must either have $(x', y')$ in $(x, y)$'s left subtree or $(x, y)$ in $(x', y')$'s right subtree, which is a contradiction to $(x', y')$ being in the right subtree of $(x, y)$.

We have shown that given any two light-edge subproblems, their relative ordering after being attached to the heavy-edge dendrogram is correct. Since the heavy-edge dendrogram is an ordered dendrogram by induction, the order in which the light-edge subproblems are traversed is correct. Furthermore, each light-edge subproblem generates an ordered dendrogram by induction. Therefore, the overall dendrogram is an ordered dendrogram.

Once the ordered dendrogram is computed, we use list ranking to perform an in-order traversal on the Euler tour of the dendrogram to give each node a rank, and write them out in order. We then filter out the non-leaf nodes to to obtain the reachability plot. Both list ranking and filtering take $O(n)$ work and $O(\log n)$ depth. □

**Implementation.** In our implementation, we simplify the process of finding the subproblems by using a sequential procedure rather than performing parallel list ranking, because in most cases parallelizing over the different subproblems already provides sufficient parallelism. We set the number of heavy edges to $n/10$, which we found to give better performance in practice, and also preserves the theoretical bounds. We switch to the sequential dendrogram construction algorithm when the problem size falls below $n/2$.

## 5.4 Parallel EMST and HDBSCAN* in 2D

### Parallel EMST in 2D

The ***Delaunay triangulation*** on a set of points in 2D contains triangles among every triple of points $p_1$, $p_2$, and $p_3$ such that there are no other points inside the circumcircle defined by $p_1$, $p_2$, and $p_3$ [84].

In two dimensions, Shamos and Hoey [210] show that the EMST can be computed by computing an MST on the Delaunay triangulation of the points. Parallel Delauny triangulation can be computed in $O(n \log n)$ work and $O(\log n)$ depth [198], and has $O(n)$ edges, and so the MST computation requires the same work and depth. We provide an implemenation of this algorithm using the parallel Delaunay triangulation and parallel implementation of Kruskal's algorithm from the Problem Based Benchmark Suite [43].

### Parallel HDBSCAN*in 2D

The ***ordinary Voronoi diagram*** is a planar subdivision of a space where points in each cell share the same nearest neighbor. The ***$k$-order Voronoi Diagram*** is a generalization of the ordinary Voronoi diagram, where points in each cell share the same $k$-nearest neighbors [29]. A ***$k$-order edge*** is a closely related concept, and defined to be an edge where there exists a circle through the two edge endpoints, such that there are at most $k$ points inside the circle [116].

De Berg et al. [85] show that in two dimensions, the MST on the mutual reachability graph can be computed in $O(n \log n)$ work. Their algorithm computes an MST

on a graph containing the $k$-order edges, where $k = \mathsf{minPts} - 3$, and where the edges are weighted by the mutual reachability distances between the two endpoints. They prove that the MST returned is an MST on the mutual reachability graph. In this section, we extend their result to the parallel setting.

To parallelize the algorithm, we need to compute the $k$-order edges of the points in parallel. This can be done by first computing the $(k+1)$-order Voronoi diagram, and then converting the edges in the Voronoi diagram to $k$-order edges, as shown by Gudmundsson et al. [116]. Specifically, we convert each Voronoi edge into a $k$-order edge by connecting the two points that induce the two cells sharing the Voronoi edge.

Meyerhenke [173] shows that the family of the order-$j$ Voronoi diagrams for all $1 \leq j \leq k$ can be computed in $O(k^2 n \log n)$ work and $O(k \log^2 n)$ depth. The algorithm works by first computing the ordinary Voronoi diagram on the input points. Then for each Voronoi cell, it computes the ordinary Voronoi diagram again on the points that induce the neighboring cells. This ordinary Voronoi diagram divides the Voronoi cell into multiple subcells, each of which corresponds to a cell in the Voronoi diagram of one higher order. This process is repeated until obtaining the order-$k$ Voronoi diagram. Lee [155] proves that the number of $k$-order edges is $O(nk)$, and so we can run parallel MST on these edges in $O(nk \log n)$ work and $O(\log n)$ depth. This gives us the following theorem.

**Theorem 9.** *Given a set of $n$ points in two dimensions, we can compute the MST on the mutual reachability graph in $O(\mathsf{minPts}^2 \cdot n \log n)$ work and $O(\mathsf{minPts} \cdot \log^2 n)$ depth.*

For computing the ordinary Voronoi diagrams on each step of Meyerhenke's algorithm, we use the parallel Delaunay triangulation implementation from the Problem Based Benchmark Suite [43] and take the dual of the resulting triangulation. However, we found it to be significantly slower than our other methods due to high work of the Voronoi diagram computations.

## 5.5 Subquadratic-work Parallel EMST

Callahan and Kosaraju's algorithm [58] first constructs a fair-split tree $T$ and its associated WSPD in $O(n \log n)$ work and $O(\log n)$ depth [57], which is improved from a previous version with $O(n \log n)$ work and $O(\log^2 n)$ depth [60]. Then, the algorithm runs Boruvka's steps for $\lceil \log_2 n \rceil$ rounds. In particular, in each round, the algorithm finds the lightest outgoing edges only for the components with size at most $2^{i+1}$, and merges the components connected by these edges. To do so, the algorithm constructs for every component a set of candidate points that contains the nearest

point outside the component. The algorithm searches for the candidates top-down on $T$, and maintains for each node in the tree, a list of all the component s that can have candidates in the subtree of that node. They ensure the size of each list is $O(1)$ using the WSPD in a manner identical to the all-nearest-neighbors algorithm of [60]. In this process, they push the lists down to the leaves of $T$, so that the candidates corresponding to a component will be the leaves that contain that component in their lists.

Let $P_j$ be the set of candidates for the $j$'th component. $P_j$ is split into $\lceil |P_j|/2^{i+1} \rceil$ subsets of size at most $2^{i+1}$ each, and the BCCP is found between each subset and the $j$'th component. At round $i$, there are $n/2^i$ components, and the BCCP routine is invoked $\sum_{j=1}^{n/2^i} \lceil |P_j|/2^{i+1} \rceil = O(n/2^i)$ times, each with size at most $2^{i+1}$. Therefore, the work for BCCP on each round is $O((n/2^i)T_d(2^{i+1}, 2^{i+1}))$. Since $T_d$ is at least linear, this dominates the work for each phase. The total work for BCCP computations is $O(T_d(n, n) \log n)$.

In our parallel algorithm, on each round, we perform both the candidate listing step and BCCP computations in parallel. Listing candidates for all components can be computed in parallel given a WSPD. In particular, this uses the top-down computation used for the all-nearest-neighbor search, parallelized using rake and compress operations [60], and takes logarithmic depth. Wang et al. [239] show that BCCP can be computed in parallel in $O(n^{2-(2/(\lceil d/2 \rceil+1))+\delta})$ expected work and $O(\log^2 n \log^* n)$ depth *whp*. Both the work and depth at each round is therefore dominated by computing the BCCPs. With $O(\log n)$ rounds, this results in $O(T_d(n, n) \log n)$ expected work and $O(\log^3 n \log^* n)$ depth *whp*, where $T_d(n, n) = O(n^{2-(2/(\lceil d/2 \rceil+1))+\delta})$. This is also the work and depth of the overall EMST algorithm, as WSPD construction only contributes lower-order terms to the complexity.

**Theorem 10.** *We can compute the EMST on a set of $n$ points in $d$ dimensions in $O(n^{2-(2/(\lceil d/2 \rceil+1))+\delta} \log n)$ expected work and polylogarithmic depth* whp.

## 5.6 Parallel Approximate OPTICS

**Parallel Algorithm.** Gan and Tao [105] propose a sequential algorithm to solve the approximate OPTICS problem, defined in Lemma 4.2 of their paper [105] (this also gives an approximation to HDBSCAN*). The algorithm takes in an additional parameter $\rho \geq 0$, which is related to the approximation factor. The algorithm makes use of the WSPD and uses $O(n \cdot \mathsf{minPts}^2)$ space, with the separation constant $s = \sqrt{8/\rho}$. They construct a base graph by adding $O(\mathsf{minPts}^2)$ edges between each well-separated pair, and then compute an MST on the resulting graph. Their

algorithm takes $O(n \log n)$ work (where the dominant cost is computing the WSPD). We observe that their algorithm can be parallelized by plugging in parallel WSPD and MST routines, resulting in an $O(n \log n)$ work and $O(\log^2 n)$ depth algorithm.

In parallel for all well-separated pairs, we compute an approximation to the BCCP for each pair. The algorithm uses the rake-and-compress algorithm of Callahan and Kosaraju [60] to obtain a set of coordinates used for approximation for every subset of the decomposition tree, taking $O(n)$ work and $O(\log n)$ depth. Then, for every pair in parallel, our algorithm computes the approximate BCCP in constant work and depth, similar to the sequential algorithm described in [58]. Overall, this step takes linear work and $O(\log n)$ depth [58, 57].

Similarly to Gan and Tao, we call the pair of points in the approximate BCCP of each pair the ***representative points***. For each well-separated pair $(A, B)$, there are four cases for generating edges between $A$ and $B$: (a) if $|A| < \mathsf{minPts}$ and $|B| < \mathsf{minPts}$, then all pairs of points between $A$ and $B$ are connected; (b) if $|A| \geq \mathsf{minPts}$ and $|B| < \mathsf{minPts}$, then the representative point of $A$ is connected to all points in $B$; (c) if $|A| < \mathsf{minPts}$ and $|B| \geq \mathsf{minPts}$, then the representative point of $B$ is connected to all points in $A$; and (d) if $|A| \geq \mathsf{minPts}$ and $|B| \geq \mathsf{minPts}$, then only the representative points are connected. The weight of the edges are

$$w(u, v) = \max\{\mathsf{cd}(u), \mathsf{cd}(v), \frac{d(u, v)}{1 + \rho}\}$$

given representative points $u$ and $v$. In our implementation, we simplify the approximate BCCP by simply picking a random pair of points from each well-separated pair, and also use the parallel MST algorithm introduced in Section 5.2, which computes the approximate BCCP on the fly for well-separated pairs that are not yet connected. This will take $O(\mathsf{minPts}^2 \cdot n)$ work due to $O(\mathsf{minPts}^2)$ edges produced for each pair, and $O(\log^2 n)$ depth. This gives us theorem 11.

**Theorem 11.** *Given a set of $n$ points, we can compute the MST required for approximate OPTICS in $O(n \log n)$ work, $O(\log^2 n)$ depth, and $O(n \cdot \mathsf{minPts}^2)$ space.*

**Experimental Results.** We study the performance of our parallel implementation for the approximate OPTICS problem, which we call ***OPTICS-GanTaoApprox***. It uses the MemoGFK optimization described in Section 5.2. We found that when run with a reasonable parameter of $\rho$ that leads to good clusters, OPTICS-GanTaoApprox is usually slower than our exact version of the algorithm (HDBSCAN*-GanTao, described in Section 5.2). The primary reason is that a reasonable $\rho$ value requires a high separation constant in the WSPD, which produces a very large number of well-separated pairs, leading to poor performance. In contrast, in the exact algorithm,

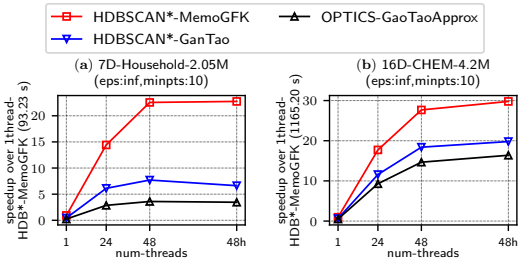



Figure 20: Speedup of HDBSCAN*MST implementations over the *best* serial baselines vs. thread count. The best serial baseline and its running time for each data set is shown on the $y$-axis label. "48h" on the $x$-axis refers to 48 cores with hyper-threading.

a small separation constant ($s = 2$) is sufficient for correctness. Figure 20 shows the speedups on two data sets for OPTICS-GanTaoApprox with $\rho = 0.125$ (corresponding to a separation constant of 8) compared with other methods. Across all of the data sets, we found OPTICS-GanTaoApprox to be slower than HDBSCAN*-GanTao by a factor of 1.00–1.96x, and slower than HDBSCAN*-MemoGFK by a factor of 1.72–7.48x.

## 5.7 Relationship between EMST and HDBSCAN*MST

We now show that for $\mathsf{minPts} \leq 3$, the EMST is always an MST of the HDBSCAN*base graph by having the same set of edges, but for higher values of minPtsit is possible that this is not the case. For example, fig. 21 gives an example where EMST is not an MST of the HDBSCAN*base graph when $\mathsf{minPts} = 4$.

**Theorem 12.** *An EMST is always an MST for the HDBSCAN*mutual reachability graph when* $\mathsf{minPts} \leq 3$.

*Proof.* For $\mathsf{minPts} \leq 2$, all edges in the HDBSCAN*mutual reachability graph have edge weights defined by Euclidean distances, and so the edge weights are identical. We now discuss the case when $\mathsf{minPts} = 3$.

Let $T$ be an EMST, and $T' \neq T$ be an MST in $G_{MR}$. We show that we can convert $T'$ to $T$ without changing the total weight. Consider any edge $(u, v) \in T$, but not in $T'$. If we add $(u, v)$ to $T'$, then we get a cycle $C$.

First, we show that $(u, v)$ cannot be the unique heaviest edge in $C$ under $d_m$. Recall that $\mathsf{cd}(p)$ is the core distance of a point $p$ and $d_m(p, q) = \max\{\mathsf{cd}(p), \mathsf{cd}(q), d(p, q)\}$. Assume by contradiction that $(u, v)$ is the unique heaviest edge in $C$ under $d_m$.

If $d_m(u,v) = d(u,v)$, then $(u,v)$ is also the unique heaviest edge in $C$ in the Euclidean complete graph, and so it cannot be in $T$, which is the EMST. This is a contradiction.

Now we consider the case where $d_m(u,v) > d(u,v)$. Without loss of generality, suppose that $d_m(u,v) = \mathsf{cd}(u)$. Then $v$ must be $u$'s unique nearest neighbor; otherwise, $d_m(u,v) = \mathsf{cd}(u) = d(u,v)$ because we have $\mathsf{minPts} = 3$. However, then all other points have larger distance to $u$ than $d(u,v)$, and $u$ must have an edge to one of these other points in the cycle $C$. Thus, $(u,v)$ cannot be the unique heaviest edge in $C$. This is a contradiction.

Now, given that $(u,v)$ is not the unique heaviest edge in $C$, we can replace one of the heaviest edges $e$ that is in $C$, but not in $T$, with $(u,v)$, and obtain another MST in $G_{MR}$ with the same weight.

Below we show that there is always such an edge $e$ in $C$. We first argue that there must be some heaviest edge in $C$ that has its Euclidean distance as its weight in $G_{MR}$. Consider a heaviest edge $(a,b)$ in $C$, and without loss of generality, suppose that $d_m(a,b) = \mathsf{cd}(a)$. If $(a,b)$ does not have its Euclidean distance as its edge weight, then $b$ must be $a$'s unique nearest neighbor. Besides $(a,b)$, $a$ must be incident to another edge in $C$, which we denote as $(a,c)$. $d_m(a,c)$ must equal $d_m(a,b)$: we have $d_m(a,c) \geq \mathsf{cd}(a) = d_m(a,b)$ because $b$ is $a$'s unique nearest neighbor, but we also have $d_m(a,c) \leq d_m(a,b)$ because $(a,b)$ is a heaviest edge in $C$. Therefore, $d_m(a,c) = d_m(a,b)$, and $(a,c)$ is one of the heaviest edges in $C$ under $d_m$. Furthermore, $d_m(a,c) = d(a,c)$ because $d(a,c) \leq d_m(a,c)$ by definition and $d(a,c) \geq \mathsf{cd}(a) = d_m(a,b) = d_m(a,c)$ because $\mathsf{minPts} = 3$ and $b \neq c$ is $a$'s unique nearest neighbor. Thus, we have shown that $(a,c)$ is a heaviest edge in $C$ that has its Euclidean distance as its weight in $G_{MR}$.

All heaviest edges that have the Euclidean distance as their weight must also be the heaviest edges in $C$ in the Euclidean complete graph, and thus they cannot all be in the EMST $T$. Therefore, there must exist some heaviest edge $e \in C$ that is in $T'$ but not in $T$. We can always find such an edge in $T'$ and swap it with the edge $(u,v)$ in $T$ to make $T'$ share more edges with $T$, without changing the total weight of $T'$ in $G_{MR}$, as both edges are heaviest edges in $C$ under $d_m$. We can repeat this process until we obtain $T$. Therefore, $T$ is also an MST in $G_{MR}$. □

## 5.8 Experiments

**Environment.** We perform experiments on an Amazon EC2 instance with 2 × Intel Xeon Platinum 8275CL (3.00GHz) CPUs for a total of 48 cores with two-way hyper-

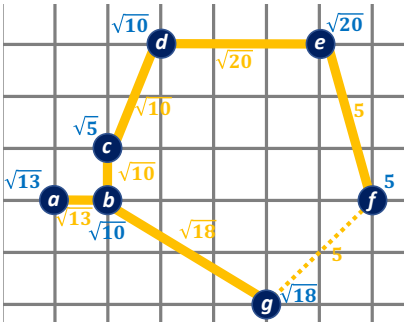



Figure 21: An example where the EMST is not an MST of HDBSCAN*base graph (call it MST*), when minPts = 4. The blue values are core distances of the points. The yellow values are weights of edges according to their mutual reachability distances. The solid edges form MST*. Both edge $(f, g)$ and $(e, f)$ are in the EMST, but cannot both be in MST* because they are the heaviest edges in the *g-b-c-d-e-f-g* cycle.

threading, and 192 GB of RAM. By default, we use all cores with hyper-threading. We use g++ compiler (version 7.4) with -O3 flag, and use Cilk for parallelism [136]. We do not report times for tests that exceed 3 hours.

We test the following implementations for EMST (note that the EMST problem does not include dendrogram generation):

- ***EMST-Naive***: The method of creating a graph with the BCCP edges from all well-separated pairs and then running MST on it.
- ***EMST-GFK***: The parallel GeoFilterKruskal algorithm described in Section 5.2 (Algorithm 6).
- ***EMST-MemoGFK***: The parallel GeoFilterKruskal algorithm with the memory optimization described in Section 5.2 (Algorithm 7).
- ***EMST-Delaunay***: The method of computing an MST on a Delaunay triangulation for 2D data sets described in section 5.4.

We test the following implementations for HDBSCAN*:

- ***HDBSCAN*-GanTao***: The modified algorithm of Gan and Tao for exact HDBSCAN*described in Section 5.2.
- ***HDBSCAN*-MemoGFK***: The HDBSCAN*algorithm using our new definition of well-separation described in Section 5.2.

Both ***HDBSCAN*-GanTao*** and ***HDBSCAN*-MemoGFK*** use the memory optimization described in Section 5.2. All HDBSCAN*running times include constructing an MST of the mutual reachability graph and computing the ordered dendrogram. We use a default value of $\mathsf{minPts} = 10$ (unless specified otherwise), which is also adopted in previous work [61, 171, 105].

Our algorithms are designed for multicores, as we found that multicores are able to process the largest data sets in the literature for these problems (machines with several terabytes of RAM can be rented at reasonable costs on the cloud). Our multicore implementations achieve significant speedups over existing implementations in both the multicore and distributed memory contexts.

**Data Sets.** We use the synthetic seed spreader data sets produced by the generator in [104]. It produces points generated by a random walk in a local neighborhood (***SS-varden***). We also use ***UniformFill*** that contains points distributed uniformly at random inside a bounding hypergrid with side length $\sqrt{n}$ where $n$ is the total number of points. We generated the synthetic data sets with 10 million points (unless specified otherwise) for dimensions $d = 2, 3, 5, 7$.

We use the following real-world data sets. ***GeoLife*** [259, 3] is a 3-dimensional data set with $24,876,978$ data points. This data set contains user location data, and is extremely skewed. ***Household*** [92, 6] is a 7-dimensional data set with $2,049,280$ points representing electricity consumption measurements in households. ***HT*** [135, 7] is a 10-dimensional data set with $928,991$ data points containing home sensor data. ***CHEM*** [97, 2] is a 16-dimensional data set with $4,208,261$ data points containing chemical sensor data. All of the data sets fit in the RAM of our machine.

**Comparison with Previous Implementations.** For EMST, we tested the sequential Dual-Tree Boruvka algorithm of March et al. [170] (part of `mlpack`), and our single-threaded EMST-MemoGFK times are 0.89–4.17 (2.44 on average) times faster. Raw running times for `mlpack` are presented in Table 4. We also tested McInnes and Healy's sequential HDBSCAN*implementation which is based on Dual-Tree Boruvka [171]. We were unable to run their code on our data sets with 10 million points in a reasonable amount of time. On a smaller data set with 1 million points (2D-SS-varden-1M), their code takes around 90 seconds to compute the MST and dendrogram, which is 10 times slower than our HDBSCAN*-MemoGFK implementation on a single thread, due to their code using Python and having fewer optimizations. We observed a similar trend on other data sets for McInnes and Healy's implementation.

The GFK algorithm implementation for EMST of [66] in the `Stann` library supports multicore execution using OpenMP. We found that, in parallel, their GFK

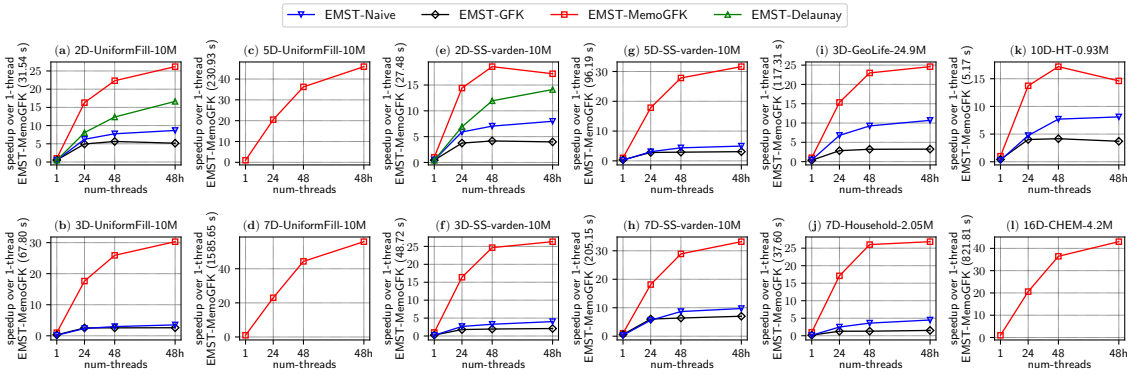



Figure 22: Speedup of EMST implementations over the *best* serial baselines vs. thread count. The best serial baseline and its running time for each data set is shown on the $y$-axis label. "48h" on the $x$-axis refers to 48 cores with hyper-threading.

implementation always runs much slower when using all 48 cores than running sequentially, and so we do not include their parallel running times in our experiments. In addition, our own sequential implementation of the same algorithm is 0.79–2.43x (1.23x on average) faster than theirs, and so we parallelize our own version as a baseline. We also tested the multicore implementation of the parallel OPTICS algorithm in [184] using all 48 cores on our machine. Their code exceeded our 3-hour time limit for our data sets with 10 million points. On a smaller data set of 1 million points (2D-SS-varden-1M), their code took 7988.52 seconds, whereas our fastest parallel implementations take only a few seconds. We also compared with the parallel HDBSCAN*code by Santos et al. [203], which mainly focuses on approximate HDBSCAN*in distributed memory. As reported in their paper, for the HT data set with minPts = 30, their code on 60 cores takes 42.54 and 31450.89 minutes to build the approximate and exact MST, respectively, and 124.82 minutes to build the dendrogram. In contrast, our fastest implementation using 48 cores builds the MST in under 3 seconds, and the dendrogram in under a second.

Overall, we found the fastest sequential methods for EMST and HDBSCAN*to be our EMST-MemoGFK and HDBSCAN*-MemoGFK methods running on 1 thread. Therefore, we also based our parallel implementations on these methods.

**Performance of Our Implementations.** Raw running times for our implementations are presented in Tables 5 and 6 in the Appendix. Table 7 shows the self-relative speedups and speedups over the fastest sequential time of our parallel implementations on 48 cores. Figures 22 and 23 show the parallel speedup as a function of thread count for our implementations of EMST and HDBSCAN*with minPts = 10,

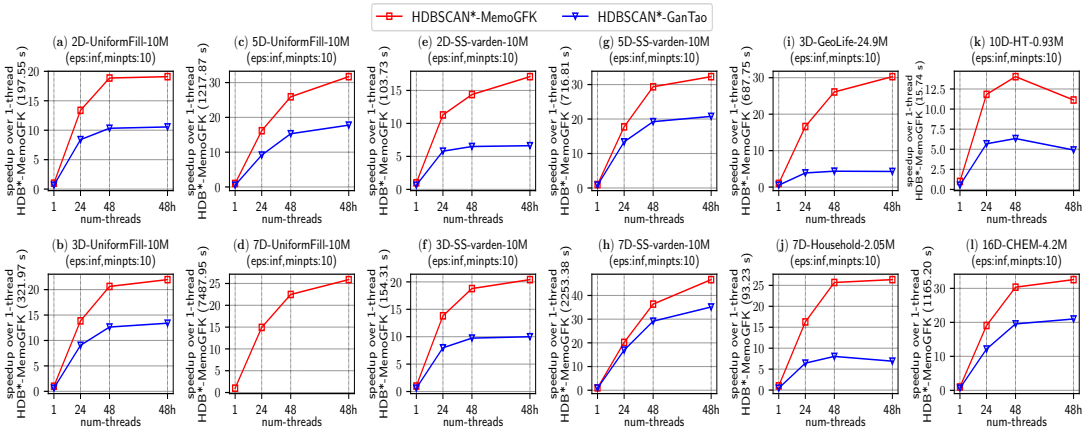



Figure 23: Speedup of HDBSCAN*implementations (minPts = 10) over the *best* serial baselines vs. thread count. The best serial baseline and its running time for each data set is shown on the $y$-axis label. "48h" on the $x$-axis refers to 48 cores with hyper-threading.

| | `mlpack` (1 thread) |
|---|---|
| 2D-UniformFill-10M | 90.09 |
| 3D-UniformFill-10M | 211.04 |
| 5D-UniformFill-10M | 964.13 |
| 7D-UniformFill-10M | 4777.29 |
| 2D-SS-varden-10M | 84.79 |
| 3D-SS-varden-10M | 139.18 |
| 5D-SS-varden-10M | 184.08 |
| 7D-SS-varden-10M | 233.28 |
| 3D-GeoLife-10M | 211.37 |
| 7D-Household-2.05M | 59.15 |
| 10D-HT-0.93M | 14.85 |
| 16D-CHEM-4.2M | 732.6 |

Table 4: Table of running times in seconds for the sequential EMST implementation from `mlpack`.

respectively, against the fastest sequential times. For most data sets, we see additional speedups from using hyper-threading compared to just using a single thread per core. A decomposition of parallel timings for our implementations on several data sets is presented in Figure 24. In addition, table 5 shows the running times of our

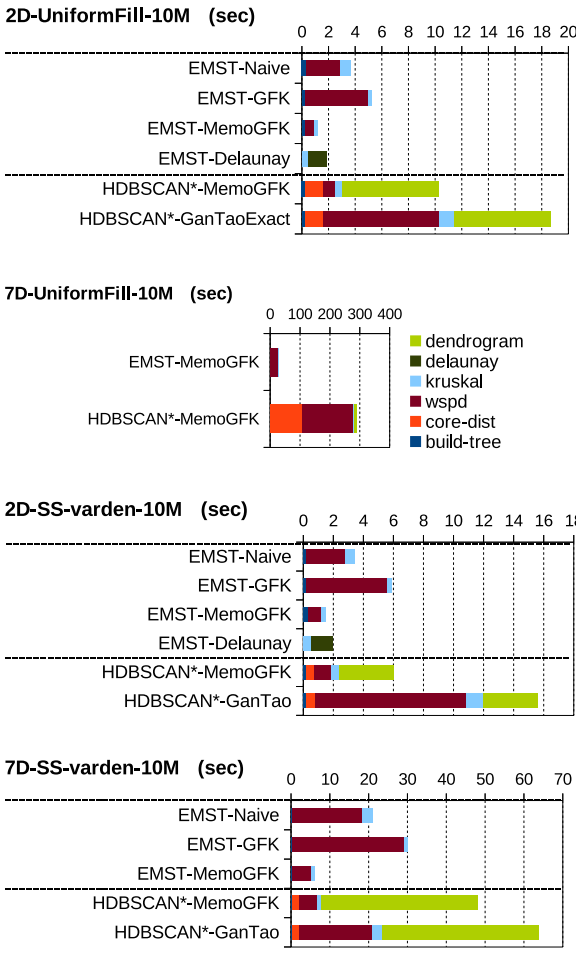



Figure 24: Decomposition of running times for constructing the EMST and HDBSCAN*MST on various data sets using all 48 cores with hyper-threading. minPts = 10 for HDBSCAN*. In the legend, "dendrogram" refers to computing the ordered dendrogram; "delaunay" refers to computing the Delaunay triangulation; "kruskal" refers to Kruskal's MST algorithm; "wspd" refers to computing the WSPD, or the sum of WSPD tree traversal times across rounds; "core-dist" refers to computing core distances of all points; and "build-tree" refers to building a $k$d-tree on all points.

| EMST | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | EMST-Naive | | EMST-GFK | | EMST-MemoGFK | | Delaunay | |
| | 1 thread | 48 cores | 1 thread | 48 cores | 1 thread | 48 cores | 1 thread | 48 cores |
| 2D-UniformFill-10M | 62.51 | 3.64 | 57.93 | 6.11 | 31.54 | **1.20** | 65.46 | 1.90 |
| 3D-UniformFill-10M | 400.57 | 19.30 | 218.02 | 26.07 | 67.80 | **2.24** | – | – |
| 5D-UniformFill-10M | – | – | – | – | 230.93 | **5.03** | – | – |
| 7D-UniformFill-10M | – | – | – | – | 1585.65 | **28.37** | – | – |
| 2D-SS-varden-10M | 57.84 | 3.45 | 60.64 | 6.90 | 27.48 | **1.60** | 64.42 | 1.95 |
| 3D-SS-varden-10M | 240.24 | 12.13 | 189.52 | 23.37 | 48.72 | **1.85** | – | – |
| 5D-SS-varden-10M | 478.40 | 19.41 | 278.10 | 31.19 | 96.19 | **3.04** | – | – |
| 7D-SS-varden-10M | 626.78 | 21.10 | 336.62 | 29.26 | 205.15 | **6.18** | – | – |
| 3D-GeoLife-10M | 271.95 | 10.97 | 328.76 | 36.31 | 117.31 | **4.77** | – | – |
| 7D-Household-2.05M | 280.28 | 8.37 | 214.08 | 24.77 | 37.60 | **1.40** | – | – |
| 10D-HT-0.93M | 19.28 | 0.64 | 12.36 | 1.40 | 5.17 | **0.35** | – | – |
| 16D-CHEM-4.2M | – | – | – | – | 821.81 | **19.11** | – | – |

Table 5: Table of running times in seconds for EMST. The fastest parallel time for each data set is in bold. The tests that do not complete within 3 hours or that run out of memory are shown as "–". The data sets with dimensionality greater than 2 are not applicable to Delaunay, and also shown as "–".

| HDBSCAN*(minPts = 10) | | | | |
|---|---|---|---|---|
| | HDBSCAN*-MemoGFK | | HDBSCAN*-GanTao | |
| | 1 thread | 48 cores | 1 thread | 48 cores |
| 2D-UniformFill-10M | 197.55 | **10.34** | 298.03 | 18.71 |
| 3D-UniformFill-10M | 321.97 | **14.66** | 517.71 | 24.04 |
| 5D-UniformFill-10M | 1217.87 | **38.41** | 2395.68 | 68.54 |
| 7D-UniformFill-10M | 7487.95 | **289.27** | – | – |
| 2D-SS-varden-10M | 103.73 | **6.07** | 163.37 | 15.66 |
| 3D-SS-varden-10M | 154.31 | **7.56** | 253.06 | 15.44 |
| 5D-SS-varden-10M | 716.81 | **22.20** | 885.92 | 34.51 |
| 7D-SS-varden-10M | 2253.38 | **48.26** | 2585.83 | 64.13 |
| 3D-GeoLife-10M | 687.75 | **22.70** | 1320.15 | 160.48 |
| 7D-Household-2.05M | 93.23 | **3.54** | 204.75 | 13.51 |
| 10D-HT-0.93M | 15.74 | **1.41** | 29.75 | 3.21 |
| 16D-CHEM-4.2M | 1165.20 | **35.77** | 1820.61 | 55.52 |

Table 6: Table of running times in seconds for HDBSCAN*with minPts = 10. The fastest parallel time for each data set is in bold. The tests that do not complete within 3 hours, or that run out of memory are shown as "–".

| | Speedup over Best Sequential | | Self-relative Speedup | |
|---|---|---|---|---|
| Method | Range | Average | Range | Average |
| EMST-Naive | 3.51-10.69x | 6.90x | 16.79-33.47x | 24.15x |
| EMST-GFK | 1.52-7.01x | 3.60x | 8.11-11.51x | 9.08x |
| EMST-MemoGFK | 14.61-55.89x | 31.31x | 14.61-55.89x | 31.31x |
| Delaunay | 14.12-16.64x | 15.38x | 33.11-34.54x | 33.82x |
| HDBSCAN*-MemoGFK | 11.13-46.69x | 26.29x | 11.13-46.69x | 26.29x |
| HDBSCAN*-GanTao | 4.29-35.14x | 13.76x | 8.23-40.32x | 20.97x |

Table 7: Speedup over the best sequential algorithm as well as the self-relative speedup on 48 cores.

implementations for EMST. table 6 shows the running times of our implementations for HDBSCAN*.

**EMST Results.** In Figure 22, we see that our fastest EMST implementations (EMST-MemoGFK) achieve good speedups over the best sequential times, ranging from 14.61–55.89x on 48 cores with hyper-threading. On the lower end, 10D-HT-0.93M has a speedup of 14.61x (Figure 22k), because for a small data set, the total work done is small and the parallelization overhead becomes prominent.

EMST-MemoGFK significantly outperforms EMST-GFK and EMST-Naive by up to 17.69x and 8.63x, respectively, due to its memory optimization, which reduces memory traffic. We note that EMST-GFK does not get good speedup, and is slower than EMST-Naive in all subplots of Figure 22. This is because the WSPD input to EMST-GFK ($S$ in Algorithm 6) needs to store references to the well-separated pair as well as the BCCP points and distances, whereas EMST-Naive only needs to store the BCCP points and distances. This leads to increased memory traffic for EMST-GFK for operations on $S$ and its subarrays, which outweighs its advantage of computing fewer BCCPs. This is evident from Figure 24, which shows that EMST-GFK spends more time in WSPD, but less time in Kruskal compared to EMST-Naive. EMST-MemoGFK spends the least amount of time in WSPD due to its pruning optimizations, while spending a similar amount of time in Kruskal as EMST-GFK. Finally, the EMST-Delaunay implementation performs reasonably well, being only slightly (1.22–1.57x) slower than EMST-MemoGFK; however, it is only applicable for 2D data sets.

**HDBSCAN*Results.** In Figure 23, we see that our HDBSCAN*-MemoGFK method achieves good speedups over the best sequential times, ranging from 11.13–46.69x on 48 cores. Similar to EMST, we observe a similar lower speedup for 10D-HT-0.93M due to its small size, and observe higher speedups for larger data sets. The dendrogram construction takes at least 50% of the total time for Figures 7a, b, and e–h,

and hence has a large impact on the overall scalability. We discuss the dendrogram scalability separately.

We find that HDBSCAN*-MemoGFK consistently outperforms HDBSCAN*-GanTao due to having a fewer number of well-separated pairs (2.5–10.29x fewer) using the new definition of well-separation. This is also evident in Figure 24, where we see that HDBSCAN*-MemoGFK spends much less time than HDBSCAN*-GanTao in WSPD computation.

We tried varying minPts over a range from 10 to 50 for our HDBSCAN*implementations and found just a moderate increase in the running time for increasing minPts.

**MemoGFK Memory Usage.** Overall, the MemoGFK method for both EMST and HDBSCAN*reduces memory usage by up to 10x compared to materializing all WSPD pairs.

**Dendrogram Results.** We separately report the performance of our parallel dendrogram algorithm in Figure 25, which shows the speedups and running times on all of our data sets. We see that the parallel speedup ranges from 5.69–49.74x (with an average of 17.93x) for the HDBSCAN*MST with minPts=10, and 5.35–52.58x (with an average 20.64x) for single-linkage clustering, which is solved by generating a dendrogram on the EMST. Dendrogram construction for single-linkage clustering shows higher scalability because the heavy edges are more uniformly distributed in space, which creates a larger number of light-edge subproblems and increases parallelism. In contrast, for HDBSCAN*, which has a higher value of minPts, the sparse regions in the space tend to have clusters of edges with large weights even if some of them have small Euclidean distances, since these edges have high mutual reachability distances. Therefore, these heavy edges are less likely to divide up the edges into a uniform distribution of subproblems in the space, leading to lower parallelism. On the other hand, we observe that across all data sets, the dendrogram for single-linkage clustering takes an average of 16.44 seconds, whereas the dendrogram for HDBSCAN*takes an average of 9.27 seconds. This is because the single-linkage clustering generates more light-edge subproblems and hence requires more work. While it is possible to tune the fraction of heavy edges for different values of minPts, we found that using $n/10$ heavy edges works reasonably well in all cases.

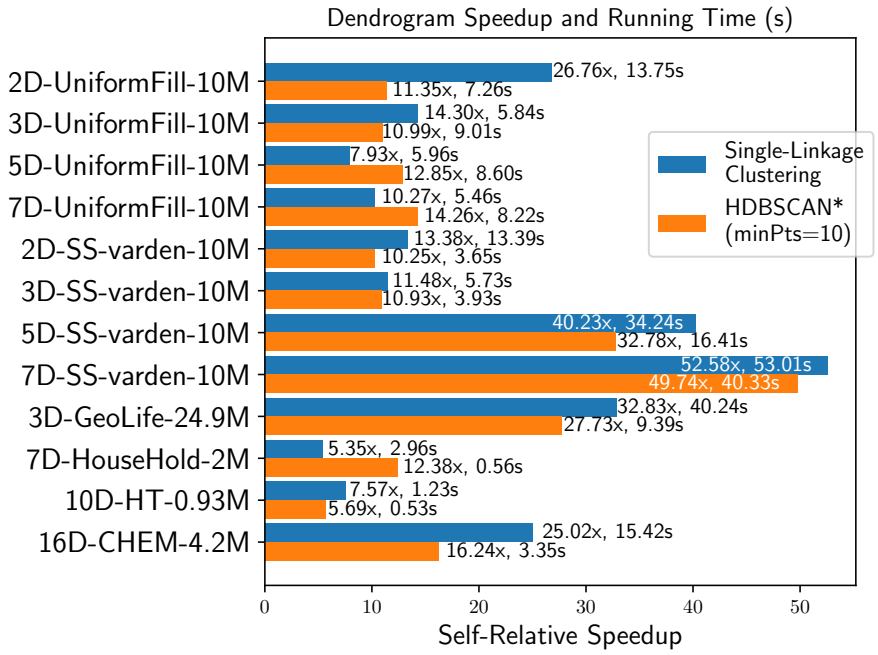



Figure 25: Self-relative speedups and times for ordered dendrogram computation for single-linkage clustering and HDBSCAN*(minPts = 10). The $x$-axis indicates the self-relative speedup on 48 cores with hyper-threading. The speedup and time is shown at the end of each bar.

# 6 A Framework for Parallel Grid-Based Clustering

## 6.1 Introduction

Grid-based clustering algorithms is a significant class of clustering algorithms, known for their efficiency in processing large multi-dimensional point data sets. These algorithms, such as GRIDCLUS [205], STING [238], CLIQUE [16], and WaveCluster [258], grid-based DBSCAN [117], approximate DPC [19], NSGC [248], GDILC [253], and ASGC [65], generally follow a similar approach by partitioning each dimension of the data space into intervals, forming a grid structure where data points belong to cells. Clusters are then formed directly from these cells. Despite the common grid structure used across these algorithms, there has been little effort in generalizing their implementations.

In this thesis, we aim to develop a framework for parallel grid-based clustering by creating high-level programming constructs that can be used to implement various parallel grid-based clustering algorithms efficiently. Our goal is to enable simple and fast parallel programming for spatial data sets. We identify several core constructs necessary for many of these algorithms, such as one that takes a cell and range as input and applies a user-defined function to all cells within the specified neighborhood, and another that traverses all cells in a user-defined order while applying a user-defined function. We implement both sequential and parallel versions of these constructs, and use them to implement several algorithms including density-based spatial clustering of applications with noise (DBSCAN) [95], the grid-based density-isoline clustering algorithm (GDILC) [253], the new shifting grid clustering algorithm (NSGC) [163], density peak clustering (DPC) [199, 19], grid clustering (GRIDCLUS) [205], and an axis-shifted grid-clustering algorithm (ASGC) [65]. We describe the grid data structure and its primitives in Section 6.2, the implementation of various grid clustering algorithms using the data structure in Section 6.3, and an experimental evaluation in Section 6.4.

## 6.2 Grid Data Structure

In this section, we describe the grid data structure implementation, and the definition and usage of the primitives that we propose. The grid data structure is a commonly used data structure in grid clustering algorithms. However, implementing and optimizing a grid data structure can be complex, time-consuming, and prone to

errors. To address this issue, we introduce a grid object that encapsulates the details and supports several parallel programming constructs. This approach simplifies programming for grid clustering algorithms by hiding implementation details.

At a high level, the grid data structure takes a multi-dimensional point data set as input and organizes it into an axis-aligned grid structure with a specified side length, denoted as $\boldsymbol{G}$. Each partition of the grid is a hypercube, referred to as a ***cell***, and each point within the cell is called an ***object***.

### Programming Constructs

We describe several fundamental parallel constructs that form the core of all grid-based clustering algorithms. These constructs serve as the primary interface through which programmers interact with the grid structure and are implemented as subroutines of the grid.

### BuildGrid

```
BuildGrid(objects P, object origin, float d)
```

The **BuildGrid** construct takes in a set of data points $P$, an separate object called the origin, whose coordinate represents that of the "lower left" of the grid data structure. The third parameter $d$ represents the length of each side of the cells in the grid.

### CellMap

```
CellMap(lambda (cell c, int id): { ... })
```

The **CellMap** construct iterates through all cells in $G$ sequentially and performs computations on each cell. It takes a lambda function as input; this function receives references to both a cell and its ID number. The computations are implemented within the lambda function. For example, **CellMap** can be used to count the total number of objects in $G$ using the following code:

```
// Example: accumulate total number of objects in numObj
CellMap(lambda (cell c, int id): { numObj += c.size })
```

We also implement a **ParallelCellMap** construct that iterates through cells in parallel. Consequently, the computation within the lambda function must be thread-safe. The example below demonstrates how to compute the total number of objects in parallel using atomic addition.

```
// Example: accumulate total number of objects in numObj in parallel
ParallelCellMap(lambda (cell c, int id): { AtomicAdd(numObj, c.size) })
```

Some of the algorithms require the cells to be processed in some order. We use a bucketing scheme, where the cells are divided into buckets. We call such a construct **ParallelOrderedCellMap**. In this construct, all element of a bucket will be processed in parallel, whereas different buckets are processed sequentially.

### NeighborCellMap

```
NeighborCellMap(cell c, lambda (cell ngh, int id): { ... })
```

The **NeighborCellMap** construct iterates through and performs computations on each cell within the neighborhood of a given cell $c$. It takes a cell whose neighborhood is to be traversed and a lambda function as input. The lambda function receives references to a neighboring cell and its ID number. Operations on the neighborhood are implemented within the lambda function. This primitive supports multiple ways of defining neighborhoods, such as spherical or rectangular shapes. The type and size of the neighborhood are given as optional parameters to the primitive. By default, the neighborhood is defined as the cells that are adjacent to $c$.

Additionally, each cell has a primitives for iterating through objects. The example below demonstrates computing the average density of cells in a neighborhood surrounding cell $c$.

```
// Example: average neighborhood density
numNgh = sum = 0
NeighborCellMap(c, lambda (cell ngh, int id): {
  numNgh ++
  sum += ngh.size
})
sum /= numNgh
```

We also propose a parallel variant of NeighborCellMap, called **NeighborObjectMap**. In this variant, instead of traversing the neighboring cells, objects inside the neighboring cells are traversed instead.

In addition, we also propose parallel variants of the mapping functions, **ParallelNeighborCellMap** and **ParallelObjectMap**, where the neighboring cells and objects are iterated in parallel.

## 6.3 Implementing Clustering Algorithms

### DBSCAN

We refer the readers to Section 4 for the definition of DBSCAN clustering. We describe the algorithm that computes the DBSCAN using our grid framework in Algorithm 8. From Line 3 to Line 6, we use a **ParallelCellMap** to mark points whose cell with higher than $minPts$ number of points as core. From Line 7 to Line 13, we call **NeighborObjMap** on every unmarked point in parallel to accumulate the number of nearby points, and determine whether they are core points. From Line 15 to 19, we again use **NeighborObjMap** on every unclustered point in parallel to check if they border any clusters.

**Algorithm 8** Implementing DBSCAN

1: **Input:** a data set $P$ of dimension dim, $\epsilon$, $minPts$
2: $G \leftarrow$ **BuildGrid**($P$, $origin$, $\epsilon/\sqrt{dim}$) ▷ $origin$: coordinate with lowest value in each dimension
3: $G$.**ParallelCellMap**(cell $c$: { ▷ Mark core points
4: **if** $c$.size $\geq minPts$ **then**
5: Mark all points in $c$ as core
6: })
7: **par-for** unmarked $p \in P$ **do** ▷ Mark remaining core points
8: $count \leftarrow 0$
9: $G$.**NeighborObjMap**($p$, obj $q$: {
10: **if** $p$.dist($q$) $\leq \epsilon$ **then**
11: $count \leftarrow count + 1$
12: })
13: Mark $p$ as core point if $count \geq minPts$
14: Compute connected components on the cells, propagate labels to core points, and form clustering $C$, where $C[p] = -1$ if $p \in P$ is non-core
15: **par-for** non-core $p \in P$ **do** ▷ Cluster border points
16: $G$.**NeighborObjMap**($p$, obj $q$: {
17: **if** $q$ is core and if $p$.dist($q$) $\leq \epsilon$ **then**
18: $C[p] \leftarrow C[q]$
19: })
20: return $C$

### Approximate DPC

The density-peak clustering (DPC) algorithm is proposed by Rodriguez and Laio [199]. The algorithm computes, for each point from the input, the local density, defined as the number of points whose distance to the point is less a user-defined threshold; and dependent distance, defined as the point's nearest neighbor with a higher local density. The algorithm then identifies cluster centers whose dependent distances are above a user-defined threshold.

The approximate DPC (Approx-DPC) algorithm is proposed by Amagata and Hara [19]. It is a grid-based clustering method that improves the efficiency of local density and dependent point computation in clustering tasks when compared with the exact algorithm. The algorithm employs a uniform grid, which consists of non-empty cells with each cell being a hypercube. Each cell in the grid maintains information about the points covered by the cell. The algorithm first computes the local density, the number of points within a radius of input parameter ***dcut***. Approx-DPC conducts an improved range search in each cell of the grid called a joint range search. It involves computing the center point of each cell, obtaining its range search result through a single query on a prebuilt spatial tree, and scanning this result to compute exact local densities for each point in the cell. This joint range search reduces unnecessary tree traversal and improves performance compared to existing methods. This process is repeated for every cell.

The algorithm then computes the dependent points, defined as the closest neighbor with a higher local density. Approx-DPC allows approximate computation based on information maintained in each cell while still ensuring clustering accuracy. For approximate dependent point computation, Approx-DPC follows specific rules depending on whether a point is equal to or different from the maximum local density point within its cell. If a dependent point cannot be determined through these rules, Approx-DPC computes it using the exact dependent point computation.

We describe the algorithm that computes the Approximate Density Peak Clustering using our grid framework in Algorithm 9. From Line 3 to Line 6, we use a **ParallelCellMap** to compute local density for each point in $P$ using range search within distance ($dMax+dCut$) from each cell. From Line 7 to Line 9, we use another **ParallelCellMap** to find $\rho_{\min}$ and $p^*$ for each cell. From Line 10 to Line 16, we again use **ParallelCellMap** to assign $p*$ as dependent points for points in each cell, and then check neighboring cells' $\rho$-min values to find additional dependent points. We then resolve remaining dependent points using exact computations. Finally, the dependency can be converted to clustering by forming connected components and discarding dependency edges with weights higher than a user-defined threshold.

**Algorithm 9** Implementing Approx-DPC

**Input**: a data set $P$ of dimension $dim$, $dCut$
$G \leftarrow$**BuildGrid**($P$, $origin$, $dCut/\sqrt{dim}$) ▷ $origin$: coordinate with lowest value in each dimension
$G$.**ParallelCellMap**(cell $c$: { ▷ Compute local density
  Perform range search on points within distance $(dMax + dCut)$ from $c$
  Compute local density for each point $p$ in $c$ using the range search result
})
$G$.**ParallelCellMap**(cell $c$: {
  Find minimum local density $\rho_{\min}$ and local max-density point $p^*$ for $c$
})
$G$.**ParallelCellMap**(cell $c$: { ▷ Compute approximate dependent points
  Assign $p^*$ as dependent points for points in $c$
  $G$.**NeighborCellMap**($p^*$, cell $ngh$: {
    **if** $ngh.\rho_{\min} \geq p^*$'s local density **then**:
      Assign $p^*$' of $ngh$ as $p^*$'s dependent point
  })
})
Resolve remaining dependent points using exact computations
Compute connected components $C$ based on the dependency while discarding dependency edges with weights higher than a user-defined threshold
Return $C$

### NSGC

The new shifting grid clustering algorithm (NSGC) [248] divides the data space into a grid with cells. The number of intervals, which determines the number of cells in the data space, is set to be $2m$, where $m$ is the number of iterations of the algorithm. To improve performance in cases where data points are distributed evenly throughout the data space, the shifting grid structure uses density profiles from nearest neighborhood cells rather than only considering the density of a single cell. This is achieved by shifting the grid by a half-cell size in each dimension, allowing for a more accurate assessment of local densities. Group assignments are made to non-empty cells based on density profiles, with neighbor cells being checked during assignment to ensure proper grouping. The order of group assignment is performed according to density profiles, starting with high-density cells and moving towards lower-density ones. The algorithm iterates until a stopping criteria is met. Specifically, if the difference between results from previous and current iterations is less than or equal to an acceptable error, then the current result is recognized as final.

We describe the new shifting grid clustering algorithm using our grid framework in Algorithm 10. The input is a data set $P$ of dimension $dim$. We first normalize the data set to values in [0, 1]. Then, we run a loop from $m = 0$ to $maxIter - 1$. In each iteration, we create a grid $G$ on $P$ with side length $gridSize = 1/m$. From Line 6 to Line 12, we use a **NeighborObjectMap** to compute the density of each cell in parallel. We order the cells by decreasing density and update their group assignment based on neighbor cluster IDs from Lines 13– 16. We repeat this process until the difference between clusterings is small enough and break out of the loop on Line 17. Finally, we return the clustering.

### GDILC

The grid-based density-isoline clustering (GDILC) algorithm [253] is a novel approach to clustering data samples based on their densities and distance thresholds. The first step in the algorithm is to calculate the density of each data sample based on the number of other samples within a certain distance threshold. This density value is then used to form a density-isoline figure, which is essentially a contour figure of density. Isolines are chosen based on a certain value, and regions circumscribed by those isolines are considered as clusters. Different isoline values can result in different clusters.

To determine which pairs of samples should be considered for clustering, a distance threshold $RT$ is derived from the distances between each pair of data samples.

**Algorithm 10** Implementing NSGC

```
1: Input: a data set P of dimension dim
2: Normalize the data set to values in [0, 1]
3: for m ← 0 to maxIter − 1 do
4:   gridSize ← 1/m
5:   G ←BuildGrid(P, origin, gridSize)  ▷ origin: coordinate with lowest value in each dimension
6:   par-for cell c ∈ G do                                  ▷ Compute cell densities
7:     c.density ← 0
8:     G.NeighborObjectMap(c, obj ngh: {
9:       for d ← 0 to dim do
10:        if abs(ngh[d] - c.center[d]) ≤ gridSize/2 then
11:          c.density ← c.density + 1
12:    })
13:  Ordering ← order cells by decreasing density
14:  G.CellMapOrdered(Ordering, cell c: {
15:    Check neighbor cluster IDs and generate a cluster ID to form clustering C
16:  })
17:  Stop if the clustering difference is small enough
18: Return C
```

Similarly, a density threshold $DT$ is obtained from the densities of samples to determine which pairs of samples should be combined into clusters. To reduce time and space requirements, a grid-based approach is employed to check only those pairs of samples within each cell in a grid whose centers are closer than $RT$. For each pair of data samples whose densities exceed $DT$ and whose distances are less than $RT$, the two clusters to which they belong are combined into one cluster. This process continues until all pairs of samples have been checked. We describe the GDILC algorithm using our grid framework in Algorithm 11. We first normalize the input data set to values in $[0, 1]$, and then build a grid on it with a user-defined side length. Next, we estimate the radius threshold ($RT$) and density threshold ($DT$) for clustering. From Line 4 to Line 10, we use **NeighborObjectMap** to calculate the average distance of each point to its neighbors, and then compute $RT$ as the average of these distances over all points. From Line 12 to Line 17, we use **NeighborObjectMap** again to count the number of neighbors within $RT$ for each point, and then compute $DT$ based on their mean density. Finally, from Line 20 to Line 25, we use **NeighborObjectMap** one more time to link points whose density is higher than DT within RT using a union-find data structure, and generate cluster labels accordingly.

### ASGC

The Axis-Shifted Grid-Clustering (ASGC) algorithm [65] is a grid-based clustering method that aims to reduce the influence of the border of predefined grids and increase the selection of size and density threshold of significant cells. The algorithm works by generating two grid structures, identifying significant cells, grouping nearby significant cells into clusters, transforming the grid structure, generating new clusters using the transformed grid structure, revising original clusters using a cluster modified function, and generating the final clustering result. In the first step, a grid structure is generated by dividing the n-dimensional data space into a predefined number of non-overlapping cells. The density of each cell is then calculated to identify significant cells whose densities exceed a predefined threshold. Nearby significant cells are grouped into clusters to generate a set of clusters denoted as $S_1$. The coordinate origin is then shifted by half the grid cell size in each dimension to generate a new grid structure used to generate another set of clusters denoted as $S_2$. The originally obtained clusters are revised using the newly obtained ones using a merging algorithm and form a final set of clusters. We describe the merging procedure in Algorithm 12 and Algorithm 13. The ASGC cluster merging algorithm involves revising the original clusters obtained from two different grid structures. The first step is to find each overlapped cluster in the second grid structure for each cluster in

**Algorithm 11** Implementing GDILC

```
1: Input: a data set P of dimension dim
2: Normalize the data set to values in [0, 1]
3: G ←BuildGrid(P, origin, gridSize) ▷ origin: coordinate with lowest value in
   each dimension; gridSize: user-defined
4: par-for point p ∈ P do                                          ▷ Estimate RT
5:   p.total ← 0
6:   p.count ← 0
7:   G.NeighborObjectMap(p, obj ngh:
8:     p.total ← p.total + ngh.dist(p)
9:     p.count ← p.count + 1
     )
10:  p.distance ← p.total/p.count
11: RT ← sum(p.total/p.count for p ∈ P)/P.size
12: par-for point p ∈ P do                                          ▷ Estimate DT
13:  p.density ← 0
14:  G.NeighborObjectMap(p, obj ngh: {
15:    if dist(p, ngh) ≤ RT then
16:      p.density ← p.density + 1
17:  })
18: meanDensity ← sum(p.density for p ∈ P)/P.size
19: DT ← based on mean density
20: par-for point p ∈ P do                                          ▷ Generate clusters
21:  Initialize a union-find data structure uf for p if its density is greater than DT
22:  G.NeighborObjectMap(p, obj ngh: {
23:    if p.density > DT and dist(p, ngh) ≤ RT then
24:      Link p and ngh in uf
25:  })
26: Generate cluster labels C based on uf for each p with p.density > DT
27: Return C
```

the first grid structure and vice versa. This generates a set of rules denoted as $R_0$, which indicate overlapping clusters between the two structures. Then, each cluster in the first grid structure is revised using a cluster revised function $CM()$ described in Algorithm 13. Finally, the revised clusters from both structures are merged to generate the final clustering result.

**Algorithm 12** ASGC Rule Generation

```
1: procedure RuleGeneration(S_1, S_2)
2:   for C_1i ∈ S_1 do
3:     for C_2j ∈ S_2 such that C_1i ∩ C_2j ≠ ∅ do
4:       Generate the rule C_1i → C_2j
5:   for C_2j ∈ S_2 do
6:     for C_1i ∈ S_1 such that C_2j ∩ C_1i ≠ ∅ do
7:       Generate the rule C_2j → C_1i
8:   R_0 = {C_1i → C_2j | C_1i ∩ C_2j ≠ ∅} ∪ {C_2j → C_1i | C_2j ∩ C_1i ≠ ∅}
9:   Revise clusters in S_1 using the cluster revised function CM()
```

**Algorithm 13** ASGC Cluster Merging

```
1: procedure CM(S_1, S_2, R_0)
2:   for C_1i ∈ S_1 do
3:     Let X' ← X
4:     repeat
5:       oldX' ← X'
6:       for each Y → Z in R_0 do
7:         if Y ⊆ X' then
8:           X' ← X' ∪ Z
9:           if Z ∈ S_1 then
10:            S_1 ← S_1 − Z
11:          else
12:            S_2 ← S_2 − Z
13:    until oldX' = X'
14:    C_1i ← X'
15:  S_1 ← S_1 ∪ S_2
```

We propose a substantially simpler implementation of the algorithm that returns the same clustering result, by making use of a parallel union-find data structure,

as shown in Algorithm 14. The input is a data set $P$ of dimension $dim$. We first normalize the data set to values in [0, 1]. We then construct two grids $S_1$ and $S_2$ on $P$ with side length $d$ and shifted by $d/2$ relative to $S_1$, respectively. We initialize a union-find structure *uf*. From Line 6 to Line 12, we use a **ParallelCellMap** on cells of grid $S_1$ to link neighboring cells whose size is greater than the threshold. We repeat this process for cells in grid $S_2$. From Line 14 to Line 18, we again use **ParallelCellMap** on cells of grid $S_1$ to link neighboring cells whose overlapping area contains shared data points.

Our new implementation obtains the same result as the original algorithm. As shown in the original rule generation (Algorithm 12), a rule is generated if two cells from $S_1$ and $S_2$ share overlapping points. In Algorithm 13, these two cells are merged based on the set of rules in an iterative manner. In our implementation (Algorithm 14), from Line 14 to Line 18, in parallel for each cell in $S_1$, we check the overlapping cells in $S_2$, and connect them into the same cluster in the union-find. We show that any two cells that are placed in the same cluster in the original algorithms are also placed in the same cluster in our implementation. Let $C_{1i}$ be a cluster in $S_1$ and $C_{2j}$ be a cluster in $S_2$ such that $C_{1i} \cap C_{2j} \neq \emptyset$. According to Algorithm 12, a rule is generated $C_{1i} \rightarrow C_{2j}$. By the merging process in Algorithm 13, if two cells share overlapping points, they are placed in the same cluster. Therefore, any cell in $C_{1i}$ that overlaps with a cell in $C_{2j}$ must be placed in the same cluster as that cell. In our implementation (Algorithm 14), we connect each cell in $C_{1i}$ to its overlapping cells in $S_2$ using a parallel union-find data structure (Line 14-18). This ensures that any cell in $C_{1i}$ that overlaps with a cell in $C_{2j}$ is connected to it and placed in the same cluster. Therefore, all cells that are placed in the same cluster as $C_{1i}$ according to Algorithm 13 are also placed in the same cluster as it according to our implementation. By repeating this process for all cells, we can conclude that our new implementation obtains the same result as the original algorithm.

### GRIDCLUS

The GRIDCLUS algorithm [205] is a grid-based clustering algorithm that generates a hierarchy of clusters. The algorithm works by first constructing an axis-aligned grid structure on the data set, then iteratively processing all cells, and then using a recursive procedure to assign the cells of existing clusters.

The algorithm first creates the grid structure, and calculates the cell density based on the number of points in each cell. The cells are then sorted to generate a sequence in the order of highest to lowest density. Then the algorithm enters an iterative process. In each iteration, the cells with a next-higher density are treated as

**Algorithm 14** Implementing ASGC

```
1: Input: a data set P of dimension dim
2: Normalize the data set to values in [0, 1]
3: S_1 ←BuildGrid(P, origin1, d)
4: S_2 ←BuildGrid(P, origin2, d)   ▷ origin2 is shifted by d/2 relative to origin1
5: Initialize union-find uf
6: S_1.ParallelCellMap(cell c: {
7:   if c.size > threshold then
8:     G1.NeighborCellMap(c, cell ngh: {
9:       if ngh.size > threshold then
10:        uf.link(c, ngh)
11:    })
12: })
13: Repeat the same process for cells in S_2
14: S_1.ParallelCellMap(cell c: {
15:   for each S_2 cell ngh that overlaps with c do
16:     if c and ngh have shared data points then
17:       uf.link(c, ngh)
18: })
```

being ***active*** and ready to be clustered. Then connected components are recomputed on all the cells that are active, forming the clustering of that iteration. The algorithm proceeds until all the cells are active.

We describe the GRIDCLUS algorithm in Algorithm 15. Given a data set $P$ of dimension $dim$ and grid size $\epsilon$, we construct a grid $G$ on $P$ with side length $\epsilon$. We then sort the cells in $G$ based on cell density. From Line 5 to Line 10, we use a **ParallelCellMapOrdered** to compute connected components of all active cells, where active cells are those from index $s$ to index $e$. We record the connected components as one level of cluster. We output all the levels as the final clustering result.

**Algorithm 15** Implementing GRIDCLUS

1: **Input:** a data set $P$ of dimension $dim$, grid size $\epsilon$
2: $G \leftarrow$**BuildGrid**($P$, $origin$, $\epsilon$) ▷ $origin$: coordinate with lowest value in each dimension
3: Sort cells in $G$ based on cell density
4: $l \leftarrow 0$
5: $G$.**ParallelCellMapOrdered**(cell $c$, int $s$, int $e$: { ▷ Compute connected components of active cells
6: Mark cells from $s$ to $e$ as active
7: Compute connected components of all the active cells
8: Record the connected components as one level of cluster $C_l$
9: $l \leftarrow l + 1$
10: })
11: Return $C_i$ for $i \in [0, l)$

## 6.4 Experimental Evaluation

We implemented the framework using C++ and used the ParlayLib [41] for parallel primitives and a work-stealing scheduler. We compile the code with the g++ compiler (version 11.3) with the −O3 flag. We benchmark all the grid clustering algorithms on an Amazon EC2 machine. We use a c5.18xlarge instance with 2 × Intel Xeon Platinum 8124M (3.00GHz) CPUs for a total for a total of 36 two-way hyper-threaded cores, and 144 GB of RAM. We test the parallel performance of all the implementations on two data sets each with 1 million data points. We test both the sequential and parallel performance and reporting the self-relative parallel speedup. The data set *2d-uniform-1m* is a uniformly distributed data set in 2

| | *2d-uniform-1m* | | | *2d-varden-1m* | | |
|---|---|---|---|---|---|---|
| | 1t time | 36h time | speedup | 1t time | 36h time | speedup |
| DBSCAN | 0.822 | 0.0505 | 15.2x | 0.413 | 0.0358 | 11.5 |
| Approx-DPC | 261 | 5.74 | 45.5x | 53.6 | 2.75 | 19.5 |
| NSGC | 8.83 | 0.602 | 14.6x | 3.90 | 0.584 | 6.68 |
| GDILC | 11.3 | 1.21 | 9.34x | 116 | 15.7 | 7.36 |
| ASGC | 1.30 | 0.117 | 11.1x | 0.672 | 0.179 | 3.74 |
| GRIDCLUS | 0.773 | 0.0505 | 15.3x | 0.420 | 0.0519 | 8.11 |

Table 8: Sequential running time (*1t*, seconds), parallel running time on 36 cores with hyper-threading (*36h*, seconds), and parallel self-relative parallel speedups of our implementations of the grid clustering algorithms using the framework.

dimensions, and *2d-varden-1m* is a clustered data set with varying density in 2 dimensions. We benchmark all of the implementations and present their self-relative parallel speedups using all the cores of the machine in Table 8.

We observe that the grid clustering implementations achieve better running time when running on *2d-uniform-1m* compared with running on *2d-varden-1m*. This is because when a data set is uniformly distributed, the data points fall more evenly into the grid cells created, and it creates a higher number of grid cells, which can be parallelized more easily. In addition, the more even distribution of points also results in better load balancing. Among the methods, GDILC has relatively lower parallel speedups on both data sets, because in earlier iterations of the algorithm, the number of grid cells is small, and the parallelism only comes from processing different grid cells in parallel.

# Part III

# Algorithms and Libraries for Parallel Computational Geometry

# 7 Introduction

Computational geometry has a wide range of applications across various domains, including computer graphics, robotics, simulation, optimization, and games. This thesis presents three new parallel algorithms that we developed for convex hull, smallest enclosing ball, and parallel batch-dynamic closest pair. We also describe the ParGeo library and the GeoGraph library for parallel computational geometry and geometric graph generation. For convex hull in both $\mathbb{R}^2$ and $\mathbb{R}^3$, we introduce a reservation technique to enable parallel modifications to the hull. For smallest enclosing ball, we propose a new sampling-based algorithm based on Larsson et al.'s [154] approach to quickly reduce the size of the data set. For closest pair, we develop a new parallel algorithm based on Golin's dynamic data structure [111]. We consolidate the implementation of these algorithms and others into unified libraries to make them more user-friendly. ParGeo is a library of parallel computational geometry algorithms that offers efficient and user-friendly solutions for geometric problems and data structures. Unlike existing libraries, ParGeo includes a wide range of problems specifically designed for parallel processing. Additionally, it includes a module that generates geometric graphs from spatial data sets to enable users to take advantage of graph algorithms to gain insights into their data. We summarize our contributions below.

- Optimized parallel randomized incremental and quickhull algorithms using the reservation technique, as well as a divide-and-conquer algorithm, for convex hull in $\mathbb{R}^2$ and $\mathbb{R}^3$. (Section 8.2)
- A parallel sampling-based algorithm for the smallest enclosing ball problem, and the first parallel implementation of the classic randomized incremental algorithm. (Section 8.3)
- A new parallel batch dynamic data structure for the closest pair problem, that is both theoretically-efficient and practical. (Section 8.4)
- The ParGeo library for parallel computing in computational geometry and graph generation. (Section 9)

# 8 New Parallel Algorithms

## 8.1 Introduction

We present three new parallel algorithms that we developed for convex hull, smallest enclosing ball, and parallel batch-dynamic closest pair. We also describe the ParGeo library and the GeoGraph library for parallel computational geometry and geometric graph generation. For convex hull in both $\mathbb{R}^2$ and $\mathbb{R}^3$, we introduce a reservation technique to enable parallel modifications to the hull. For smallest enclosing ball, we propose a new sampling-based algorithm based on Larsson et al.'s [154] approach to quickly reduce the size of the data set. We also provide the first parallel implementation of the classic randomized incremental algorithm. For closest pair, we develop a new parallel algorithm based on Golin's dynamic data structure [111] called the sparse partition to support batch-dynamic updates. We describe these algorithms in Sections 8.2, 8.3, and 8.4, respectively.

## 8.2 Convex Hull

The convex hull of a set of points $P$ in $\mathbb{R}^d$ is the smallest convex polyhedron containing $P$ (in this thesis, we assume the dimensionality $d$ is a constant). It is common to represent the convex hull using a set of ***facets***. The boundary of two facets is a ***ridge***. For example, in $\mathbb{R}^3$, assuming the points are in general position (no four points are on the same plane), each facet is a triangle, and each ridge is a line that borders two facets (see Figure 41(a)).

The randomized incremental algorithm and the quickhull algorithm are the most widely used algorithms for solving convex hull in practice. The randomized incremental algorithm for $\mathbb{R}^d$ was proposed by Clarkson and Shor [74]. Given a point data set $P$ in $\mathbb{R}^d$, the randomized incremental algorithm first constructs a $d$-simplex, a generalization of a tetrahedron in $d$-dimensions as the initial hull. Then, the algorithm adds the points to the polyhedron in a random order, updating the hull if necessary. In practice, the quickhull algorithm [114, 31], another incremental algorithm, is often used. Unlike the randomized algorithm, the quickhull algorithm processes a point that is furthest from a facet, which enables the hull to be expanded more quickly. The quickhull algorithm is by far one of the most common implementations for convex hull due to its simplicity and efficiency [5, 8, 9, 1, 4, 96]. There have also been works that study parallel implementations of quickhull, but they are either

limited to $\mathbb{R}^2$ [177, 219], or do not return the exact convex hull for $\mathbb{R}^3$ [220, 229]. Recently, Blelloch et al. [46] proposed a new randomized incremental algorithm that is highly parallel in theory. However, the algorithm does not seem to be practical due to numerous data structures required for bookkeeping.

In this section, we describe our new parallel reservation-based algorithm. Our algorithm is able to express both the randomized incremental convex hull algorithm and the quickhull algorithm. Specifically, unlike a sequential incremental algorithm that adds one point per round, we add multiple points in parallel per round. We resolve conflicts caused by the parallel insertion using a reservation technique. We also apply a general parallelization technique based on divide-and-conquer, which in combination with our parallel incremental algorithm, leads to faster implementations in practice.

### Parallel Reservation-Based Algorithm

Our parallel reservation-based algorithm can be implemented as either a randomized incremental algorithm or a quickhull algorithm. We will first introduce the overall structure of the algorithm. Then, we will describe the details with respect to the implementations, and compare with existing approaches. We will base our description in the context of $\mathbb{R}^3$ for the sake of clarity, but the algorithm can be extended to $\mathbb{R}^d$ for any constant integer $d \geq 2$.

We first give a high-level overview of the algorithm. Given an ordered set of points $P = \{p_1, p_2, \ldots, p_n\}$, we let $P_r = \{p_1, p_2, \ldots, p_r\}$ be the prefix of $P$ of size $r$, and $CH(P_r)$ be the convex hull on $P_r$. We start the construction by first arbitrarily selecting four points from $P$ that do not lie on the same plane, forming a tetrahedron. We then make these four points the first four in $P$, and denote the tetrahedron as $CH(P_4)$. Then, the algorithm proceeds iteratively, but on each round, rather than inserting just $p_r$ to form $CH(P_r)$, we process a batch of points in parallel. On each round, let each point outside of $CH(P_{r-1})$ be called a ***visible point***. We first select a batch of visible points, and try to add them to $CH(P_{r-1})$ in parallel in the same round.

The key challenge of this approach is that some of these points cannot be processed in parallel due to concurrent modifications on the shared structures of the convex polyhedron. We use a reservation algorithm to resolve these conflicts, such that we only process the points that modify disjoint facets of the polyhedron. Specifically, each point will perform a priority concurrent write with its ID to reserve all of its visible facets. Points that have its ID written to all of its visible facets are successful. We then process the successful points in parallel by enabling them to

make concurrent modifications to $CH(P_{r-1})$. At the end of the round, in parallel, we to filter out points that are no longer visible. The algorithm will terminate when there are no more visible points.

Our reservation-based algorithm can be used to implement the parallel randomized incremental algorithm or the quickhull algorithm for convex hull. For the randomized incremental algorithm, we randomly permute the input points at the beginning, and on each round attempt to add a prefix of the permuted points to the convex hull. For the quickhull algorithm, on each round, we instead select a set of points furthest from a subset of facets.

### Detailed Algorithm for the Convex Hull

**Overview** We first give a high-level overview of the algorithm, whose pseudocode is shown in Figure 17. Given an ordered set of points $P = \{p_1, p_2, \ldots, p_n\}$, we let $P_r = \{p_1, p_2, \ldots, p_r\}$ be the prefix of $P$ of size $r$, and $CH(P_r)$ be the convex hull on $P_r$. We start the construction by first selecting four points from $P$ that do not lie on the same plane, forming a tetrahedron (Line 1). We then make these four points the first four in $P$, and denote the tetrahedron as $CH(P_4)$. Then, the algorithm proceeds iteratively, but on each round, rather than inserting just $p_r$ to form $CH(P_r)$, we process a batch of points in parallel. On each round, let each point outside of $CH(P_{r-1})$ be called a ***visible point***. We first select a batch of visible points (Line 3), and try to add them to $CH(P_{r-1})$ in parallel in the same round.

The key challenge of this approach is that some of these points cannot be processed in parallel due to concurrent modifications on the shared structures of the convex polyhedron. We use a reservation algorithm to resolve these conflicts, such that we only process the points that modify disjoint facets of the polyhedron (Lines 4–9). Specifically, each point will perform a **WriteMin** with its ID to reserve all of its visible facets (Lines 4–6). Points that have its id written to all of its visible facets are successful (Lines 7–9). We then process the successful points in parallel by enabling them to make concurrent modifications to $CH(P_{r-1})$ (Line 10–14). At the end of the round, we run a **ParallelPack** to filter out points that are no longer visible (Line 15). The algorithm will terminate when there are no more visible points.

**Detailed Algorithm** Next, we describe the algorithm in greater detail. Figure 26 illustrates the processing of a single visible point $p_r$. We denote a facet as a ***visible facet*** of $p_r$ if point $p_r$ is in the half space away from the center of the convex hull. We first retrieve the set of visible facets of $p_r$ via facets stored in the visible point. The visible facets of $p_r$ form a closed region, whose boundary is a set of ridges known

**Algorithm 17** Pseudocode for the parallel reservation-based convex hull algorithm (which includes the randomized incremental algorithm and the quickhull algorithm).

**Require:** $P$: 3-dimensional points, $r$: batch size
**Ensure:** 3-dimensional convex hull

```
1:  CH ← initialize with 4 points
2:  while P is not empty do
3:    Q ← a batch of size r of visible points in P
4:    for all q ∈ Q do                                  ▷ Reservation (parallel loop)
5:      for all f ∈ q.visibleFacets do
6:        WriteMin(&f.reservation, q.id)
7:    for all q ∈ Q do                                  ▷ Check Reservation (parallel loop)
8:      for all f ∈ q.visibleFacets do
9:        q.success ← q.success ∧ (f.reservation = q.id)
10:   for all q ∈ Q do                                  ▷ Process Successful Points (parallel loop)
11:     if q.success then
12:       delete q's visible facets
13:       create new facets of q
14:       update CH
15:   P ← Pack(P, visible)
```

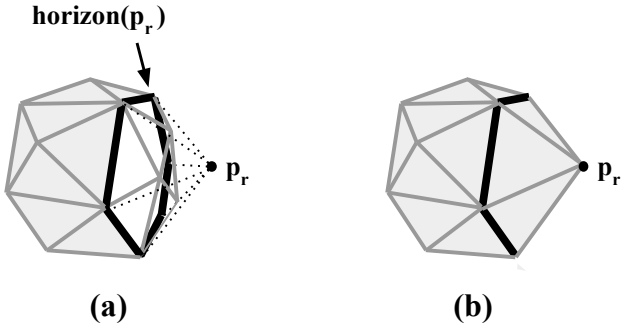



Figure 26: Illustration of adding a visible point $p_r$ to the convex hull. (a) shows the convex hull prior to the addition of $p_r$. The visible facets are in white, while the non-visible facets are in gray. The thicker line segments correspond to the horizon. (b) shows the convex hull after adding $p_r$ with newly created facets.

as the ***horizon***. We delete the visible facets from $CH(P_{r-1})$, and add new facets, where each new facet is formed by adding two ridges from a horizon ridge to $p_r$.

Because of the structural changes to the convex hull that occur when adding a visible point, concurrent structural changes can cause data races, which need to be avoided. We show an example of the conflict in Figure 27, where we are attempting to add two visible points $p_r$ and $p_{r+1}$ in parallel. As shown in the figure, the closed region formed by the visible facets of each visible point overlap with each other in three facets, which are highlighted in yellow. Should the two visible points be processed in parallel, the resulting polyhedron may not be well-defined due to data races. When processed sequentially, $p_{r+1}$'s visible facets would have been different, involving newly created facets by $p_r$.

Our reservation algorithm only allows a subset of the visible points that update disjoint facets of the convex hull to be processed in parallel in each round. At a high level, we use the lexicographical order of the visible points to determine the priority in processing a facet (a smaller ID has higher priority). In the example shown in Figure 27, since $p_r$ has a smaller ID than $p_{r+1}$, the three conflicting facets can only be processed by $p_r$ in that round. In the pseudocode in Figure 17, we allocate an extra data field in each facet for performing reservations (Lines 4–6). For each visible point in parallel, we iterate through its visible facets and use a parallel **WriteMin** to write its ID to the facets' "reservation" fields. Then on Lines 7–9, we determine which visible points successfully reserved all of its facets. Again, in parallel for each visible point, we check each of its visible facets for a successful reservation by comparing the value of the reservation field with its token. The reservation of a visible point is

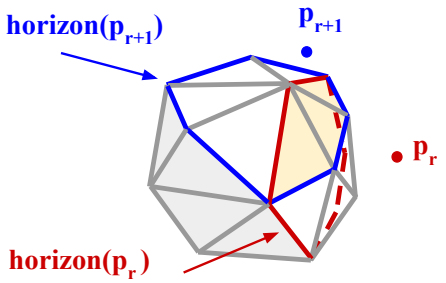



Figure 27: This figure illustrates the attempt to add $p_r$ and $p_{r+1}$ in parallel. The visible points and horizons of $p_r$ and $p_{r+1}$ are in red and blue, respectively. The visible facets to either visible points are in white/yellow, while the other facets are in gray. The overlap of the three visible facets between the $p_r$ and $p_{r+1}$ is in yellow.

only successful if its ID is stored in all of its visible facets. Then, on Lines 10–14, we process the visible points whose reservations are successful, adding them to the hull and updating the appropriate data structures. Finally, on Line 15, we process the points in $P$ such that those remaining as visible points are packed to replace the original $P$, and the non-visible points are discarded. Note that the visible points that succeeded in the reservation are no longer visible points because they are now part of the convex hull. Some of the remaining points will also no longer be visible points due to the growth of the convex hull.

We use a simple and fast data structure to keep track of the visibility relationship between the visible points and the facets. At each step of the algorithm, when a visible point is processed, it needs to identify the set of visible facets. On the other hand, for the facets undergoing structural changes, they need to identify and redistribute their visible points to new facets. To find the set of visible facets of $p_r$, it is inefficient to iterate through all the facets of $CH(P_{r-1})$. While existing approaches [86] keep track of the visibility between visible points and *all* of their visible facets, we found such an approach to be slow. The reason is that each vertex is associated with multiple facets, making the cost of storing and updating the data structure high. We only store the reference of an arbitrary visible facet to each visible point, from which we use a local breadth-first search to retrieve all the visible facets only when needed. For storing the visible points in the facets, we simply assign each point to one of its visible facets. During point redistribution, we gather the points stored in each visible facet into an array, and in parallel we distribute each point to a new visible facet replacing the original visible facet. Each such point also stores a reference to this visible facet.

**Randomized Incremental Algorithm** Our reservation-based algorithm can be used to implement the parallel randomized incremental algorithm. At the start of the algorithm, we randomly permute $P$. Line 3 will then take a prefix of the remaining points in $P$ to process. On each round, we choose a prefix of $c \cdot numProc$ visible points to perform the reservation, where $c$ is a small constant and $numProc$ is the number of processors. Compared with the existing parallelization approach by Blelloch et al. [46], our approach is much simple because we avoid the use of complicated data structures.

**Quickhull Algorithm** Our reservation-based algorithm also applies to the quickhull algorithm. Specifically, on Line 3 of the algorithm, we select a set of visible points that are furthest from their respective visible facets. The number of visible points that we choose to process on each round is again $c \cdot numProc$.

The 3-dimensional quickhull algorithm is one of the most widely used convex hull algorithms in practice, and so we compare with some of the existing approaches based on quickhull. Since concurrent insertions of visible points creates data races, existing 3 dimensional implementations compromise either correctness or scalability. For Stein et al.'s CudaHull algorithm [220], on each round, the algorithm chooses the furthest point for each facet, and replaces each facet with three new facets, which is done in parallel on the GPU. However, such an approach does not produce a convex polyhedron. Therefore, the implementation uses the CPU to fix the concave artifacts produced via ridge rotation at the end of each round. However, Gao et al. [106] pointed out that certain artifacts cannot be fixed by Stein et al.'s algorithm, leaving concavities in the final polyhedron. Tang et al.'s GPU-based heuristic for convex hull [224]. It first generates a "pseudohull" polyhedron using a quickhull-like algorithm, in which the points are removed. The convex hull is then computed on the remaining points sequentially on the CPU. A clear drawback is that the last phase of the algorithm is not parallel, causing a scalability bottleneck for certain data sets. In comparison, our approach computes a correct convex hull while also achieving high parallel scalability. The correctness is because for each visible point, our algorithm considers all the visible facets and replaces them with new facets. Meanwhile, Stein et al.'s algorithm only considers one visible facet for each visible point, and replaces it directly with three facets, giving rise to concavities during the process.

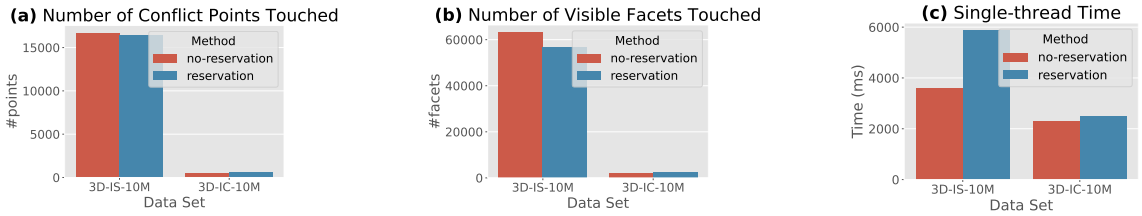



Figure 28: The plots show the overhead of reservation compared with without reservation. (a) and (b) show the number of visible points and facets touched by the algorithms, respectively. (c) shows the single-thread running time of the algorithms.

### Overhead of Reservation

Our parallel algorithms are work-efficient since it does the same amount of asymptotic work as the sequential counterparts for both the randomized incremental and quickhull algorithms. In addition, given that the expected number of visible facets associated with each visible point is constant [86], the amount of extra work done to perform the reservations on Lines 3–9 is a constant multiplicative factor in each round. When the number of facets is low, instead of using reservations, we process only a single point per round, and we choose the point from the facet that is visible to the most visible points, which maximizes the volume increase of the convex hull.

Figure 28 shows an empirical comparison on the amount of overhead incurred by the reservation algorithm. Specifically, the comparison is between the reservation-based quickhull algorithm and our optimized sequential quickhull algorithm, both running on one thread, for data sets containing 10 million points in 3 dimensions (described in Section 8.2). The purpose of running on one thread is to measure the amount of work without parallelism. The comparison is based on the number of visible points and facets touched during the algorithm as well as the running time. As we can see from Figures 28(a) and (b), the reservation-based algorithm does not necessarily cause more points or facets to be touched during the algorithm as a majority of the reservations succeed. For example, for the *3D-IS-10M* data set, the number of visible points and facets touched is similar to that of the non-reservation algorithm. For the *3D-IC-10M* data set, the reservation-based approach actually touches fewer visible points and facets, due to the different order in which the visible points are selected between the two algorithms. For both data sets, the reservation algorithm incurs some overhead in doing the work of reservations, as shown by the single-threaded running times in Figure 28(c); however the increase in running time is modest. This overhead is reasonable since it enables parallelism, as we will show in our experiments.

### Point Culling via Pseudohull Computation

We also implement a multicore variant of Tang et al.'s pseudohull heuristic [224], originally proposed for the GPU. Starting from an initial tetrahedra, we recursively grow each facet into three new facets, using the furthest point from the facet, similar to the quickhull algorithm. The visible points associated with the facet are redistributed to the new facets. This results in a polyhedron, and the points in the interior of the polyhedron will not be part of the convex hull. Therefore, we can prune away the points inside the polyhedron and compute the convex hull on the rest of the points.

There are several differences in our implementation from Tang et al.'s algorithm. Our implementation executes the recursive calls on different facets in parallel, whereas Tang et al.'s implementation maps the algorithm to the GPU architecture by pre-allocating space for the facets and visible points, and runs the algorithm in an iterative manner in lock-step. Specifically, successively generated facets and points points associated them are updated by multiple threads in parallel in each iteration. We use a parallel maximum-finding routine to find the furthest point of each facet in each call. Rather than growing the pseudohull until there are no more visible points as done by Tang et al., we set a threshold on the number of points associated with a facet, below which we stop growing the pseudohull. This prevents stack overflow on large and skewed data sets due to too many recursive calls, and the extra unpruned points do not contribute significantly to the work of the final computation of the convex hull. At the end of pruning, we use our parallel reservation-based quickhull algorithm to compute the final hull on the remaining points, whereas Tang et al.uses a sequential implementation.

### Parallel Divide-and-Conquer

We adopt a common parallelization strategy using divide-and-conquer, which calls our reservation-based algorithm as a subroutine. Some early convex hull algorithms are based on divide-and-conquer, notably, the algorithm by Preparata and Hong [192]. The algorithm splits the input into two spatially disjoint subsets by a mid-point along one of the axis, recursively computes the convex hull on each subset, and then merges the results together. Later work [15, 82, 20] extended this approach to the parallel setting. However, most of these approaches rely on complicated subroutines to merge convex hulls, which are not practical and have not been implemented, to the best of our knowledge.

We implement a practical divide-and-conquer algorithm by partitioning the input into $c \cdot numProc$ equal subsets, where $c$ is a small constant and $numProc$ is the

number of processors. For each subset, the convex hull of the subset is computed by a single processor using the sequential quickhull algorithm, but run in parallel across the different subsets. Then, the vertices of the outputs of the subproblems are collected to form a new input, from which the final convex hull is computed using our reservation-based parallel algorithm described in Section 8.2.

### Experimental Evaluation

**Data Sets** We use four types of synthetic data sets. The first is **Uniform** (**U**), consisting of points distributed uniformly at random inside a hypercube with side length $\sqrt{n}$, where $n$ is the number of points. The second type **UniformSphere** (**US**) is similar to the first, but the points are distributed in a hypersphere instead. We also generate ***OnSphere*** (**OS**) and ***OnCube*** (**OC**) data sets, where points are uniformly distributed on the surface of a hypersphere and a hypercube, respectively. The surfaces have a thickness equal to 0.1 times the diameter or side length of the sphere or cube. The last type is **VisualVar** (**V**), a clustered data set with variable density, produced by Gan and Tao's generator [104]. The generator produces points by performing a random walk in a local region, but jumping to random locations with some probability. For each of these two types, we generate them in 2, 3, 5, and 7 dimensions, and for up to 1 billion points. We name the data sets in the format of **Dimension-Name-Size**.

We also use the following real-world data sets from the Stanford 3D Scanning Repository [10]: ***3D-Thai-5M*** is a 3-dimensional point data set of size 4999996 from a scanned thai-statue; and ***3D-Dragon-3.6M*** is a 3-dimensional point data set of size 3609600 from a scanned statue of a dragon.

**Testing Environment** The experiments are run on an AWS c5.18xlarge instance with 2 Intel Xeon Platinum 8124M CPUs (3.00 GHz), for a total of 36 two-way hyper-threaded cores and 144 GB RAM. Our experiments use all hyper-threads unless specified otherwise. We compile our benchmarks with the `g++` compiler (version 9.3.0) with the `-O3` flag, and use ParlayLib [41] for parallelism.

We test the following implementations for convex hull (our new implementations are underlined). All implementations are for both $\mathbb{R}^2$ and $\mathbb{R}^3$, unless stated otherwise.

- ***CGAL***: sequential C++ implementation of quickhull in CGAL [96].
- ***Qhull***: sequential C++ implementation of quickhull [8] by Barber et al. [31].
- ***RandInc***: our implementation of the parallel randomized incremental algorithm described in Section 8.2.

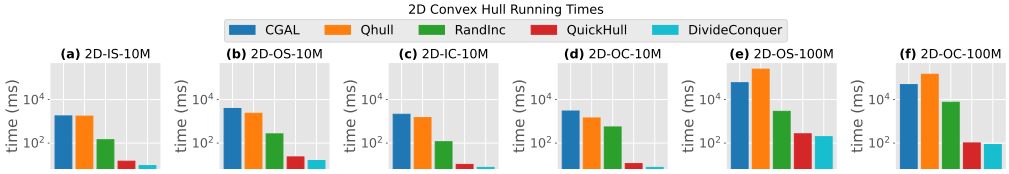



Figure 29: Running times of implementations across different data sets for 2-dimensional convex hull on 36 cores with 2-way hyper-threading.

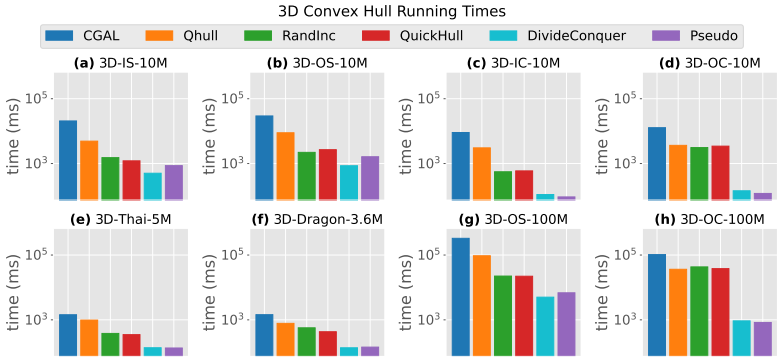



Figure 30: Running times of implementations across different data sets for 3-dimensional convex hull on 36 cores with 2-way hyper-threading.

- ***QuickHull***: for $\mathbb{R}^2$, it is a simple recursive parallel algorithm [40], and we use the implementation in PBBS [215]; for $\mathbb{R}^3$, we use our parallel quickhull algorithm described in Section 8.2.

- ***Pseudo***: our implementation of the pseudoHull heuristic proposed by Tang et al. [224] for 3-dimensional convex hull described in Section 8.2. The final stage of the computation uses our *quickHull* algorithm for $\mathbb{R}^3$.

- ***DivideConquer***: our divide-and-conquer algorithm described in Section 8.2.

We show a comparison of running times across methods using 36 cores with two-way hyper-threading in Figures 29 and 30. Our implementations achieve significant speedup compared to existing sequential implementations. Our fastest parallel implementations achieve speedups of 190–559x (325x on average) over *CGAL* for 2-dimensional convex hull, and speedups of 10.5–124x (61.4x on average) over *CGAL* for 3-dimensional convex hull. Our fastest parallel implementations have speedups of 147–1673x (605x on average) over 2-dimensional *Qhull*, and speedups of 5.68–43.8x

(19.9x on average) over 3-dimensional *Qhull*. When run using a single thread, our parallel implementations achieve speedups of 3.26–12.4x and 1.31–5.05x over *CGAL* for 2 and 3 dimensions, respectively; and 3.39–47.6x and 0.99–2.06x speedups over *Qhull* for 2 and 3 dimensions, respectively.

For $\mathbb{R}^2$, *DivideConquer* is always the fastest method due to having high scalability from processing many independent subproblems in parallel. For $\mathbb{R}^3$, the fastest two methods are *DivideConquer* and *Pseudo*. We observe that on data sets with a larger output size, *Pseudo* is slower than *DivideConquer* (Figures 30(a), (b), and (g)). This is because the final computation after pruning takes longer given that there are a higher number of remaining points after pruning. For instance, the number of remaining points for *3D-IS-10M* and *3D-IC-10M* after pruning are 83669 and 2316, respectively, and *Pseudo* is relatively slower on the former. We observe that *RandInc* and *QuickHull* take relative longer compared with the fastest methods for data sets with a smaller output size (Figures 30(c)–(e) and (h)). This is caused by higher contention during the reservation of facets, since there are fewer facets on the intermediate hull. For example, for *3D-IS-10M* and *3D-IC-10M*, the output sizes are 14163 and 423, respectively. During the computation, *3D-IC-10M* exposes fewer facets for reservation, leading to a lower success rate during the reservations.

*DivideConquer* achieves the best parallel speedup (42.78x and 16.55x on average for $\mathbb{R}^2$ and $\mathbb{R}^3$, respectively). This is because the bulk of the running time is spent in computing independent convex hulls across different threads. On the other hand, the incremental algorithms, *RandInc* and *QuickHull*, demonstrate lower scalability because of load-imbalance caused by the different amounts of work to process each conflict point being processed in parallel.

We also compare with the performance of GPU-based algorithms. Specifically, we compare with *CudaHull* proposed by Stein et al. [220]. The *CudaHull* algorithm incrementally adds batches of points in parallel, and after every batch, fixes the concavities on the polyhedron to form the convex hull. On a data set with 10 million points and 3 dimensions, Stein et al. reported that their implementation takes 89 ms on an Nvidia GeForce GTX 580 with 512 cores, where there is also a small fraction of time being spent on the CPU. Our best implementations, *DivideConquer* and *Pseudo* take 113 and 93 ms, respectively on a data set of the same size and dimension on multi-core CPUs.

## 8.3 Smallest Enclosing Ball

The smallest enclosing ball of $P$ in $\mathbb{R}^d$ is the smallest $d$-sphere containing $P$. It is well-known that the smallest enclosing ball is unique and defined by a ***support set***

of $d + 1$ points on the surface of the ball (see Figure 41(b)).

Welzl [246] showed that by using a randomized incremental algorithm, the smallest enclosing ball can be computed in $O(n)$ time in expectation for constant $d$. The algorithm iteratively expands the support set of the ball by adding points in a random order until the ball contains all of the points. The algorithm was later improved by Gartner [107] with practical optimizations for speed and robustness. Larsson et al. [154] proposed practical parallel algorithms that uses a new method for expanding the support set, and the implementation works on both CPUs and GPUs. Later, Blelloch et al. [45] proposed a parallel algorithm based on Welzl's algorithm, but without any implementations.

In this section, we describe our new algorithms for the smallest enclosing ball problem based on Larsson et al.'s approach [154]. We propose a sampling-based algorithm to quickly reduce the size of the data set. We also provide the first parallel implementation of Welzl's classic algorithm.

### Sampling-Based Algorithm

Given a ball $B$, we define ***visible points*** to be points that lie outside of $B$. Existing approaches for computing the smallest enclosing ball focus on expanding the support set in an iterative manner, and output the enclosing ball when there are no more visible points. Welzl's algorithm expands the support set by adding points in a random order [246]. In comparison, Larsson et al.'s approach scans the input to search for good support sets in a round-based manner. In $\mathbb{R}^3$, Larsson's algorithm divides the space into eight orthants centered at the center of $B$. On each round, the input is scanned to find the furthest visible points in each orthant. $B$ is then updated to the next intermediate solution using the existing support set of $B$ and the new visible points found during the scan. The algorithm iterates until there are no more visible points. It is parallelized within each round by performing the scan on the input in parallel.

We find each iteration in Larsson et al.'s algorithm to be unnecessarily expensive due to having to scan the entire data set on every round. Our approach is to use a sampling heuristic to first obtain a good initial ball, inspired by Welzl's randomized algorithm. Specifically, we use small random samples to obtain good estimates of the support set at a negligible cost.

We show the pseudocode of our algorithm in Figure 19. At a high level, our sampling-based algorithm consists of two phases: the sampling phase (Line 2–9) and the final compute phase (Line 10–15). First, we initialize the ball using a few arbitrary points (Line 1). Then, we iterate through a random permutation of the

input to take multiple samples (Line 2–9). On each iteration, we scan through a constant-sized segment of the input, which is effectively equivalent to a random sample. We perform an orthant scan similar to Larsson's approach. Our implementation of orthant scan will return a new estimate of the support set based on the sample, and a boolean *hasOutlier* indicating whether the sample contains visible points with respect to the current smallest enclosing ball $B$ (Line 4). We recompute $B$ using the new support set. If there are visible points in the current sample, we will continue the sampling process with our new $B$. If there are no visible points in the sample, the support set likely contains most of the points, and so we terminate sampling and move on to the next phase. Now, with a good estimate of the optimal smallest enclosing ball, we run Larsson's orthant scan to compute the final smallest enclosing ball (Line 10–15). The sampling phase allows us to generate good support sets without having to scan the entire input.

**Algorithm 19** Pseudocode for the parallel sampling-based algorithm for smallest enclosing ball.

**Require:** $P$ : $d$-dimensional points, $c$ : batch size
**Ensure:** $B$ : $d$-dimensional smallest enclosing ball
1: $B \leftarrow$ ball()
2: $scanned \leftarrow 0$
3: **while** $scanned < n$ **do**
4: (hasOutlier, support) $\leftarrow$ orthantScan($P[scanned : \min(scanned + c, n) - 1], B$)
5: $scanned \leftarrow scanned + c$
6: **if** not hasOutlier **then**
7: **break** ▷ current sample does not violate $B$
8: **else**
9: $B \leftarrow$ constructBall(support)
10: **while** hasOutlier **do**
11: (hasOutlier, support) $\leftarrow$ orthantScan($P, B$)
12: **if** not hasOutlier **then**
13: **return** $B$
14: **else**
15: $B \leftarrow$ constructBall(support)

We parallelize the orthant scan, which is the most expensive operation of the algorithm. Specifically, we divide the input array to orthant scan into blocks, and processed each block sequentially, but in parallel across different blocks. Afterward,

the extrema for the orthants obtained from the blocks are merged, and a new support set is computed on these points and the existing support set of $B$.

### Parallel Welzl's Algorithm and Optimizations

We also implemented and optimized the parallel version of Welzl's algorithm described by Blelloch et al. [45]. Welzl's sequential algorithm uses a random permutation of the input $P$ and processes the points one by one. If the algorithm encounters a visible point $p_i$ with respect to the current bounding ball $B$, $B$ is recomputed on $P_i$, the prefix of points up until $p_i$, using recursive calls to the algorithm. Blelloch et al.'s parallel algorithm also uses a random permutation of $P$. Across iterations, the algorithm processes prefixes of $P$ of exponentially increasing size. If the prefix contains at least one visible point, the earliest visible point $p_i$ is identified, and $B$ is recomputed on prefix $P_i$ by recursively calling the parallel algorithm. Each prefix is processed in parallel.

We implement the algorithm with some practical optimizations. When there are numerous visible points in the prefix, the work of the parallel algorithm will increase significantly, because each time a visible point is discovered, the points after the visible point in the same prefix will have to be reprocessed in the next round. Therefore, given that there will be more visible points in the initial rounds when the prefix size is small ($< 500000$), we process these prefixes sequentially by calling Welzl's sequential algorithm. This also reduces the amount of overhead from parallel primitives, since there is limited parallelism for small prefixes.

In addition, we extend existing optimizations of Welzl's sequential algorithm to the parallel setting. We implement the move-to-front heuristic [246], which upon encountering a visible point, moves the visible point to the front of $P$, so that it will be processed earlier in recursive calls, reducing the number of subsequent visible points. We also parallelize the pivoting heuristic proposed by Gartner [107]. In this heuristic, upon encountering a visible point, rather than processing the visible point directly, we search $P$ for a *pivot point* furthest away from the center of the current $B$, and use the pivot point to compute the new $B$ instead of the visible point. We use a parallel maximum-finding algorithm to identify the pivot point.

### Experimental Evaluation

For the smallest enclosing ball, we use the same experimental setup and data sets as in Section 8.2. We test the following implementations for smallest enclosing ball (our new implementations are underlined). All implementations are for both $\mathbb{R}^2$ and $\mathbb{R}^3$.

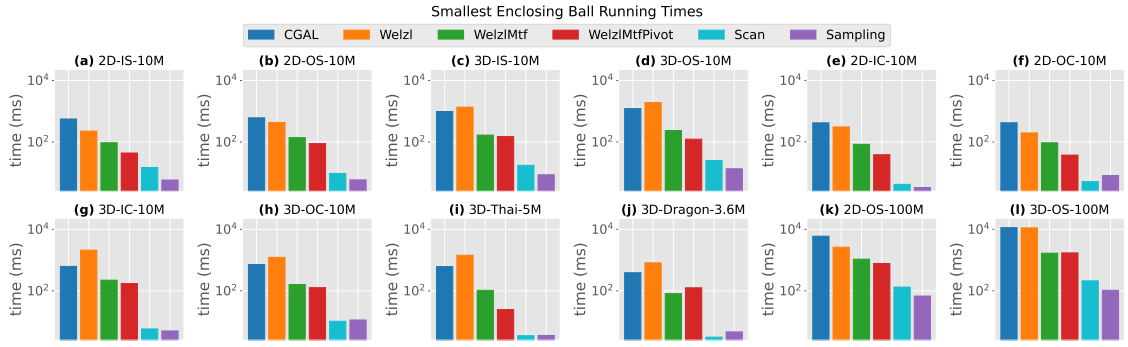



Figure 31: Running times of implementations across different data sets for smallest enclosing ball on 36 cores with 2-way hyper-threading.

- ***CGAL***: sequential C++ implementation of Welzl's algorithm in CGAL [96].
- ***Orthant-scan***: our implementation of Larsson et al. [154]'s orthant-scan algorithm.
- ***Sampling***: our parallel sampling algorithm described in Section 8.3.
- ***Welzl***: our parallel implementation of Welzl's algorithm described in Section 8.3.
- ***WelzlMtf***: the same as *Welzl*, but with the move-to-front heuristic [15].
- ***WelzlMtfPivot***: the same as *Welzl*, but with both the move-to-front and the pivoting heuristic [107].

For smallest enclosing ball, we show the comparison across implementations using 36 cores with two-way hyper-threading in Figure 31. Our fastest parallel implementations have speedups of 70–178x (109x on average) over *CGAL*. On one thread, our fastest implementations achieve speedup of 2.81–7.05x (4.96x on average) over *CGAL*.

Our sampling-based method is the fastest for eight out of the twelve data sets, whereas *Orthant-scan* without sampling is the fastest for the other four. We observe that the sampling phase on average scans only about 5% of the data set, and results in up to 2.55x (1.47x on average) speedup compared to just running *Orthant-scan*. Comparing across different implementations of Welzl's algorithms, we see that the move-to-front, and the pivoting heuristic implemented in parallel consistently improve the running times. Specifically, *WelzlMtf* is 2.09–13.9x faster than *Welzl*,

and *WelzlMtfPivot* is 3.4–58.6x faster than *Welzl*. We also see that *Sampling* and *Orthant-scan* are 4.63–34.8x and 2.96–40.3x faster than *WelzlMtfPivot*, respectively.

We also compare with existing GPU implementation of the smallest enclosing ball by Kallberg and Larsson [145]. According to Kallberg and Larsson. Their algorithm is based on repeated farthest-point queries. Their implementation on an Nvidia Quadro K4000M GPU with 960 cores takes 13 ms to process a data set with 5 million points and 3 dimensions, whereas our *Sampling* implementation takes 4.95 ms to process a data set with 10 million points and 3 dimensions.

## 8.4 Parallel Batch-dynamic Closest Pair

We consider a metric space $(S, d)$ where $S$ contains $n$ points in $\mathbb{R}^k$, and $d$ is the $L_t$-metric where $1 \leq t \leq \infty$. The ***static closest pair*** problem computes and returns the ***closest pair distance*** $\delta(S) = \min\{d(p, q) \mid p, q \in S, p \neq q\}$, and the closest pair $(p, q)$. The ***dynamic closest pair*** problem computes the closest pair of $S$, and also maintains the closest pair upon insertions and deletions of points. A ***parallel batch-dynamic*** data structure processes batches of insertions and deletions of points of size $m$ in parallel. In this thesis, we propose a new parallel batch-dynamic data structure for closest pair on $(S, d)$.

There is a rich literature on sequential dynamic closest pair algorithms [181, 223, 207, 216, 217, 208, 59, 146, 39, 111, 14]. However, as far as we know, none of the existing dynamic algorithms have been implemented and none of them are parallel. The main contribution of the thesis is the design of a theoretically-efficient and practical parallel batch-dynamic data structure for dynamic closest pair, along with a comprehensive experimental study showing that it performs well in practice. Our solution is inspired by the sequential solution of Golin et al. [111], which takes $O(n)$ space to maintain $O(n)$ points and supports $O(\log n)$ time updates, and is the fastest existing sequential algorithm for the $L_t$-metric. Our parallel solution takes a batch update of size $m$ and maintains the closest pair in $O(m(1+\log((n+m)/m)))$ expected work and $O(\log(n+m)\log^*(n+m))$ depth (parallel time) with high probability (*whp* holds with probability at least $1-1/n^c$ for an input of size $n$ and some constant $c > 0$). Compared to the sequential algorithm of Golin et al., our algorithm is work-efficient (i.e., matches the work of the sequential algorithm) for single updates, and has better depth for multiple updates since we process a batch of updates in parallel. Our data structure is based on efficiently maintaining a sparse partition of the points (a data structure used by Golin et al. [111]) in parallel. This requires carefully organizing the computation to minimize the work and depth, as well as using a new parallel batch-dynamic binary heap that we design. This is the first parallel batch-dynamic

binary heap in the literature, and may be of independent interest.

We implement our data structure with optimizations to improve performance. In particular, we combine the multiple heaps needed in our theoretically-efficient algorithm into a single heap, which reduces overheads. We also implement a parallel batch-dynamic $k$d-tree to speed up neighborhood queries. We evaluate our algorithm on both real-world and synthetic data sets. On 48 cores with two-way hyper-threading, we achieve self-relative parallel speedups of up to 38.57x across various batch sizes. Our algorithm achieves throughputs of up to $1.35 \times 10^7$ and $1.06 \times 10^7$ updates per second for insertions and deletions, respectively.

In addition, we implement and evaluate four parallel algorithms for the static closest pair problem. There has been significant work on sequential [210, 195, 37, 98, 36, 128, 110, 148, 90, 63, 30] and parallel [27, 164, 157, 47, 45] static algorithms for the closest pair. As far as we know, none of the existing parallel algorithms have been evaluated and compared empirically. We implement a divide-and-conquer algorithm [47], a variant of Rabin's randomized algorithm [195], our parallelization of the sequential sieve algorithm [148], and a randomized incremental algorithm [45]. On 48 cores with two-way hyper-threading, our algorithms achieve self-relative parallel speedups of up to 51.45x.[7] Our evaluation of the static algorithms shows that Rabin's algorithm is on average 7.63x faster than the rest of the static algorithms. Finally, we compare our parallel batch-dynamic algorithm with the static algorithms and find that it can be advantageous to use the batch-dynamic algorithm for batches containing up to 20% of the data set. Our source code is publicly available at `https://github.com/wangyiqiu/closest-pair`.

### Related Work

**Static Closest Pair.** The problem of finding the closest pair of points under the $L_t$ metric has been a long-studied problem in computational geometry. There have been several deterministic sequential algorithms [210, 37, 36, 128] that solve the problem optimally in $O(n \log n)$ time under the standard algebraic decision tree model. Under a different model where the floor function is allowed with unit-cost, Rabin [195] solved the problem in $O(n)$ expected time. Fortune and Hopcroft [98] presented a deterministic algorithm with $O(n \log \log n)$ running time under the same model. Later, Khuller and Matias [148] came up with a simple randomized algorithm that takes $O(n)$ expected time using a sieve data structure. Golin et al. [110] described a randomized incremental algorithm for the problem that takes $O(n)$ expected time.

[7] With hyper-threading, the parallel speedup can be more than the total core count.

Later, Dietzfelbinger et al. [90] filled in some details for Rabin's algorithm concerning hashing and duplicate grouping. Banyassady and Mulzer [30] gave a simpler analysis for Rabin's algorithm. Chan [63] gave an algorithm that takes $O(n)$ expected time in a randomized optimization framework. Maheshwari et al. [166] designed a new algorithm for closest pair in the doubling-metric space in expected $O(n \log n)$ time.

For parallel algorithms, Atallah and Goodrich [27] came up with the first parallel algorithm for closest pair using multi-way divide-and-conquer. The algorithm takes $O(n \log n \log\log n)$ work and $O(\log n \log\log n)$ depth. MacKenzie and Stout [164] designed a parallel algorithm inspired by Rabin [195] that takes $O(n)$ work and $O(1)$ depth in expectation. Blelloch and Maggs [47] parallelized the divide-and-conquer approach in [37, 36], and their algorithm takes $O(n \log n)$ work and $O(\log^2 n)$ depth. Recently, Blelloch et al. [45] designed a randomized incremental algorithm for the problem that takes $O(n)$ expected work and $O(\log n \log^* n)$ depth *whp* Lenhof and Smid [157] solved a close variant, the $K$-closest pair problem for the $L_t$-metric, in $O(n \log n \log\log n + K)$ work and $O(\log^2 n \log\log n)$ depth, where $K$ is the number of closest pairs to return.

**Dynamic Closest Pair.** For the sequential dynamic closest pair problem, there have been semi-dynamic algorithms that focus on only insertions or only deletions. For the deletion-only case, Supowit [223] gave an algorithm that maintains the minimal Euclidean distance for points in $k$ dimensions in $O(\log^k n)$ amortized time per deletion. The method uses $O(n \log^{k-1} n)$ space. For the insertion-only case, Schwarz and Smid [207] designed a data structure for the $L_t$-metric space that takes $O(n)$ space and handles each insertion in $O(\log n \log\log n)$ time. Schwarz et al. [208] designed an optimal data structure taking $O(n)$ space and $O(\log n)$ time per insertion for the same problem.

For sequential fully-dynamic closest pair algorithms supporting both insertions and deletions, Overmars [181] gave an $O(n)$ time update algorithm. Smid [216] gave a dynamic data structure of size $O(n)$, that maintains closest pair of $k$-dimensional points in the $L_t$-metric space, in $O(n^{2/3} \log n)$ time per update.

Later work improved the sequential running time to polylogarithmic time per update. Smid [217] used a data structure of size $O(n \log^k n)$ that maintains the closest pair in the $L_t$-metric in $O(\log^k n \log\log n)$ amortized time per update. Callahan and Kosaraju [59] presented a general technique for dynamizing problems in Euclidean-space that make use of the well-separated pair decomposition [60]. For dynamic closest pair, their algorithm requires $O(n)$ space and $O(\log^2 n)$ time per update. Kapoor and Smid [146] proposed new dynamic data structures that solve the problem in the $L_t$-metric using $O(n)$ space and deterministic $O(\log^{k-1} n \log\log n)$ update time.

Bespamyatnikh [39] described a closest pair data structure for the $L_t$-metric space that takes $O(n)$ space, and has $O(\log n)$ deterministic update time. The main idea is to dynamically maintain a fair-split tree and a heap of neighbor pairs. The algorithm does not currently seem to be practical. Golin et al. [111] described a randomized data structure for the problem in the $L_t$-metric space, which takes $O(n)$ expected space and supports insertions and deletions in $O(\log n)$ expected time. Our work is inspired by Golin et al. [111], and is the first to parallelize dynamic batch-updates for the closest pair problem.

In addition, there has been related work on similar problems. Agarwal et al. [14] proposed a kinetic and dynamic data structure for 2-dimensional all-nearest-neighbors as well as the closest pair, which uses $O(n \log n)$ space and processes each update in $O(\log^3 n)$ time. Eppstein [94] solved a stronger version of the dynamic closest pair problem by supporting arbitrary distance functions. The algorithm maintains the closest pair in $O(n \log n)$ time per insertion and $O(n \log^2 n)$ amortized time per deletion using $O(n)$ space. Cardinal and Eppstein [62] later designed a more practical version of this algorithm. Chan [64] presented a modification of Eppstein's algorithm [94], which improves the amortized update time to $O(n)$.

### Review of the Sparse Partition Data Structure

We give an overview of the sequential dynamic closest pair data structure proposed by Golin et al. [111], which is based on the serial static closest pair algorithm by Khuller and Matias [148]. Our new parallel algorithm also uses this data structure, which is referred to as the *sparse partition* of an input set.

**Sparse Partition** For a set $S$ with $n$ points, a ***sparse partition*** [111] is defined as a sequence of 5-tuples $(S_i, S'_i, p_i, q_i, d_i)$ with size $L$ $(1 \leq i \leq L)$, such that **(1)** $S_1 = S$; **(2)** $S'_i \subseteq S_i \subseteq S$; **(3)** If $|S_i| > 1$, $p_i$ is drawn uniformly at random from $S_i$, and we compute the distance $d_i = d(p_i, q_i)$, to $p_i$'s closest point $q_i$ in $S_i$; **(4)** For all $x \in S_i$: **a)** if the closest point of $x$ in $S_i$ (denoted as $d(x, S_i)$) is larger than $d_i/3$, then $x \in S'_i$; **b)** if $d(x, S_i) \leq d_i/6k$ then $x \notin S'_i$; and **c)** if $x \in S_{i+1}$, there is a point $y \in S_i$ such that $d(x, y) \leq d_i/3$ and $y \in S_{i+1}$; **(5)** $S_{i+1} = S_i \setminus S'_i$.

The sparse partition is constructed using these rules until $S_{L+1} = \emptyset$. It contains $O(\log n)$ levels in expectation, as $|S_i|$ decreases geometrically. The expected sum of all $|S_i|$ is linear [111]. We call $p_i$ the ***pivot*** for partition $i$. At a high level, $S'_i$ contains points that are far enough from each other, and the threshold $d_i$ that defines whether points are "far enough" decreases for increasing $i$. In particular, for any $1 \leq i < L$, $d_{i+1} \leq d_i/3$ as shown in Golin et al. [111]. Hence, the closest pair will likely show up

in deeper levels that do not contain many points. Based on the definition, each $S'_i$ is non-empty, and $\{S'_1, \ldots, S'_L\}$ is a partition of $S$.

**A Grid-Based Implementation of Sparse Partition** We now describe Golin et al.'s implementation of the sparse partition. There are $L$ levels of the sparse partition, and we refer to each as ***level*** $i$ for $1 \leq i \leq L$. We maintain each level using a grid data structure, similar to many closest pair algorithms (e.g., [195, 110, 148, 111]).

To represent $S_i$, we place the points into a grid $G_i$ with equally-sized axis-aligned grid boxes with side length $d_i/6k$, where $k$ is the dimension, and $d_i$ is the closest pair distance of the randomly chosen pivot $p_i$. This can be done using hashing. Denote the ***neighborhood*** of a point $p$ in $G_i$ relative to $S$ by $N_i(p, S)$, which refers to the set of points in $S \setminus \{p\}$ contained in the collection of $3^k - 1$ boxes bordering the box containing $p$, as well as $p$'s box. We say that point $p$ is ***sparse*** in $G_i$ relative to $S$ if $N_i(p, S) = \emptyset$. We use this notion of sparsity to compute $S'_i = \{p \in S_i : p \text{ is sparse in } G_i \text{ relative to } S_i\}$, which satisfies definition (4) of the sparse partition. The points in $S'_i$ are stored in a separate grid.

An example of the grid structure in two dimensions is shown in Figure 32. We illustrate the grid $G_i$ for the $S_i$ of each level, as well as the pivot $p_i$ and its closest neighbor $q_i$. The grid size is set to $d_i/6k = d(p_i, q_i)/12$. The sparse points, represented by the hollow blue circles, have empty neighborhoods, and do not have another point within a distance of $d_i/3$. The solid black circles, representing the non-sparse points, are copied to the grid $G_{i+1}$ for $S_{i+1}$. In $S_3$, all points are sparse.

To construct a sparse partition, the sequential algorithm proceeds in rounds. On round $i$, the $i$'th level is constructed. We start with $i = 1$ where $S_1 = S$, and iteratively determine the side length of grid $G_i$ based on a random pivot, and place $S_i$ into $G_i$. Then, we compute $S'_i$ based on the definition of sparsity above, and set $S_{i+1} = S_i \setminus S'_i$. The algorithm proceeds until $S_i = S'_i$ (i.e., $S_{i+1} = \emptyset$). The expected work for construction is $O(n)$ since $|S_i|$ decreases geometrically [111]. A single insertion of point $q$ starts from $S_1$, and proceeds level by level. When $q$ is non-sparse in $S_i$, it will be added to $S_{i+1}$, and can promote points from $S'_i$ to $S'_{i+1}$ if $q$ falls within their neighborhood. The insertion of $q$ will stop if it becomes sparse at some level, at which point the insertion algorithm finishes. A deletion works in the opposite direction, starting from the last level where the deleted point exists, and working its way back to level 1. Each insertion or deletion takes $O(\log n)$ expected work.

**Obtaining the Closest Pair** As observed by both Khuller and Matias [148] and Golin et al. [111], although the grid data structure rejects far pairs, and becomes

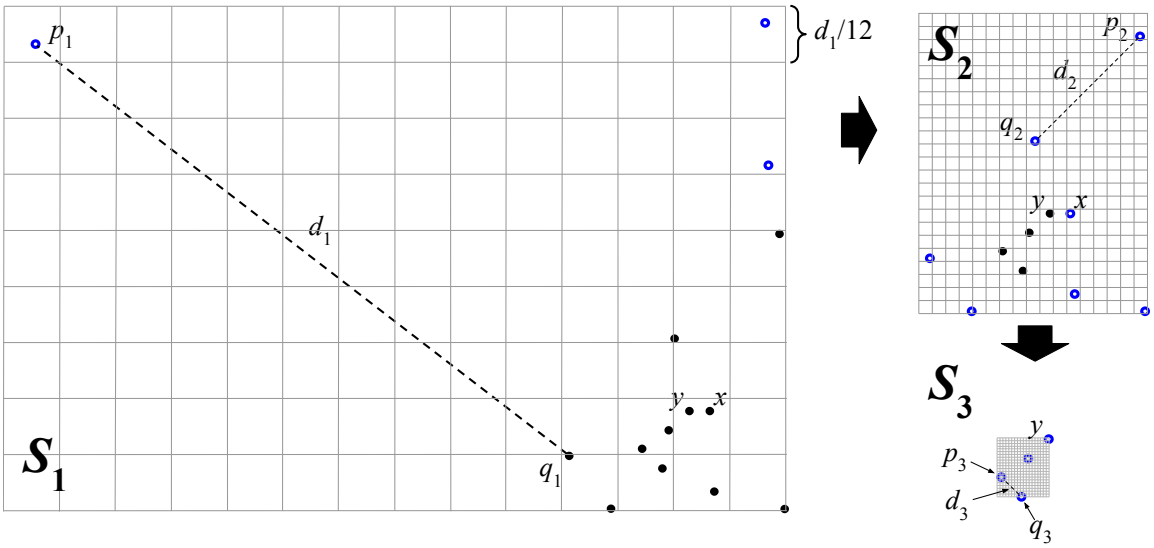



Figure 32: This figure contains an example of 14 points in $\mathbb{R}^2$, for which a grid-based sparse partition $(S_i, S'_i, p_i, q_i, d_i)$ for $1 \leq i \leq 3$ is constructed. On each level, we use a dotted line to indicate $d_i$, the Euclidean distance between the pivot $p_i$ and its closest neighbor $q_i$, and we set the grid size to be $d_i/6k = d_i/12$. We denote non-sparse points as solid black circles and sparse points as hollow blue circles. The $S'_i$ sets are represented implicitly by the set of hollow blue circles in each $S_i$. We denote the true closest pair by letters $x$ and $y$.

more fine-grained with a larger $i$, the grid at the last ($L$'th) level does not necessarily contain the closest pair. For example, as illustrated in Figure 32, $S_3$ for the last level does not contain the closest pair $(x, y)$, as $x$ is sparse on level 2 and not included in $S_3$. Therefore, we need to check more than just the last level.

The ***restricted distance*** $d^*_i(p)$ [111] is the closest pair distance from point $p$ to any point in $\bigcup_{0 \leq j \leq k} S'_{i-j}$, and defined as $d^*_i(p) := \min\{d_i, d(p, S'_{i-k} \cup S'_{i-k+1} \cup \ldots \cup S'_i)\}$, where $p \in S'_i$. Golin et al. show that $\delta(S) = \min_{L-k \leq i \leq L} \min_{p \in S'_i} d^*_i(p)$, meaning that the closest pair can be found by taking the minimum among the restricted distance pairs for all points in last $k+1$ levels of $S'_i$. For completeness, we present the following lemmas and theorem from Golin et al. [111] to show that $\delta(S) = \min_{L-k \leq i \leq L} \min_{p \in S'_i} d^*_i(p)$.

**Lemma 1.** *$d^*_i(p) > d_i/6k$ for $p \in S_i$.*

*Proof.* Let $1 \leq j \leq i$ and let $q \in S'_j$, where $q$ is an arbitrary point. Since $p \in S_j$, it

follows from properties of the sparse partition (section 8.4) that $d(p,q) \geq d(q,S_j) > d_j/6k \geq d_i/6k$. Therefore the distance from $p$ to its closest pair $d_i^*(p) > d_i/6k$. □

**Lemma 2.** $d_L/6k < \delta(S) \leq d_L$.

*Proof.* Let $\delta(S) = d(p,q)$ for some $p \in S'_i$ and $q \in S'_j$, and without loss of generality $i \leq j$. It follows from the definition of the sparse partition that $p, q \in S_i$, and we have $d(p,q) = d(p,S_i) > d_i/6k$, hence $d(p,q) > d_L/6k$.

$\delta(S) \leq d_L$ obviously holds since $d_L$ is the distance between two points. □

**Theorem 13.** $\delta(S) = \min_{L-k \leq i \leq L} \min_{p \in S'_i} d_i^*(p)$.

*Proof.* Since the restricted distance is the distance between two points, we have that $\delta(S) \leq \min_{1 \leq i \leq L} \min_{p \in S'_i} d_i^*(p)$. Let $\delta(S) = d(p,q)$ for some $p \in S'_i$ and $q \in S'_j$. Assume without loss of generality that $j \leq i$, and it is obvious that $d(p,q) = d(p, \bigcup_{h \leq i} S'_h) \geq d_i^*(p)$. Therefore $\delta(S) \geq \min_{1 \leq i \leq L} \min_{p \in S'_i} d_i^*(p)$, and hence $\delta(S) = \min_{1 \leq i \leq L} \min_{p \in S'_i} d_i^*(p)$.

We now prove that $i$ cannot be less than $L-k$. By Lemma 1, we have $\min_{p \in S'_i} d_i^*(p) > d_i/6k$. We also know from Lemma 2, and from the properties of the sparse partition ($d_{i+1} \leq d_i/3$), that for $i < L-k$, $d_i/6k \geq d_{L-k-1}/6k \geq (3^{k+1}/6k) \cdot d_L > d_L \geq \delta(S)$. Therefore, $\delta(S) \geq \min_{L-k \leq i \leq L} \min_{p \in S'_i} d_i^*(p)$. □

The sequential algorithm [111] computes the restricted distance for each point in $S'_i$, and stores them in min-heaps $H_i$, for $1 \leq i \leq L$. To obtain the closest pair, we simply read the minimum in $H_i$ for $L-k \leq i \leq L$ to obtain $k+1$ values, and then take the overall minimum. This takes $O(1)$ work and depth.

We summarize all notations used in Table 9.

### Parallel Batch-Dynamic Data Structure

For our batch-dynamic data structure, we assume that the updates are independent of the random choices made in our data structure.

**Parallel Construction** In this section, we introduce our parallel batch-dynamic algorithm, including the construction of the sparse partition (defined in Section 8.4) and how to handle batch updates.

As shown in Algorithm 20, we start with an initial point set $S$, on which we construct a grid structure recursively level by level, until all points become sparse. Starting with $S_i = S$, the algorithm works on point set $S_i$ for level $i$. We first pick a pivot point $p_i \in S_i$ and obtain its closest pair by computing distances to all other

| Notation | Definition |
|---|---|
| $k$ | Dimensionality of the data set. |
| $S$ | Point data set $\{p_1, p_2, \ldots, p_n\}$ in $\mathbb{R}^k$. |
| $n$ | Size of $S$ ($\|S\|$). |
| $m$ | Size of a batch update. |
| $d(p, q)$ | Distance between points $p, q \in S$. |
| $\delta(S)$ | $\min\{d(p, q) : p, q \in S, p \neq q\}$, i.e., the distance of the closest pair in set $S$. |
| $d(p, S)$ | $\min\{d(p, q) : q \in S \setminus p\}$, i.e., the distance of $p$ to its nearest neighbor in set $S$. |
| $d_i^*(p)$ | The restricted distance of point $p$, $d_i^*(p) := d(p, S'_{i-k} \cup S'_{i-k+1} \cup \ldots \cup S'_i)$. |
| $(S_i, S'_i, p_i, q_i, d_i)$ | The 5-tuple representing each level of the sparse partition data structure, where $S_i$ and $S'_i$ are point sets, $p_i$ is the pivot point, $q_i$ is the closest point to $p_i$ in $S_i$, and $d_i := d(p_i, q_i)$. |
| $G_i$ | The grid structure with box side length $d_i/6k$ at level $i$ of the sparse partition. |
| $H_i$ | The parallel heap associated with level $i$ of the sparse partition. |
| $L$ | The number of levels in the sparse partition. |
| $b_i(p)$ | The box containing $p$ on level $i$. |
| $b_i^\sigma(p)$ | The box with a offset of $\sigma$ relative to $b_i(p)$. $\sigma$ is a $k$-tuple over $\{-1, 0, 1\}$, where the $j$'th component indicates the relative offset in the $j$'th dimension. |
| $BN_i(p)$ | The box neighborhood of $p$, i.e., the collection of the $3^k$ boxes bordering and including the box containing $p$ on level $i$. |
| $BN_i^\sigma(p)$ | The partial box neighborhood of $p$, i.e., the intersection of $N_i(p)$ with the boxes bordering and including $b_i^\sigma(p)$. |
| $N_i(p, S)$ | The neighborhood of $p$ in set $S$, i.e., the set of points in $S \setminus p$ contained in $N_i(p)$. |

Table 9: Summary of Notation.

**Algorithm 20** Construction

**Require:** Point set $S$.
**Ensure:** A sparse partition and its associated heaps.

1: **procedure** MAIN
2: BUILD($S$, 1) ▷ Initially, $S_i := S$
3: **procedure** BUILD($S_i$, $i$)
4: Choose a random point $p_i \in S_i$.
5: Calculate $d_i := d(p_i, S_i)$, set the grid side length to $d_i/6k$, and store $p_i$'s nearest neighbor as $q_i$.
6: Create a parallel dictionary $S_i^{\mathsf{dict}}$ to store points in $S_i$ keyed by box ID.
7: In parallel, compute the box ID of each point in $S_i$ based on the grid size, and store the point in the box keyed by the box ID in $S_i^{\mathsf{dict}}$.
8: Create a parallel dictionary $S_i'^{\mathsf{dict}}$ to store points in $S_i'$ keyed by box ID.
9: In parallel, determine if each point $x$ in $S_i$ is sparse by checking $N_i(q, S_i)$ (using $S_i^{\mathsf{dict}}$). Store the sparse points in $S_i'$ and $S_i'^{\mathsf{dict}}$, and the remaining points in a new point set represented by an array $S_{i+1}$.
10: In parallel for each point $x \in S_i'$, compute $d_i^*(x)$ by checking its neighborhoods (using $S_j'^{\mathsf{dict}}$) in $S_j'$ where $i - k \le j \le i$.
11: **fork** Create a heap for $\{d_i^*(x) : x \in S_i'\}$.
12: **if** $S_{i+1}$ is not empty **then**
13: BUILD($S_{i+1}$, $i + 1$)
14: **join**

points in parallel, followed by computing the minimum to determine the side length of the grid boxes, which we use a parallel dictionary [109] to store (Lines 5–7). We check the sparsity of each point $x$ in parallel by looking up neighboring boxes using the dictionary, and store the sparse points $S_i'$ in a new parallel dictionary and the remaining points in a new point set array $S_{i+1}$ (Line 9). Then, we compute the restricted distances of all points in $S_i'$ in parallel (Line 10), and spawn a thread to asynchronously construct the heap $H_i$ to store the restricted distances (Line 11). We recursively call the construction procedure on $S_{i+1}$ to construct the next level until all points in a level are sparse (Line 13).

We now present the analysis for the parallel construction algorithm. For each call to BUILD, given $O(|S_i|)$ points, insertions to the parallel dictionary take $O(|S_i|)$ work and $O(\log^* |S_i|)$ depth *whp* Line 5 computes the distance of $p_i$ to each $q \in S_i$, taking $O(|S_i|)$ work and $O(1)$ depth. Then we obtain $q_i$ via a parallel minimum computation in the same work and depth. Checking the sparsity of points takes

$O(|S_i|)$ work and $O(1)$ depth, since each point checks $3^k$ boxes to see whether any are non-empty (Line 7). Therefore, except for the cost of Line 11 and the recursive call on Line 13, each call to BUILD takes $O(|S_i|)$ expected work and $O(\log^* |S_i|)$ depth *whp*. Line 11 creates a parallel heap with $O(|S_i'|)$ entries, which takes $O(|S_i'|)$ work and $O(\log |S_i'|)$ depth, which we prove in section 8.4. Since $\sum |S_i| = O(n)$ [111], the total work across all calls to BUILD is hence $O(n)$ in expectation. Since our heap is of linear size, the total space usage of our data structure is also $O(n)$ in expectation.

We now prove that the algorithm has polylogarithmic depth by first showing Lemma 3.

**Lemma 3.** *Algorithm 20 makes* $O(\log n)$ *calls to* BUILD whp *and the sparse partition has* $O(\log n)$ *levels* whp

*Proof.* We show that with at least 1/2 probability, $|S_{i+1}| \leq |S_i|/2$. Consider relabeling the points in $S_i = \{r_1, r_2, \ldots, r_{|S_i|}\}$ such that $d(r_1, S_i) \leq d(r_2, S_i) \leq \ldots \leq d(r_{|S_i|}, S_i)$. If we pick the pivot $p_i = r_j$, then for every $r_k$ with $k > j$, we have $d(r_j, S_i) \leq d(r_k, S_i)$, and so $r_k$ is not in $S_{i+1}$. Since the pivot is chosen randomly, it can be any $r_k$ with equal probability, and with 1/2 probability $k \leq |S_i|/2$, implying that $|S_{i+1}| \leq |S_i|/2$.

By definition, we have at most $\log_2 n$ levels where the set size decreases by at least a half. We define such levels as *good levels*. We want to analyze the number of pivots needed to be chosen until we have $\log_2 n$ good levels, at which point there are no additional levels in the data structure. Let $X$ be a random variable denoting the number of levels needed until we see $\log_2 n$ good levels. $X$ follows a negative binomial distribution with parameters $k = \log_2 n$ successes, and $p = 1/2$ probability of success.

Since pivots are chosen independently across the levels, we can use the following form of the Chernoff bound for variables following a negative binomial distribution [122]:

$$\Pr[X > L] \leq e^{-\delta^2 k/(2(1-\delta))}, \text{ where } 0 < \delta < 1 \text{ and } L = \frac{k}{(1-\delta)p}.$$

If we set $\delta = 7/8$, we have $L = 16 \log_2 n$ and

$$\Pr[X > 16 \log_2 n] \leq e^{-(7/8)^2 \log_2 n/(2(1-7/8))} = e^{-(49/16)\log_2 n} < n^{-3}.$$

This means that the sparse partition has no more than $16 \log_2 n = O(\log n)$ levels *whp* which also bounds the number of recursive calls to BUILD in Algorithm 20. □

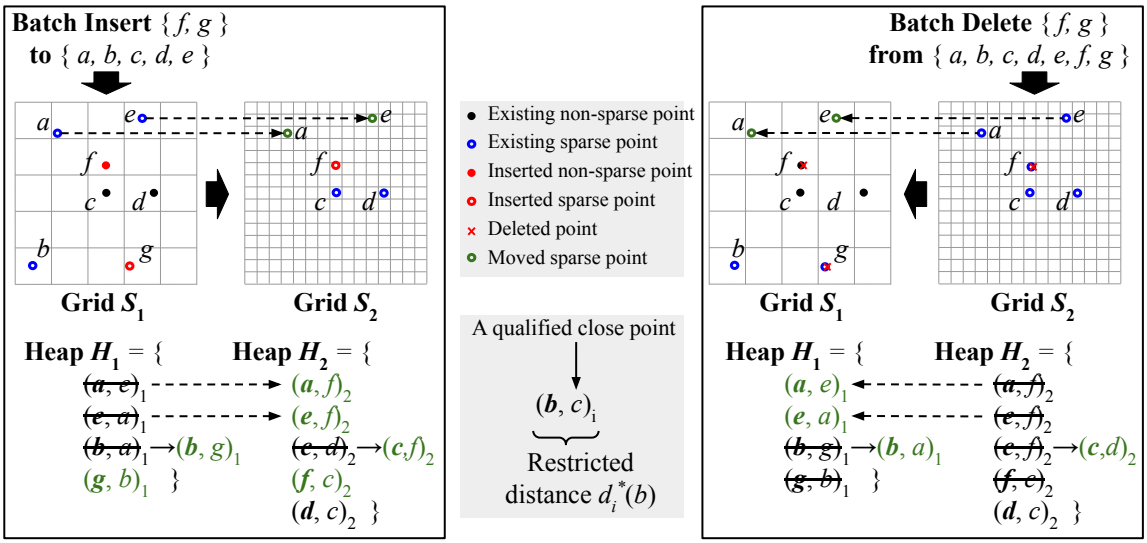



Figure 33: The figure illustrates the interaction between our parallel batch-dynamic insertion (left) and deletion (right) algorithms with the data structure. For ease of illustration, we do not show all the points in the data set. We show the data structure with two levels, and explicitly show $S_i$ and $H_i$ for each level. The grid structure in the upper half of the figures determines the sparsity of points. We represent different types of points as defined in the middle legend. In the lower half of the figures, we show the heaps with the restricted distances that they store. We show the pair defining the restricted distance of a sparse point $x$ on level $i$ as $(\boldsymbol{x}, y)$ if another point $y$ is the closest sparse point to $x$ in levels $i - j$ where $0 \leq j \leq 2$. For both insertion and deletion, we annotate the direction of the update between grids using large bold arrows (i.e., insertion starts with $S_1$ and deletion starts with $S_2$). We indicate the movement of points and heap entries using dotted arrows.

As mentioned earlier, the depth of each call to BUILD is $O(\log^* n)$ *whp* excluding the cost for heap construction and the recursive call. Since the heap insertions are asynchronous, they take a total of $O(\log n)$ depth. Therefore, the total depth of Algorithm 20 is $O(\log n \log^* n)$ *whp*

**Theorem 14.** *We can construct a data structure that maintains the closest pair containing $n$ points in $O(n)$ expected work, $O(\log n \log^* n)$ depth* whp *and $O(n)$ expected space.*

**Algorithm 21** Batch Insert

**Require:** $(S_i, S'_i, p_i, q_i, d_i)$ and $H_i$ for $1 \leq i \leq L$; a batch $Q$ to be inserted.

1: **procedure** MAIN
2: INSERT($Q$, $\emptyset$, 1)
3: **procedure** INSERT($Q_i$, $down_i$, $i$)
4: $(Q_{i+1}, down_{i+1}) :=$ GRIDINSERT($Q_i$, $down_i$, $i$))
5: HEAPUPDATE($i$)
6: **if** $(Q_{i+1} \cup down_{i+1}) \neq \emptyset$ **then**
7: INSERT($Q_{i+1}$, $down_{i+1}$, $i+1$)
8: **procedure** GRIDINSERT($Q_i, down_i, i$)
9: Determine if $p_i$, $q_i$, and $d_i$ should change when inserting $Q_i$ and $down_i$, which happens with probability $(|Q_i|+|down_i|)/(|Q_i|+|down_i|+|S_i|)$, or if a new point is closer to $p_i$ than the previously closest point $q_i$.
10: If $p_i$, $q_i$, or $d_i$ change on Line 9, or if $i > L$, call BUILD($Q_i \cup down_i \cup S_i, i$) to rebuild subsequent levels, and terminate the batch insertion.
11: Insert each point in $down_i$ and $Q_i$ into the dictionary of $S_i$ in parallel.
12: For each point $x$ in $Q_i$ in parallel, check if it is sparse in $S_i$. If so, insert $x$ into the dictionary of $S'_i$, and otherwise, insert $x$ into $Q_{i+1}$.
13: For each point $x$ in $down_i$ in parallel, check if it is sparse in $S_i$. If so, insert $x$ into the dictionary of $S'_i$, and otherwise, insert $x$ into $down_{i+1}$.
14: In parallel, for each point $x$ in $Q_i$, and for each point $r$ in the neighborhood $N(x, S'_i)$, delete $r$ from $S'_i$, and insert $r$ into $down_{i+1}$.
15: **return** $(Q_{i+1}, down_{i+1})$;

**Parallel Insertion** Next, we present our parallel algorithm that processes a batch $Q$ of $m$ insertions or deletions. For $m \geq n$, we can simply rebuild the data structure on all of the points using theorem 14 to obtain the desired bounds. We now describe the case for $m < n$. For batch updates, there are two main tasks: updating the grid and updating the heap. We first describe updating the grid. We let $Q_i$ be the subset of points in $Q$ that are inserted at level $i$, and $down_i$ be the set of points that move from level $i-1$ to level $i$ due to the insertion of $Q_i$. We start with a simple example of an insertion in fig. 33 (left), which originally contains five points $\{a, b, c, d, e\}$. For simplicity, we assume that the pivot remains unchanged and also omits $S'_i$. $Q_1 = \{f, g\}$ is the set of points inserted into the grid at level 1. We first update $S_1$ to include $f$ and $g$, and then update $S_2$ to include $Q_2 = \{f\}$ but not $g$, since $g$ is already sparse in $S_1$. In the example, the insertion of $Q_1$ triggers further point movements of $\{a, e\}$ from level 1 to level 2, as the sparse points $a$ and $e$ in $S_1$

become non-sparse due to the insertion of $f$.

As shown in Algorithm 21, the update proceeds recursively level by level (Lines 3–7). Each call to the procedure INSERT$(Q_i, \mathit{down}_i, i)$ updates $(S_i, S'_i, p_i, q_i, d_i)$ and $H_i$. Initially, $Q_1 = Q$ and $\mathit{down}_1$ is empty, as shown on Line 2. For each level $i$, we update the pivot and rebuild the level with probability $(|Q_i|+|\mathit{down}_i|)/(|Q_i|+|\mathit{down}_i|+|S_i|)$ to ensure that the pivot is still selected uniformly at random among the points in $S_i$, and we also update the pivot if a new point is closer to $p_i$ than the previous closest point $q_i$ (Lines 9–10). Otherwise, we insert the points in both $\mathit{down}_i$ and $Q_i$ into the dictionary representing $S_i$. We then check if the points that we inserted are sparse, and insert the sparse ones into the dictionary representing $S'_i$. The points that are not sparse will be added to sets $\mathit{down}_{i+1}$ and $Q_{i+1}$ and passed on to the next level (Lines 11–13). We then determine additional elements of $\mathit{down}_{i+1}$ by including the points in the neighborhood of $Q_i$ in $S'_i$ (Line 14). In general, $\mathit{down}_{i+1}$ is computed by $\mathit{down}_{i+1} = \{x \mid x \in N_i(q, S'_i \cup \mathit{down}_i)$ for some $q \in Q_i\}$. If $Q_{i+1}$ and $\mathit{down}_{i+1}$ are empty, nothing further needs to be done for subsequent levels, and the tuples $(S_l, S'_l, p_l, q_l, d_l)$ for $i < l \leq L$ remain unchanged.

We now argue that the algorithm is correct. Consider a round $i$ that inserts a non-empty $Q_i \cup \mathit{down}_i$. After the insertion, the pivot is still chosen uniformly at random, since on Line 9, we choose $p_i$ such that each point in $S_i \cup Q_i \cup \mathit{down}_i$ has the same probability of being chosen. All sparse points in $Q_i$ and $\mathit{down}_i$ inserted into $S_i$ are included in $S'_i$ (Lines 12–13). Line 14 additionally ensures that all points that were originally sparse in $S'_i$, but are no longer sparse after the insertion, are removed from $S'_i$. Given that the non-sparse points in the original $S_i$ will not become sparse due to the batch insertion, $S'_i$ must contain exactly all of the sparse points of the updated $S_i$.

**Analysis.** In our algorithm, each point can be moved across multiple levels as a result of a batch insertion.

**Lemma 4.** $|\bigcup_{1\leq i\leq L} \mathit{down}_i| \leq m \cdot 3^k = O(m)$

*Proof.* We want to prove that the number of points moved across sparse partitions for the insertion of $Q$ with $m$ points is $O(m)$, i.e., $|\bigcup_{1\leq i\leq L} \mathit{down}_i| \leq m \cdot 3^k = O(m)$. Our proof shares some notation with Golin et al. [111]. For a level $i$ and a point $q \in Q$, we let $\mathit{down}_i(q) = \mathit{down}_i \cap N_i(q, S'_i)$. Let $b_i(q)$ denote the box that contains point $q$. Let the ***box neighborhood*** of $q$ in $G_i$, denoted by $BN_i(q)$, consist of $b_i(q)$ itself and the collection of $3^k - 1$ boxes bordering $b_i(q)$. We number the $3^k$ boxes in $BN_i(q)$ as a $k$-tuple over values $\{-1, 0, 1\}$, where the $j$'th component indicates the relative offset of the box with respect to $b_i(q)$ in the $j$'th dimension. We denote the box with a relative offset of $\sigma$ with respect to $b_i(q)$ as $b_i^\sigma(q)$, where $\sigma$ is the $k$-tuple


Figure 34: (a) shows $BN_j(q)$, the box neighborhood of point $q$. Box $b_j^{(1,1)}(q)$ is marked in red. The partial box neighborhood $BN_j^{(1,1)}(q)$ is shaded. (b) shows that $b_{j+1}^{(1,1)}(q)$ is spatially contained in $BN_j^{(1,1)}(q)$.

(see fig. 34a). We further define the ***partial box neighborhood*** of a point $q$, denoted by $BN_i^\sigma(q)$, as the set of boxes in $BN_i(q)$ that intersect with the boxes bordering on and including $b_i^\sigma(q)$ (see fig. 34a).

We first show that $\sum_{l>j} |down_l(q)| \leq 3^k$ for a single insertion $q$ with respect to level $j$. Let $x \in down_{j+1}(q)$ for some level $j$. By definition, $x$ is in a box of $BN_j(q)$ for some $q \in Q$ and $N_j(x, S) = \emptyset$. Therefore, the boxes in the partial box neighborhood $BN_j^\sigma(q)$ contain no points other than $x$ itself. Suppose $x$ is in the box $b_j^\sigma(q)$. Now consider the box neighborhood of $x$ in level $l$ where $l > j$, and for the sake of contradiction, suppose there is some other point $y \in b_l^\sigma(q)$. Since $d_l \leq d_{j+1} \leq d_j/3$ by properties of the sparse partition, $b_l^\sigma(q)$ is spatially contained in $BN_j^\sigma(q)$, and hence we have $y \in BN_j^\sigma(q)$, which is a contradiction (see fig. 34b). Therefore, for any level $l > j$, there cannot be any point in $down_l(q)$ with signature $\sigma$ except for $x$. For any $q \in Q$, since the number of partial box neighborhoods $BN_l^\sigma(q)$ across all value of $\sigma$ that do not share any points with each other is at most $3^k$, we have that $\sum_{l>j} |down_l(q)| \leq 3^k$.

We have shown that the number of points moved by a single insertion across levels is at most $3^k$, and now we extend the argument to batch insertion. Consider another point $p \in Q$ inserted in the same batch as $q$. If $BN_j(p)$ and $BN_j(q)$ do not share any boxes, then the argument above holds for $p$ and $q$ independently. We are concerned with the case where $BN_j(p)$ and $BN_j(q)$ overlap. However, analyzing $p$ and $q$ separately can only overcount the point $x$ in the above argument if it appears in the partial box neighborhoods of both $BN_j^{\sigma'}(p)$ and $BN_j^\sigma(q)$ for some $\sigma$ and $\sigma'$. This argument can be extended to an arbitrary subset of $Q$ beyond $p$ and $q$ in a similar manner. Therefore, we can analyze each point separately to get an upper

bound of $\sum_{q \in Q, 1 \le i \le L} |down_i(q)| \le m \cdot 3^k = O(m)$. □

**Lemma 5.** $\sum_{1 \le i \le L} E[|Q_i|] = O(m)$

*Proof.* Consider points $r$ in $Q_i$ in an increasing order of $d(r, S_i \cup Q_i)$. There is a $1/2$ chance that the pivot $p_i$ is chosen such that $d(p_i, S_i \cup Q_i)$ is not larger than that of at least half of the points in $Q_i$, making them sparse and not in $Q_{i+1}$. Therefore $|Q_{i+1}| \le |Q_i|/2$ in expectation. Given that $|Q_1| = O(m)$, we know $\sum_{1 \le i \le L} E[|Q_i|] = O(m)$. □

We obtain Lemmas 6 and 9.

**Lemma 6.** *We can maintain a sparse partition for a batch of $m$ insertions in $O(m)$ amortized work in expectation and $O(\log(n+m)\log^*(n+m))$ depth* whp

*Proof.* The expected cost of rebuilding on Line 10 summed across all rounds is proportional to the batch size. First, we re-select the pivot and rebuild with probability $(|Q_i| + |down_i|)/(|S_i| + |Q_i| + |down_i|)$. When the pivot $p_i$ is unchanged, it may update its closest point to $q_i^*$ from $Q_i \cup down_i$. It is easy to show that $q_i^*$ can be the nearest neighbor of at most $3^k - 1$ points in $S_i$. Hence, considering all candidates $Q_i \cup down_i$, it follows that they can be the nearest neighbors to $O(3^k \cdot (|Q_i| + |down_i|))$ points in $S_i$. Therefore, the pivot distance changes with probability at most $3^k \cdot (|Q_i| + |down_i|)/|S_i|$, in which case we rebuild the sparse partition. The expected work of rebuilding at level $i$ is $O((|S_i| + |Q_i| + |down_i|) \cdot ((|Q_i| + |down_i|)/(|S_i| + |Q_i| + |down_i|) + 3^k \cdot (|Q_i| + |down_i|)/|S_i|)) = O(m)$. As we terminate the insertion algorithm when a rebuild occurs, the rebuild can occur at most once for each batch, which contributes $O(m)$ in expectation to the work and $O(\log(n+m)\log^*(n+m))$ *whp* to the depth by Theorem 14.

For the rest of the algorithm, in terms of work, Line 11–13 does work proportional to $O(\sum_i (|Q_i| + |down_i|)) = O(m)$ across all the levels due to Lemmas 4 and 5. On Line 14, the number of points in the neighborhood $N_i(x, S_i')$ of each $x$ is upper bounded by $3^k$ since the points in $S_i'$ are sparse, therefore it takes $O(3^k \cdot m) = O(m)$ expected work. Note that the work is amortized due to resizing the parallel dictionary when necessary. In terms of depth, looking up and inserting points takes $O(\log^*(n+m))$ depth using the parallel dictionary. Therefore, all operations in Lines 11–14 takes $O(\log^*(n+m))$ depth, and across all $O(\log(n+m))$ *whp* rounds, the total depth is $O(\log(n+m)\log^*(n+m))$ *whp*. □

**Algorithm 22** Batch Delete

**Require:** $(S_i, S'_i, p_i, q_i, d_i)$ and $H_i$ for $1 \le i \le L$; a batch $Q$ to be deleted.

**function** MAIN
  Determine $Q_i$ for all $1 \le i \le L$. Specifically, for each level from $L$ to 1, compute $Q_i = Q \cap S_i$.
  DELETE($\emptyset$, $L$).
  Rebuild from the level with the smallest $i$ that needed a rebuild. Specifically, call BUILD($S_i, i$).
**procedure** DELETE($up_i$, $i$)
  $up_{i-1}$ := GRIDDELETE($up_i$, $i$).
  HEAPUPDATE($i$).
  **if** $i - 1 \ge 1$ **then**
    DELETE($up_{i-1}$, $i - 1$).
**procedure** GRIDDELETE($up_i$, $i$)
  Determine if $p_i$, $q_i$, or $d_i$ should change after deleting $Q_i$, which happens if at least one of $p_i$ or $q_i$ is in $Q_i$. If so, mark level $i$ for rebuild.
  Insert each point in $up_i$ into the dictionary of $S'_i$ in parallel.
  For each point $x$ in $Q_i$ in parallel, delete $x$ from $S_i$ and $S'_i$.
  For each point $r$ in $N_i(x, S_i)$ where $x \in Q_i$, check $N_{i-1}(r, S_{i-1})$. If $N_{i-1}(r, S_{i-1}) \subseteq Q_{i-1}$, then delete $r$ from $S_i$ and $S'_i$, and insert $r$ into the set $up_{i-1}$.
  **return** $up_{i-1}$

**Parallel Deletion** Deletions work similarly in the reverse direction as shown in Figure 33 (right). The pseudocode for our batch deletion algorithm is shown in Algorithm 22. It takes as input $(S_i, S'_i, p_i, q_i, d_i)$ and $H_i$ for $1 \le i \le L$, and a batch of points $Q$ to be deleted. We update the data structure level by level similar to the insertion algorithm, but in the opposite direction, starting at the last level $L$. We define $Q_i$ for level $i$ as $Q \cap S_i$. From the property of the sparse partition, we have $Q_j \subseteq Q_i$ for all $i < j \le L$. At each level, we delete each point in $Q_i$ from $S_i$, and also from $S'_i$ if it exists.

While the insertion algorithm moves sets of points $down_i$ from level $i-1$ to level $i$, the deletion algorithm moves points in the opposite direction, from level $i+1$ to $i$. We define $up_i$ to be the set of points that move from level $i+1$ to level $i$, i.e., $up_i = \{x \in S_{i+1} : N_i(x, S_i) \subseteq Q_i\}$. They are the points $x$ in $S_{i+1}$ that only contain points from $Q_i$ in their neighborhoods $N_i(x, S_i)$ in level $i$; when $Q_i$ is deleted, they will become sparse in $S_i$, and will no longer be in $S_{i+1}$. Eventually, the points in

$up_i \setminus up_{i-1}$ are added to both $S_i$ and $S'_i$.

Initially, we determine $Q_i$ for all levels via a backward pass starting from level $L$ (Line 2). Given $Q_i \subseteq Q_j$ for $i > j$, when a point is added to $Q_i$, it will be added to all $Q_j$ where $j < i$. We pass an empty $up_L$ to procedure DELETE (Line 3). In the procedure DELETE (Line 5), the algorithm performs the deletion from the grid at level $i$ (Line 6), updates the heap (Line 7), and then recursively calls DELETE on level $i-1$ until deletion is complete on level 1 (Line 9). Like in the insertion algorithm, we determine whether to rebuild at each level, but unlike insertion we delay the rebuild until the end of the algorithm (Line 4). We call rebuild just once, on the level with the smallest $i$ that needs a rebuild (as this will also rebuild all levels greater than $i$).

In the procedure GRIDDELETE($up_i, i$), we determine if the pivot needs to change based on whether at least one of $p_i$ and $q_i$ are in $Q_i$. If so, we mark level $i$ for rebuilding (Line 11). We then insert $up_i$ into $S'_i$ and delete the points in $Q_i$ from $S_i$ and $S'_i$ if they exist (Lines 12–13).

We determine $up_{i-1}$ by finding the points that will become sparse in level $i-1$ (Line 14). Since the movement of $up_{i-1}$ from level $i$ to $i-1$ is due to the deletion of $Q_i$, we enumerate the candidates for $up_{i-1}$ from $N_i(x, S_i)$ where $x \in Q_i$. Then, for each candidate $r$, we check if $N_{i-1}(r, S_{i-1})$ only consists of points in $Q_{i-1}$, which are to be deleted in $i-1$. If so, $r$ will move up to a level less than or equal to $i-1$, and so we add $r$ to $up_{i-1}$. A few details need to be noted to make the computation of $up_{i-1}$ take $O(m)$ work. First, when checking the neighborhood $N_i(x, S_i)$ for the candidates $r$, we should only check a neighboring box if it contains at most one point, since otherwise the candidate would not be sparse in $S_{i-1}$. This bounds the work of enumerating candidates to $O(3^k \cdot m)$. We next describe the process for checking for each candidate $r$ whether $N_{i-1}(r, S_{i-1})$ contains only points in $Q_{i-1}$. Naively checking all of the points in the neighborhoods of each candidate could lead to quadratic work. In our algorithm, we use a parallel dictionary to implement a temporary, empty grid structure with grid size equal to that of $S_{i-1}$. We insert points in $Q_{i-1}$ into this grid in parallel. This takes $O(|Q_i|)$ work and $O(\log^* m)$ depth per level. Now for each candidate $r$, we compare the number of points in each box of $N_{i-1}(r, S_{i-1})$ to the corresponding box in the temporary grid, and if they are equal, then we know that the box only contains points in $Q_{i-1}$.

**Analysis.** Since the probability of a rebuild at each level $i$ is $|Q \cap S_i|/|S_i|$ and the work for the rebuild is $O(|S_i|)$, the expected work of rebuilding at level $i$ is $O(|Q|) = O(m)$. Since we do at most one rebuild across all levels, it contributes $O(m)$ in expectation to the work and $O(\log(n+m)\log^*(n+m))$ *whp* to the depth.

The total size of $Q_i$ and $up_i$ across all of the levels is proportional to the batch

size. Since the point movement is the exact opposite of that of batch insertion, the proof is very similar. We omit the proof and just present the lemmas below.

**Lemma 7.** $|\bigcup_{1\le i\le L} up_i| \le m \cdot 3^k = O(m)$

**Lemma 8.** $\sum_{1\le i\le L} E[|Q_i|] = O(m)$

For the rest of the algorithm, not including the rebuild and heap update, it follows from Lemmas 7 and 8 and a similar analysis to the insertion algorithm that Lines 12–14 take $O(m)$ amortized work in expectation and $O(\log(n+m)\log^*(n+m))$ depth across all levels. Therefore, we can maintain a sparse partition under a batch of $m$ deletions in $O(m)$ amortized work in expectation and $O(\log(n+m)\log^*(n+m))$ depth *whp*, as stated in Lemma 9. We describe the cost of the heap update in Appendix 8.4.

**Lemma 9.** *We can maintain a sparse partition for a batch of $m$ deletions in $O(m)$ amortized work in expectation and $O(\log(n+m)\log^*(n+m))$ depth* whp

**Maintaining the Heaps $H_i$** Now we describe the parallel updates of min-heaps $H_i$ associated with each level $i$ of the sparse partition. Recall that $H_i$ contains the restricted distances $d_i^*(q)$ for $q \in S'_i$. By definition, $d_i^*(q)$ is the closest distance of $q$ to another point in $S'_{i-l}$ where $0 \le l \le k$ ($k$ is the dimensionality). Therefore, following an update on $S'_i$, we need to update the $d_i^*(q)$'s in $H_{i+l}$ for $0 \le l \le k$, and $q \in S'_{i+l}$. We use same example in Figure 33 (left), where we denote the restricted distance of point $x$ as $(\boldsymbol{x}, y)_i = d_i^*(x) = d(x,y)$, where $y \in \bigcup_{0\le j\le k} S'_{i-j}$ is another point that defines $x$'s closest distance. As shown in Figure 33 (left), due to the insertion of the sparse point $g$ to $S_1$, entry $(\boldsymbol{g}, b)_1$ is added to $H_1$. Some entries in $H_1$ are moved due to the point movements, e.g., $(\boldsymbol{a}, e)_1$ from $H_1$ is moved and updated to $(\boldsymbol{a}, f)_2$ in $H_2$ because $a$ has moved from $S_1$ to $S_2$, and $f$ is now closer. Some entries are updated, e.g., $(\boldsymbol{c}, d)_2$ is updated to $(\boldsymbol{c}, f)_2$ in $H_2$ since the new point $f$ is closer to $c$ than $d$.

On each level $i$, we maintain a parallel min-heap $H_i$ storing the restricted distances for each point in $S'_i$. In this section, we elaborate on the HEAPUPDATE procedure on Line 5 of Algorithm 21 and Line 7 of Algorithm 22. Each call to HEAPUPDATE($i$) updates $H_{i+l}$ for $0 \le l \le k$ so that they contain the updated restricted distances in level $i$.

**Definitions.** Here we define some terms that we use in the algorithm description and analysis. During a batch insertion, we process each level $i$ with inputs $Q_i$ and $down_i$ (Algorithm 21). By definition, $down_i$ contains the points moved from level $i-1$ to levels $i$ and greater. We say that point $x$ ***starts moving*** at level $i$ if $x \in$

**Algorithm 23** Naive Heap Update

**Require:** $(S_i, S'_i, p_i, q_i, d_i)$ and $H_i$ with updated grids; point set $M_1$ that start moving at level $i$ and point set $M_2$ that stop moving at level $i$.

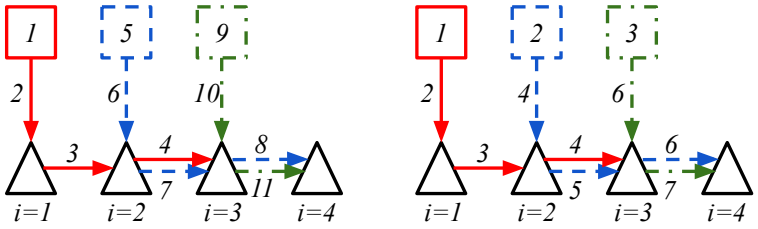
1: **procedure** HEAPUPDATE-NAIVE($i$)
2: Batch delete $d_i(p)$ $\forall p \in M_1$ from $H_i$.
3: **for** $0 \leq l \leq k$ **do**
4: Batch delete $d_{i+l}(q)$ from $H_{i+l}$ such that $d^{i+l}(q) = d(q, p)$ for some $p \in M_1$.
5: In parallel, recompute $di + l(q)$ using the grid of $Si + l$ for all $q$ whose old $di + l(q)$ was just deleted.
6: Batch insert new $d_{i+l}(q)$ for all $q$ into $H_{i+l}$.
7: Compute and batch insert $d_i(p)$ $\forall p \in M_2$ into $H_i$.
8: **for** $0 \leq l \leq k$ **do**
9: Batch delete from $H_i$ the $d^{i+l}(q)$ for each point $q \in Si + l'$, if $d(q, p) < d_{i+l}(q)$ for some $p \in M_2$.
10: Batch insert into $H_i$ the new $d_{i+l}(q) := d(q, p)$ for each aforementioned point $q$ on the previous line.

$down_{i+1} \setminus down_i$. We say that point $x$ ***stops moving*** at level $i$ if $x \in down_i \setminus down_{i+1}$, or if point $x \in Q_i \setminus Q_{i+1}$, i.e., $x$ is sparse and stays in $S'_i$. Finally, point $x$ ***moves through*** level $i$ if it is in $down_i \cap down_{i+1}$.

During a batch deletion, we process each level $i$ with input $up_i$, and delete points in $Q$ from the level if they exist (Algorithm 22). Similar to insertion, $up_i$ contains the points moved from level $i + 1$ to levels $i$ and less. We say that point $x$ starts moving at level $i$ if $x \in up_{i-1} \setminus up_i$; or if $x \in Q$ is deleted from $S'_i$. We say that point $x$ stops moving at level $i$ if $x \in up_i \setminus up_{i-1}$. Finally, we say that point $x$ moves through level $i$ if it is in $up_i \cap up_{i-1}$.

**Parallel Update.** The heap $H_i$ contains the restricted distance $d^*_i(q)$ for $q \in S'_i$. By definition, $d^*_i(q)$ is the closest distance of $q$ to another point in $S'_{i-l}$ where $0 \leq l \leq k$ ($k$ is the dimension of the data set). Therefore, following an update on $S'_i$, we need to update the $d^*_i(q)$ in $H_{i+l}$ for $0 \leq l \leq k$, and $q \in S'_{i+l}$. Specifically, the update happens when $d^*_i(q) = d(q, p)$, but $p$ starts moving at level $i$; or when $p$ stops moving at level $i$ and $d(q, p) < d^*_i(q)$. Since the update of $S'_i$ initiates the update on some heap $H_{i+l}$ for $0 \leq l \leq k$, we call level $i$ the ***initiator*** and each heap $H_{i+l}$ a ***receptor*** of the initiator.

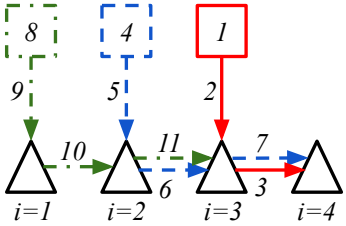
We first start with a more intuitive but less parallel algorithm, which is shown in Algorithm 23. It takes as input the updated sparse partition $(S_i, S'_i, p_i, q_i, d_i)$, the set of points $M_1$ that start moving at level $i$, and a set of points $M_2$ that stop moving



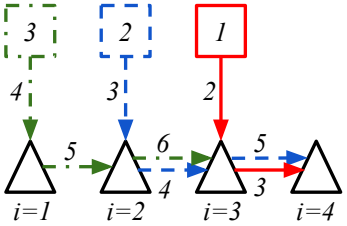

(a) Naive insertion.




(b) Parallel insertion.




(c) Naive deletion.




(d) Parallel deletion.

Figure 35: This figure shows examples of how heap updates work during insertion and deletion. Calls to GridInsert and GridDelete are shown by the boxes. Calls to HeapUpdate are shown by the arrows, whereas the actual heaps $H_i$ are shown by the triangles. The example considers dimension $k = 2$ and shows $H_i$ for $i = 1, 2, 3, 4$, but considers only updating levels $i = 1, 2, 3$. We number the calls by the order that they happen, and two calls have the same number if they can be done in parallel. For clarity, we annotate the operations associated with different levels in different colors and line-styles.

at level $i$. Lines 2–6 process the set of points $M_1$. We first batch delete $d_i^*(p)$ from $H_i$ for all $p$ in $M_1$, since they start moving at level $i$ (Line 2). We then update each receptor heap if it stores some $d_i^*(q)$ that is generated by a deleted point $p \in M_1$. To know each potential point $q$, we iterate over the neighborhood of each $p$ in $M_1$ and then check if $d_i^*(q)$ needs to be updated (Lines 3–6). Lines 7–10 process $M_2$. We compute new restricted distances and batch insert the points in $M_2$ into $H_i$, since they stop moving at level $i$ (Line 7). We then update the receptor heaps when a heap contains the restricted distance of point $q$, but $q$ has a smaller distance to a newly inserted $p \in M_2$ than to its previous closest point (Lines 8–10).

Using our batch-parallel binary heap, which we will describe in detail in Section 8.4, each batch update of the heap takes $O(\log(n+m))$ depth. The computation of the new restricted distances takes $O(1)$ depth. Therefore, the naive heap update algorithm takes $O(\log(n + m))$ depth per call. For the batch insertion algorithm in Algorithm 21, GridInsert on level $i + 1$ is blocked by HeapUpdate-Naive

of level $i$, to prevent multiple initiators updating the same receptor simultaneously. Figure 35a illustrates four levels of the data structure. The updates of level 2 shown in blue dashed lines are blocked until the completion of level 1 shown in red solid lines, and similarly for the remaining levels. Since there are $O(\log(n+m))$ levels *whp*, this leads to a overall depth of $O(\log^2(n+m))$ *whp* for the heap updates. A similar argument applies for the batch deletion algorithm, except that the order that the levels are updated proceeds in descending value of $i$ starting with $L$, as shown in Figure 35c.

We improve the depth of the heap update to $O(\log(n+m))$ *whp* by pipelining the heap updates. For insertion, a crucial observation is that the grid update of level $i+1$ can start right after that of $i$. Meanwhile, we run the heap updates in lock-step in parallel with the grid updates, as illustrated in Figure 35b—for each heap, the update from level $i+1$ can start right after that of level $i$ is completed. For example, the update on $H_2$ initiated by level 2 (blue dashed arrow with a value of 4) can start right after the update initiated by level 1 (red arrow with a value of 3). Like insertion, deletion follows similar strategy in the reverse order, as illustrated in Figure 35d.

**Analysis.** All updates on the heaps in our data structure are a result of points that start or stop moving at some level. First, we are concerned with those added to or deleted from $S'_i$, and hence $H_i$. Since the $S'_i$ for $1 \leq i \leq L$ are disjoint sets, $O(m)$ points from $Q$ are inserted or deleted from $H_i$ across all $1 \leq i \leq L$. Second, points in $down_i$ and $up_i$ for $1 \leq i \leq L$ also cause heap updates, and the total number of heap updates from these points is $O(m)$ by Lemmas 4 and 7. Therefore, across all levels, there are $O(m)$ updates to the heap. For all $p$ that start or stop moving across all levels, finding and recomputing $d^*_{i+l}(q)$ for $0 \leq l \leq k$ takes $O(m)$ work and $O(1)$ depth. This is because the cost associated with each $q$ whose $d^*(q)$ depends on $p$ involves searching the constant number of boxes surrounding $q$ and $p$ in $k$ sparse sets, where $k$ is constant, as shown by Golin et al. [111]. In section 8.4, we will show that batch insertions and deletions on a heap take $O(\log(n+m))$ depth. Since the heap updates are performed in parallel with the grid updates, they are not on the critical path of the computation. The total depth is dominated by the grid updates, which is $O(\log(n+m)\log^*(n+m))$ *whp*.

This leads to our overall bounds of amortized $O(m(1+\log((n+m)/m)))$ work and $O(\log(n+m)\log^*(n+m))$ depth *whp* for an update of size $m$, where the amortization comes from resizing the parallel dictionaries.

**Theorem 15.** *Updating our data structure for a batch of $m$ insertions/deletions takes amortized $O(m(1+\log((n+m)/m)))$ expected work and $O(\log(n+m)\log^*(n+m))$*

*depth* whp

Obtaining the closest pair from our data structure takes $O(1)$ work and depth. We simply call find-min on $H_i$ for $L-k \leq i \leq L$, and then take the overall minimum.

### Parallel Batch-Dynamic Binary Heap

One of the key components in parallelizing our closest pair algorithm is a parallel binary heap that supports batch updates (inserts and deletes) and find-min efficiently. This heap allows us to perform the parallel construction in linear work and perform updates with low depth. Our data structure may be of independent interest, since to the best of our knowledge, the only existing work on parallelizing a binary heap is on individual inserts or deletes [190].

A binary heap is a complete binary tree, where each node contains a key that is smaller than or equal to the keys of its children. Sequentially, the construction of a binary heap takes linear work, and each insert and delete takes $O(\log n)$ work [79]. The heap is represented as an array, and uses relative positions within the array to represent child-parent relationships. Sequentially, each insertion adds a new node at the end of the heap and runs Up-Heap to propagate the node up to the correct position in the heap. A deletion first swaps the node to delete with the node to the end of the heap, reduces the heap size by one, and then runs Up-Heap followed by Down-Heap (to propagate a node down to its correct position) for the node swapped to the middle of the heap.

Our parallel heap that supports batch updates (inserts and deletes) and finding the minimum element (find-min) efficiently is crucial to our data structure for the closest pair. Although we could implement a parallel heap using a parallel binary search tree, which supports a batch of $m$ updates to a set of $n$ elements in $O(m \log(n+m))$ work and $O(\log(n+m))$ depth [79], it supports more functionality (i.e., returning the minimum $K$ elements) than we need. In fact, the $O(m \log(n+m))$ work bound is tight for a binary search tree, since we can use it for comparison sorting.

Our batch-dynamic heap only needs to support the find-min operation rather than maintaining the full ordering, and hence it has a better work bound of $O(m(1+\log((n+m)/m)))$. Furthermore, it allows us to construct the initial heap in linear work (by setting $m$ to the number of points and $n = 0$ in the work bound), as needed for Theorem 14. The pseudocode of our algorithm is shown in algorithm 24, and its discussion is in Section 8.4. We present the analysis in this section.

Central to our parallel batch-dynamic binary heap is a new parallel Heapify algorithm, that takes $m$ updates from a valid heap of $n$ elements, and returns another valid heap (the pseudocode can be found in algorithm 24). It runs in two phases:

**Algorithm 24** Parallel HEAPIFY Algorithm

**Require:** A binary min-heap of size $n$ with $m$ updates, each of which is a triple $(v_i, k_i, k_i')$, indicating to update key $k_i$ to $k_i'$ on node $v_i$.

**Ensure:** An updated binary heap.

```
1: procedure HEAPIFY
2:   Let S^+ be the set of nodes with keys to be increased.
3:   Use integer sort to group the nodes in S^+ to S_l^+ by the level l in the heap (the
     root has level 0).
4:   for l ← ⌊log_2 n⌋ − 1 to 0 do
5:     for each v_i ∈ S_l^+ in parallel do
6:       DOWN-HEAP(v_i)
7:   Let S^− be the set of nodes with keys to be decreased.
8:   Use integer sort to group the nodes in S^− to S_l^− by the level l in the heap.
9:   for l ← 1 to ⌊log_2 n⌋ do
10:    for each v_i ∈ S_l^− in parallel do
11:      UP-HEAP(v_i)
```

the first phase works on increase-key updates, and the second phase on decrease-key updates. In both phases, we first use parallel integer sorting [197, 234] to categorize all updates based on the level where the update belongs. Simply running the UP-HEAP and DOWN-HEAP calls for the different updates in parallel does not achieve work-efficiency and low depth, and also leads to potential data races. Therefore, we *pipeline* each level of the UP-HEAP and DOWN-HEAP procedure. Specifically, in the first phase, once the first swap for the DOWN-HEAP in level $i$ is finished, we can immediately start the DOWN-HEAP on level $i-1$, instead of waiting for the DOWN-HEAP in level $i$ to completely finish (the root is at level 0, and level numbers increase going down). The swaps in the DOWN-HEAP calls from level $i-1$ will never catch up with the swaps from level $i$. Pipelining the second phase with UP-HEAP is more complicated. Our parallel UP-HEAP is run in a level-synchronous manner from the top level down to the bottom level. For each node on each level, both of its children may want to swap with the parent for having a larger value. In the parallel algorithm, we only make the child with the smaller value swap with the parent and continue its update to the upper levels, while the update for the other child terminates. We prove in section 8.4 that our parallel HEAPIFY algorithm takes $O(m(1+\log(n/m)))$ work and $O(\log n)$ depth.

We now explain how to perform batch insertions and deletions. A batch of $m$ insertions to a binary heap of size $n$ can be implemented using decrease-keys. We first

add the $m$ elements to end of the heap with keys of $\infty$. Then, we decrease the keys of these $m$ elements to their true values and run the parallel HEAPIFY algorithm. A batch of $m$ deletions can be processed similarly, but the deletions will generate "holes" in the tree structure, and so we need an additional step to fill these holes first. We pack the last $m$ elements in the heap based on whether they are deleted. Then, we use them to fill the rest of the empty slots by deletions, and run the parallel HEAPIFY algorithm. Hence, batch insertions and deletions take $O(m(1 + \log((n + m)/m)))$ work and $O(\log(n + m))$ depth.

**Analysis of the HEAPIFY Algorithm** **Correctness.** The correctness can be shown inductively on subtrees of increasing height. For the base case, all leaf nodes are valid binary heap subtrees, each containing one node. Then on the first iteration, we run DOWN-HEAP for updated keys on the second to last level. If the increased keys violate the heap property, then DOWN-HEAP will heapify this subtree, which has two levels. Similarly, for each node $v$ with increased keys on level $i$, both of $v$'s childrens' subtrees are valid binary heap subtrees, and so after DOWN-HEAP, the subtree rooted at $v$ is a valid binary heap subtree. The correctness for UP-HEAP can be shown symmetrically. The main difference is that in UP-HEAP, the update paths can overlap, but the correctness is guaranteed since it is implemented in a round-synchronous manner. In addition, when both children are updating a parent, we only allow the smaller child to continue its update path, while terminating that of the larger child, thereby satisfying the heap property.

**Work.** We now consider the work of this algorithm. Let $h$ be the height of the binary heap. For the worst case analysis, we always assume that DOWN-HEAP pushes a node to the leaf and that UP-HEAP pushes a node to the root. The case for DOWN-HEAP is simple—for $m = 2^r - 1$ increase-keys, the worst case is when they are in the top $r$ levels. Each DOWN-HEAP is independent and the total work is

$$\sum_{i=0}^{r} 2^i(h - i) = O(m(h - r + 1)) = O\Big(m\Big(1 + \log\Big(\frac{n}{m}\Big)\Big)\Big).$$

The work for UP-HEAP is more involved. Let $m_i$ be the number of increase-keys on level $i$. We know that $m_i \le 2^i$ and $\sum m_i \le m$. For level $i$, the work for all calls to UP-HEAP is upper bounded by the number of nodes on the path from the root to all updated nodes in level $i$. It can be shown that the number of such nodes is $O\Big(m_i\Big(1 + \log \frac{n}{2^{h-i}m_i}\Big)\Big)$ (Theorem 6 in [42]). Hence, the overall work for all levels is $W = O\Big(\sum_{i=0}^{\log_2 n} m_i\Big(1 + \log \frac{n}{2^{h-i}m_i}\Big)\Big)$. Let $m' = \sum_i m_i$, and we know that $m' \le m$.

To bound the work, we consider the maximum value of $W$ for any given $m'$. We can use the method of Lagrange multipliers, and compute the partial derivative of $m_i$ (without the big-$O$), which solves to

$$\frac{\partial}{\partial m_i} W = \frac{\partial}{\partial m_i}\left(m_i\left(1 - \log_2 \frac{2^{h-i} m_i}{n}\right)\right) = \log_2 \frac{n}{2^{h-i} m_i} - \frac{1}{\ln 2} + 1.$$

Since the constraint for $\sum m_i$ is linear, $W$ is maximized when $\frac{\partial}{\partial m_i} W = \frac{\partial}{\partial m_j} W$ for all levels $0 \le i, j \le h$, which solves to $m_i = cm'/2^{h-i+1}$ for some value of $c$ such that $m' = \sum m_i$. Note that $1 < c < 2$. Plugging this in gives

$$\begin{aligned} W &= O\left(\sum_{i=0}^{\log_2 n} \frac{m'}{2^{h-i+1}}\left(1 + \log \frac{n}{2^{h-i}(cm'/2^{h-i+1})}\right)\right) \\ &= O\left(\left(\sum_{i=0}^{\log_2 n} \frac{m'}{2^{h-i+1}}\right)\left(1 + \log\left(\frac{n}{m'}\right)\right)\right) = O\left(m\left(1 + \log\left(\frac{n}{m}\right)\right)\right). \end{aligned}$$

In addition to Down-Heap and Up-Heap, we also need to integer sort the updates on Lines 3 and 8, which takes $O(m)$ work. Hence, the total work for algorithm 24 is $O\big(m\big(1 + \log\big(\frac{n}{m}\big)\big)\big)$.

**Depth.** The integer sort on Lines 3 and 8 takes $O(\log m)$ depth. Directly running algorithm 24 gives $O(\log^2 n)$ depth—there are $O(\log n)$ tree levels, and on each level, Up-Heap or Down-Heap requires $O(\log n)$ depth. We can improve the depth bound to $O(\log n)$ using pipelining, as discussed in section 8.4. The Up-Heap and Down-Heap calls at all levels will have begun by the $i$'th round, and each call takes $O(\log n)$ rounds to finish. Each round takes $O(1)$ depth, and so the overall depth is $O(\log n)$. The pipelining does not increase the work.

**Theorem 16.** *For a batch-parallel binary heap of size $n$ and a batch update (a mix of inserts, deletes, and increase/decrease-keys) of size $m$, updating the heap takes $O(m(1 + \log((n+m)/m)))$ work and $O(\log(n+m))$ depth, and find-min takes $O(1)$ work.*

### Implementations

**Simplified Data Structure.** While the sparse partition maintains $(S_i, S'_i, p_i, q_i, d_i)$ and $H_i$ for each level $1 \le i \le L$, we found that implementing $S'_i$ and its associated

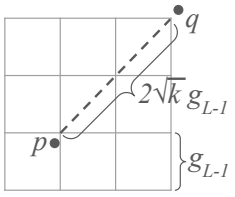



Figure 36: Illustration for the proof of Lemma 10: Let $\delta(S)$ be $p$ and $q$. Suppose by contradiction, $d(p,q) > 2\sqrt{k}g_{L-1}$, where $k = 2$ in this example. Then $p$ and $q$ will be sparse in level $L-1$, hence all the points will be sparse in level $L-1$.

heap $H_i$ on every level was inefficient in practice. We found it more efficient to only maintain $(S_i, p_i, q_i, d_i)$ for $1 \leq i \leq L$, and one heap $H^*$ that stores the closest neighbor distances for all $q$ in $S_j$, where $j = L - \lceil \log_3 2\sqrt{k} \rceil$. When $L$ changes due to insertion or deletion, we recompute $j$ and rebuild $H^*$ if necessary.

We now prove that $H^*$ contains the closest pair. Specifically, we prove that any point pair $(a,b)$ where $a, b \in S \setminus S_j, \forall j \leq L - \lceil \log_3 2\sqrt{k} \rceil$ cannot give rise to the closest pair distance, which we denote as $\delta(S)$.

**Lemma 10.** *$\delta(S) < d(a,b)$ for any $a, b \in S \setminus S_j, \forall j \leq L - \lceil \log_3 2\sqrt{k} \rceil$.*

*Proof.* The side length of the grid $G_i$ at level $i$ is $g_i = d_i/6k$, as defined earlier. Without loss of generality, consider level $j$, and let $a$ and $b$ be two points such that $a, b \in S_{j-1} \setminus S_j$ (this can easily be generalized to $a$ and $b$ being sparse on different levels that are both $< j$). We know $d(a,b) > g_{j-1} \geq 3g_j$ by properties of the sparse partition (Section 8.4). On the other hand, we also know that $\delta(S) \leq 2\sqrt{k}g_{L-1}$ , since otherwise, the pair that defines $\delta(S)$ would have been sparse in level $L-1$, and the last level $L$ would not have existed, which is a contradiction (see Figure 36). Given the property of the sparse partition, we have $3g_j \geq 2\sqrt{k}g_{L-1}$ for all $j \leq L - \lceil \log_3 2\sqrt{k} \rceil$. We can verify that this is true as $3 \cdot g_{L-\lceil \log_3 2\sqrt{k} \rceil} \geq 3 \cdot 3^{\lceil \log_3 2\sqrt{k} \rceil} \cdot g_L \geq 3^{\lceil \log_3 2\sqrt{k} \rceil} \cdot g_{L-1} \geq 2\sqrt{k} \cdot g_{L-1}$. Therefore, $d(a,b) > \delta(S)$. □

Since $a$ and $b$ satisfying Lemma 10 are both sparse in some $S_h$ where $h < L - \lceil \log_3 2\sqrt{k} \rceil$, it follows that $d(a,q) > d(q, S_{L-\lceil \log_3 2\sqrt{k} \rceil})$ and $d(b,q) > d(q, S_{L-\lceil \log_3 2\sqrt{k} \rceil})$ for any $q \in S_{L-\lceil \log_3 2\sqrt{k} \rceil}$. Therefore, the closest pair distance $\delta(S) = d(p,q)$ for some $p, q \in S_{L-\lceil \log_3 2\sqrt{k} \rceil}$, and will not involve $a$ or $b$.

Our implementation uses the parallel heap from [222]. Additionally, we compute $S'_i$ from $S_i$ on the fly when needed.

**Neighborhood Search.** Some of the work bounds are exponential in the dimensionality $k$, e.g., a grid's box neighborhood is of size $3^k$. For $k \geq 5$, the straightforward implementation is inefficient due to a large constant overhead in the work. Hence, we implement a parallel batch-dynamic $k$d-tree for $k \geq 5$. This is because performing a range query on the tree works better in practice, as it only needs to traverse the non-empty boxes in the neighborhood instead of all boxes. Our dynamic $k$d-tree is a standard spatial median $k$d-tree [38], augmented with the capability for parallel batch updates. Each internal node maintains metadata on the points in its subtree, which are partitioned by a spatial median along the widest dimension. The points are only stored at leaf nodes. We flatten a subtree to a single leaf node when it contains at most 16 points.

The tree supports batch insertion by first adding the batch to the root, and then traversing down multiple branches of the tree in parallel. At each internal node, we partition the inserted batch by the spatial median stored at the node, and modify its metadata, such as the point count and the coordinates of its bounding box. At each leaf node, we directly modify the metadata and store the points. The tree supports batch deletions by modifying the metadata, and marking the deleted points at the leaves as invalid. We manage the memory periodically to free up the invalid entries.

**Static Algorithms.** In addition to our batch-dynamic closest pair algorithm, we implement several sequential and parallel algorithms for the static closest pair problem. As far as we know, this thesis presents the first experimental study of parallel algorithms for static closest pair. We implement a parallel divide-and-conquer algorithm by Blelloch and Maggs [47], a simplified and parallel version of Rabin's algorithm [195] that we designed, a parallel version of Khuller and Matias's [148] sieve algorithm that we designed, and a parallel randomized incremental algorithm by Blelloch et al. [45]. We explain more details about these static algorithms and their implementations in section 8.4.

### Static Algorithms and Implementations

In addition to our batch-dynamic closest pair algorithm, we implement several parallel algorithms for the static closest pair problem, which we describe in this section. We evaluate all of them against each other, and compare them to our parallel batch-dynamic algorithm in Section 8.4. As far as we know, this thesis presents the first experimental study of parallel algorithms for the static closest pair problem.

**Divide-and-Conquer Algorithm** The first divide-and-conquer algorithm for closest-pair was introduced by Bentley [37], and has $O(n \log n)$ work and is optimal in the

algebraic decision tree model. Blelloch and Maggs [47] parallelize this algorithm, and their algorithm takes $O(n \log n)$ work and $O(\log^2 n)$ depth. Atallah and Goodrich [27] present another parallel algorithm based on multi-way divide-and-conquer, which takes $O(n \log n \log \log n)$ work and $O(\log n \log \log n)$ depth.

We implement the divide-and-conquer algorithm by Blelloch and Maggs [47]. The main idea of the algorithm is to divide the space containing all the points $S$ along an axis-aligned hyperplane by the median point along a dimension fixed throughout the algorithm, to form left and right subproblems. We then recursively find the closest pair in each of the two subproblems in parallel to obtain results $\delta_L$ and $\delta_R$. Then, we merge the two subproblems, and consider the points near the median point, which are the points within a distance of $\min\{\delta_L, \delta_R\}$ from the median point. We call the set of such points a ***central slab***, and use an efficient "boundary merging" technique to obtain $\delta_M$. The closest pair will have distance $\delta(S) = \min\{\delta_L, \delta_R, \delta_M\}$. Finding the median and performing the merge can be done using standard parallel primitives.

Blelloch and Maggs' [47] parallel algorithm requires the central slab to be sorted in a dimension $d$ different from the dimension that is used to divide the problem. Their algorithm orders the points along $d$ by performing recursive partitioning and merging at each level of the divide-and-conquer algorithm. Since the central slab can be linear in size, the merging algorithm is efficient in theory. However, we find that the central slab is very small for inputs that arise in practice. Therefore, in our algorithm, we simply use comparison sorting to sort the central slab when needed without using partitioning and merging, which results in better performance in practice. We also coarsen the base case, and switch to a quadratic-work brute-force algorithm when the subproblem size is sufficiently small.

**Rabin's Algorithm** Rabin's algorithm [195] is the first randomized sequential algorithm for the problem. Assuming a unit-cost floor function, Rabin's algorithm has $O(n)$ expected work. MacKenzie and Stout [164] design a parallel algorithm based on Rabin's algorithm, and achieve $O(n)$ work and $O(1)$ depth in expectation. Specifically, the algorithm first takes a random sample of $n^{0.9}$ points, and finds the closest pair on this sample recursively. Then it forms a grid structure with side length equal to the closest pair distance. Each box along with its points are classified into sparse or dense based its point count. The pairwise distances of the dense points (fewer than $n^{0.4}$) are computed using a quadratic work algorithm, after which the minimum is taken. Then, all points find their closest sparse points by checking neighboring sparse boxes in parallel, after which the minimum of these distances are taken. The algorithm uses some parallel primitives with high constant factor overheads, and are unlikely to be practical.

We design a simpler parallel version of Rabin's algorithm. Our algorithm takes a sample of $n^c$ points where $c < 1$, and recursively computes the closest distance $\delta'$ of the sample. Then, we construct a grid structure on all of the points $S$ using a parallel dictionary, where the box size is set to $\delta'$. For each point $x \in S$, we find its closest point by exploring $N(x, S)$, and then take the minimum among that of all $x$ to obtain $\delta(S)$. In terms of work, MacKenzie and Stout [164] showed by recursively finding the closest pair on a sample of size $n^c$, the total work is $O(n)$ in expectation. We find $c = 0.8$ to work well in practice. In terms of depth, our implementation has $O(\log n)$ levels of recursion, each taking $O(\log^* n)$ depth *whp*, which includes parallel dictionary operations and finding the minimum in parallel. The total depth is $O(\log n \log^* n)$ *whp*. In the recursion, we coarsen the base case by switching to a brute-force algorithm when the problem is sufficiently small.

**Sieve Algorithm** Khuller and Matias [148] propose a simple sequential algorithm called the sieve algorithm that takes $O(n)$ expected work (the dynamic algorithm by Golin et al. [111] is based on the sieve algorithm). The algorithm proceeds in rounds, where in round $i$, it chooses a random point $x$ from the point set $S_i$ (where $S_1 = S$) and computes $d_i(x)$, the distance to its closest neighbor. Then, the algorithm constructs a grid structure on $S_i$, where each box has a side length of $d_i(x)$. It then moves the points that are sparse in $S_i$ into a new set $S_{i+1}$, and proceeds to the next round, until $S_{i+1}$ is empty. Finally, the algorithm constructs a grid structure on $S$ with boxes of size equal to the smallest box computed during the algorithm. For each point $x \in S$, we compute its closest neighbor using the grid by traversing its own box and the boxes bordering on it. Finally, we take the minimum among distances obtained by all the points to obtain $\delta(S)$.

The sequential algorithm takes $O(n)$ expected work as the number of points decreases geometrically from one level to the next. We obtain a parallel sieve algorithm by using our parallel construction for the sparse partition in Algorithm 20, but without the heap. Our parallel sieve algorithm takes $O(n)$ expected work and $O(\log n \log^* n)$ depth *whp*

**Incremental Algorithm** Golin et al. [110] present a sequential incremental algorithm for closest pair with $O(n)$ expected work. Blelloch et al. [45] present a parallel version of this incremental algorithm, which we implement. The parallel algorithm works by maintaining a grid using a dictionary, and inserting the points in a randomized order in batches of exponentially increasing size. The side length of the grid box is the current closest pair distance, which is initialized to the distance between the first two points in the randomized ordering. For the $i$'th point inserted, the

algorithm will check its neighborhood for a neighbor with distance smaller than the current grid side length. When such a neighbor is found, the algorithm rebuilds the grid for the first $i$ points using the new side length, and continues with the insertion. Since the parallel algorithm inserts points in batches, for each batch we find the earliest point $i$ remaining in the batch that causes a grid rebuild, perform the rebuild on all points up to and including $i$, remove these points from the batch, and repeat until the batch is empty. After all batches are processed, the pair whose distance gives rise to the final grid side length is the closest pair. The algorithm takes $O(n)$ expected work and $O(\log n \log^* n)$ depth *whp*

### Experiments

**Algorithms Evaluated.** We evaluate our parallel batch-dynamic algorithm by benchmarking its performance on batch insertions (***dynamic-insert***) and batch deletions (***dynamic-delete***). We also evaluate the four static implementations described in Section 8.4, which we refer to as ***divide-conquer***, ***rabin***, ***sieve***, and ***incremental***. In addition, we implement and evaluate sequential versions of all of our algorithms that do not have the overheads of parallelism. Our implementations use the Euclidean metric ($L_2$-metric).

**Data Sets.** We use the synthetic seed spreader (SS) data sets produced by the generator in [104]. It produces points generated by a random walk in a local neighborhood, but jumping to a random location with some probability. ***SS-varden*** refers to the data sets with variable-density clusters. We also use a synthetic data set called ***Uniform***, in which points are distributed uniformly at random inside a bounding hyper-cube with side length $\sqrt{n}$, where $n$ is the total number of points. The points have double-precision floating-point values. We generated the synthetic data sets with 10 million points for dimensions $k = 2, 3, 5, 7$. We name the data sets in the format of ***Dimension-Name-Size***. We also use the following real-world data sets: ***7D-Household-2M*** [92] is a 7-dimensional data set containing household sensor data with $2,049,280$ points excluding the date-time information; ***16D-Chem-4M*** [97, 2] is a 16-dimensional data set with $4,208,261$ points containing chemical sensor data; and ***3D-Cosmo-298M*** [153] is a 3-dimensional astronomy data set with $298,246,465$ points.

**Testing Environment.** Our experiments are run on an `r5.24xlarge` instance on Amazon EC2. The machine has 2 × Intel Xeon Platinum 8259CL CPU (2.50 GHz) CPUs for a total of 48 cores with two-way hyper-threading, and 768 GB of RAM. By default, we use all cores with hyper-threading. We use the `g++` compiler (version

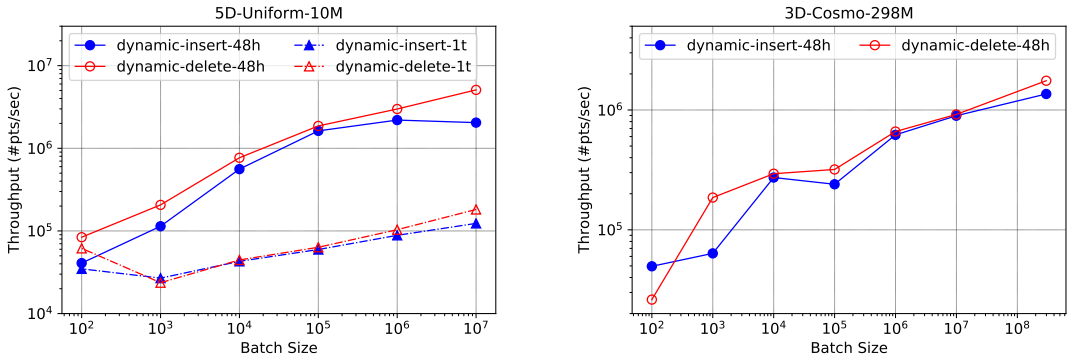



Figure 37: Plots of throughput vs. batch size in log-log scale for our parallel batch-dynamic algorithm on 5D-Uniform-10M and 3D-Cosmo-298M. The algorithm on 48-cores with hyper-threading and 1 thread has a suffix of "48h" and "1t", respectively. For 3D-Cosmo-298M, we omit the 1-thread times as the experiments exceeded our time limit.

7.5) with the `-O3` flag, and use Cilk Plus for parallelism [136]. We use the ***-48h*** and ***-1t*** suffixes in our algorithm names to denote the 48-core with hyper-threading and single-threaded times, respectively. We allocate a maximum of 2 hours for each test, and do not report times for tests that exceed this limit.

**Influence of Batch Size on Throughput.** In this experiment, we evaluate our batch-dynamic algorithm by measuring their throughput as a function of the batch size. For insertions, we insert batches of the same size until the entire data set is inserted. For deletions, we start with the entire data set and delete batches of the same size until the entire data set is deleted. We compute throughput by the number of points processed per second. We vary the batch size from 100 points to the size of the entire data set. Our parallel batch-dynamic algorithm achieves a throughput of up to $1.35\times10^7$ points per second for insertion, and up to $1.06\times10^7$ for deletion, under the largest batch size. On average, it achieves $1.75 \times 10^6$ for insertion and $1.94 \times 10^6$ for deletion across all batch sizes. We show plots of throughput vs. batch size for 5D-Uniform-10M and 3D-Cosmo-298M in Figure 37. We see that the throughput increases with larger batch sizes because of a lower relative overhead of traversing the sparse partition data structure, and the availability of more parallelism.

**Efficiency of Batch Insertions.** We evaluate the performance of dynamic batch insertion vs. using a static algorithm to recompute the closest pair. Specifically, we simulate a scenario where given the data structure storing the closest pair among $c$ data points, we perform an insertion of $b$ additional points. We compare the time

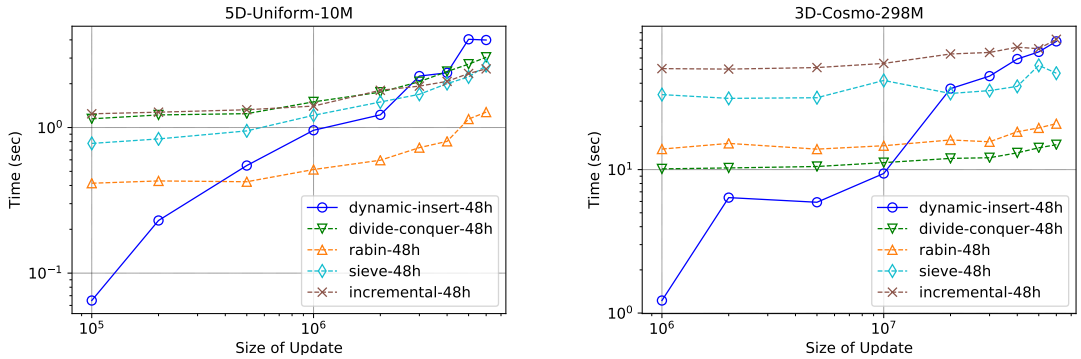



Figure 38: Plots of running time (in seconds) vs. insertion batch size for the dynamic and static methods using 48 cores with hyper-threading on 5D-Uniform-10M and 3D-Cosmo-298M. The plots are in log-log scale.

taken by the dynamic algorithm to process one batch insertion of size $b$, vs. that of a static algorithm for recomputing the closest pair for all $c + b$ points. We set $c$ to contain 40% of the data set and vary $b$. Figure 38 shows the running time as a function of $b$ for 5D-Uniform-10M and 3D-Cosmo-298M. For 5D-Uniform-10M, we see that our batch-dynamic algorithm outperforms the fastest among the static algorithms when the insertion batch size is smaller than 500,000. For 3D-Cosmo-298M, we see that the dynamic method outperforms the fastest static algorithm when the insertion batch is smaller than 10 million. In general, both the static and dynamic algorithms require more time to process the updates when the batch size is larger. The dynamic algorithm is much more advantageous for small to moderate batch sizes.

**Efficiency of Batch Deletions.** We evaluate the performance of dynamic batch deletion vs. using a static algorithm to recompute the closest pair. In this experiment, we are given the closest pair of all $n$ points in the data set, and perform a deletion of $b$ points. We compare the time taken for the dynamic algorithm to process one batch deletion of size $b$, vs. that of a static algorithm for recomputing the closest pair for the $n - b$ remaining points. Figure 39 shows the running time vs. deletion batch size for 5D-Uniform-10M and 3D-Cosmo-298M. For 5D-Uniform-10M, the dynamic algorithm outperforms the fastest static algorithm when the batch size is less than 3 million. For 3D-Cosmo-298M, the dynamic algorithm outperforms the static algorithm when the batch size is less than 60 million. In general, our dynamic algorithm requires more time to process the update when the batch size is larger, while the converse is true for the static algorithms.

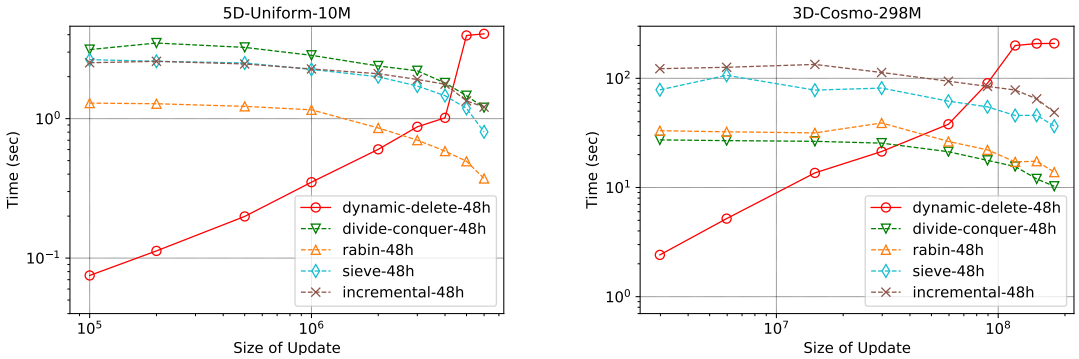



Figure 39: Plots of running time (in seconds) vs. deletion batch size for the dynamic and static methods using 48 cores with hyper-threading on 5D-Uniform-10M and 3D-Cosmo-298M. The plots are in log-log scale.

Compared to the fastest static algorithms on our data sets, we find that it is faster to use our dynamic algorithm for batch sizes of up to 20% of the data set.

**Static Methods.** We evaluate and compare the static algorithms and present all detailed running times in Table 10. Among the four parallel static algorithms, Rabin's algorithm is on average 7.63x faster than the rest of the algorithms across all data sets. The divide-and-conquer, sieve, and the incremental algorithms are on average 17.86x, 2.29x, and 2.73x slower than Rabin's algorithm, respectively. The divide-and-conquer algorithm actually achieves the fastest parallel running time on 7 out of the 11 data sets. However, it is significantly slower for most of the higher dimensional data sets, due to its higher complexity with increased dimensionality. The sieve algorithm and the incremental algorithm, though doing the same amount of work in theory as Rabin's algorithm, have higher constant factor overheads.

**Parallel Speedup and Work-Efficiency.** We measure the parallel speedups of our implementations by dividing the 1-thread time by the 48-core with hyper-threading time. Our parallel batch-dynamic algorithm achieves up to 38.57x self-relative speedup (15.10x on average across all batch sizes), averaging over both insertions and deletions. Our static implementations achieve up to 51.45x speedup (29.42x on average). Specifically, the divide-and-conquer algorithm, Rabin's algorithm, the sieve algorithm, and the incremental algorithm achieve average self-relative speedups of 35.17x, 33.84x, 29.22x, and 19.45x, respectively; in addition, they achieve an average speedup of 19.10x, 23.56x, 10.56x, and 9.23x, respectively, over the fastest serial algorithm for each data set.

Our parallel implementations when run on one thread demonstrate modest over-

| | Divide-Conquer | | | Rabin | | | Sieve | | | Incremental | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Seq | 1t | 48h | Seq | 1t | 48h | Seq | 1t | 48h | Seq | 1t | 48h |
| 2D-Uniform-10M | 9.54 | 9.62 | 0.24 | 11.2 | 11.6 | 0.28 | 23.3 | 24.5 | 0.81 | 22.1 | 17.7 | 1.02 |
| 3D-Uniform-10M | 24.9 | 25.2 | 0.66 | 28.4 | 30.5 | 0.78 | 60.3 | 60.6 | 1.82 | 50.5 | 46.2 | 2.50 |
| 5D-Uniform-10M | 101 | 136 | 3.04 | 25.3 | 28.4 | 1.28 | 56.7 | 60.6 | 2.63 | 49.2 | 50.3 | 2.40 |
| 7D-Uniform-10M | 561 | 618 | 14.7 | 81.7 | 82.8 | 1.70 | 124 | 135 | 4.24 | 93.7 | 106 | 4.58 |
| 2D-SS-varden-10M | 7.58 | 8.95 | 0.23 | 10.5 | 11.2 | 0.26 | 22.2 | 22.8 | 0.94 | 23.4 | 17.5 | 1.11 |
| 3D-SS-varden-10M | 17.3 | 19.1 | 0.51 | 28.4 | 29.1 | 0.77 | 58.4 | 58.3 | 1.68 | 48.7 | 43.1 | 1.97 |
| 5D-SS-varden-10M | 24.9 | 33.4 | 0.82 | 22.6 | 26.1 | 1.43 | 47.2 | 49.3 | 2.58 | 40.4 | 41.7 | 2.44 |
| 7D-SS-varden-10M | 43.1 | 50.3 | 1.33 | 33.1 | 34.0 | 1.61 | 64.4 | 70.9 | 3.00 | 43.4 | 48.0 | 2.53 |
| 7D-Household-2M | 342 | 392 | 13.4 | 7.23 | 7.70 | 0.40 | 15.9 | 18.1 | 0.73 | 13.8 | 15.7 | 0.94 |
| 16D-Chem-4M | 315 | 499 | 202 | 38.3 | 39.8 | 1.38 | 88.2 | 96.7 | 2.68 | 59.1 | 70.8 | 3.91 |
| 3D-Cosmo-298M | 750 | 747 | 20.7 | 1243 | 1625 | 31.6 | 3383 | 2819 | 70.6 | 3456 | 2629 | 104 |

Table 10: Running times (in seconds) of static algorithms. "Seq" denotes the sequential implementation. "1t" and "48h" denote the parallel implementation run on 1 thread and 48 cores with hyper-threading, respectively.

heads over their sequential counterparts. Our parallel batch-dynamic algorithm running on 1 thread has only 1.13x lower throughput on average over our sequential implementation of the algorithm. For the static algorithms, the parallel divide-and-conquer, Rabin's, sieve, and incremental algorithms running on 1 thread are only 1.18x, 1.08x, 1.04x, and 1.00x slower on average, respectively, than their corresponding sequential algorithms.

# 9 ParGeo: A Library for Parallel Computational Geometry

## 9.1 Introduction

In this thesis, we present the ParGeo library for parallel computational geometry, which includes a rich set of parallel algorithms for geometric problems and data structures, including $k$d-trees, $k$-nearest neighbor search, range search, well-separated pair decomposition, Euclidean minimum spanning tree, spatial sorting, and geometric clustering. ParGeo also contains a collection of geometric graph generators, including $k$-nearest neighbor graphs and various spatial networks. Algorithms from ParGeo can either run sequentially, or run using parallel schedulers such as OpenMP or Cilk.

While there exist numerous libraries for computational geometry, most of them are not designed for parallel processing. For example, Libigl [137] is a library that specializes in the construction of discrete differential geometry operators and finite-element matrices. However, only some aspects of Libigl take advantage of parallelism. In comparison, the algorithms and implementations of ParGeo are designed for parallelism, and target a different set of problems. CGAL (Computational Geometry Algorithms Library) [96] is a well known library of computational geometry algorithms that includes a wide range of packages, but most implementations are not parallel. Batista et al. [34] targeted a few important algorithms, including spatial sorting, box intersection, and Delaunay triangulation for shared-memory parallel processing, with code in CGAL. In comparison, ParGeo targets similar classes of problems as CGAL, but all of our implementations are highly parallel.

In addition to its other features, ParGeo includes a module for generating geometric graphs from spatial datasets. When combined with GeoGraph, a sister framework with a user-friendly interface, users can easily access powerful parallel geometric graph generators. Graphs are a powerful tool for representing relationships in data, with applications ranging from social network analysis to transportation planning. However, analyzing large graphs efficiently requires high-performance parallel programs, which can be challenging for non-experts in high-performance computing. Fortunately, there exist programming frameworks that provide highly-optimized parallel implementations of graph processing functions.

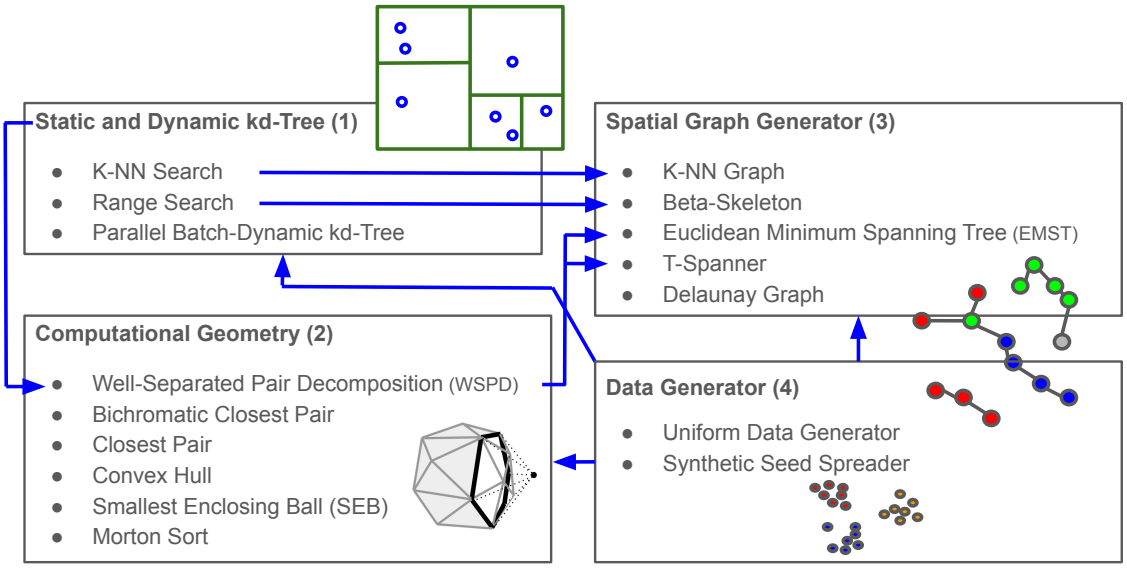



Figure 40: The figure shows an overview of modules in ParGeo. An arrow indicates that a component is used inside another component. In this thesis, we present new algorithms and techniques for the modules highlighted in green.

## 9.2 ParGeo Modules and Problems Studied

We present an overview of the modules of ParGeo in Figure 40, highlighting how the modules interact with each other.

ParGeo contains efficient multicore implementations of static and dynamic $k$d-trees and related algorithms (Module (1)). The code supports $k$d-tree based spatial search, including $k$-nearest neighbor and range search. Our code is optimized for fast $k$d-tree construction by performing the split in parallel (either by spatial median or by object median), and the queries themselves are data-parallel. ParGeo also includes a new cache-oblivious algorithm for $k$d-tree construction and a parallel-batch dynamic $k$d-tree using the logarithmic method described by Yesantharao et al. [255].

ParGeo contains a module for parallel computational geometry algorithms (Module (2)). Our $k$d-tree can be used to generate a well-separated pair decomposition [60] (WSPD), which can in turn be used to compute the hierarchical DBSCAN [244], ParGeo contains parallel implementations for the bichromatic closest pair, closest pair, convex hull, smallest enclosing ball, and Morton sorting.

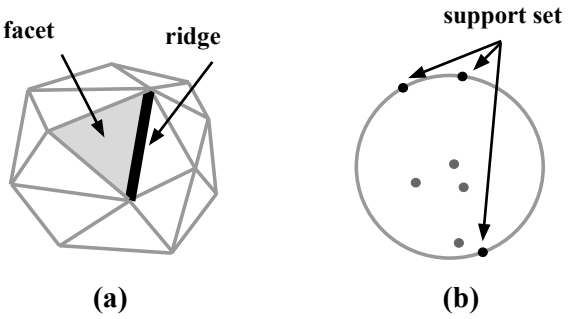



Figure 41: (a) A facet and a ridge of a convex hull in $\mathbb{R}^3$. (b) The support of smallest enclosing ball in $\mathbb{R}^2$.

In addition, ParGeo contains a collection of geometric graph generators (Module (3)) for point data sets. Our $k$d-tree's $k$-NN search is used to generate the $k$-NN graph, and the range search is used to generate the $\beta$-skeleton. Our WSPD generated from the $k$d-tree can also be used to compute the Euclidean minimum spanning tree [58, 244], and spanners [60]. ParGeo also generates the Delaunay graph.

In previous sections, we covered the algorithms and implementation for computational geometry. In this section, we will shift our focus to parallel graph generators and experimental evaluations.

## 9.3 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 [168, 99, 56, 160, 147], manifold learning [225], outlier detection [123], and proximity search [182, 67, 209]. 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 $k$d-tree, a binary tree data structure commonly used for $k$-NN queries [100]. A $k$d-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 $k$d-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 [32, 33]. 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 [84], 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 $\beta$-skeleton graphs [151], both of which have a variety of real-world applications [237, 247, 194, 196, 232, 18, 226, 142]. We use the parallel incremental Delaunay triangulation implementation from the Problem Based Benchmark Suite [215].

The ***$\beta$-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 $\beta$. The $\beta$-skeleton shares the same vertex set as the Delaunay graph, but only contains a subset of the Delaunay edges [232]. We use the $k$d-tree to construct the $\beta$-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 $\beta$. The check can be reduced to several range searches in a $k$d-tree. The $\beta$-skeleton generalizes other well known spatial network graphs, such as the Gabriel graph and the relative neighborhood graph [151, 142].

ParGeo contains a point data generator module (Module (4)) for which can generate uniformly distributed data sets, and clustered data sets of varying densities [104]. These data sets are used for benchmarking of the other modules.

## 9.4 Performance Evaluation

To demonstrate the efficiency of our proposed algorithms and library, we perform a comprehensive set of experiments on synthetic and real-world geometric data sets, and compare the performance across our parallel implementations as well as optimized sequential baselines. On 36 cores with two-way hyper-threading, our best convex hull implementation achieves up to 44.7x self-relative speedup and up to 559x

| Implementation | $\mathbf{T_1}$ | $\mathbf{T_{36h}}$ | Speedup |
|---|---|---|---|
| *kd-tree Build (2d)* | 5.51 | 0.43 | 12.70x |
| *kd-tree Build (5d)* | 8.39 | 0.89 | 9.40x |
| *kd-tree k-NN (2d)* | 31.45 | 0.68 | 46.34x |
| *kd-tree Range Search (2d)* | 17.14 | 0.37 | 46.61x |
| *Dynamic kd-tree Construction (5d)* | 6.70 | 0.60 | 10.70x |
| *Dynamic kd-tree Insert (5d)* | 8.80 | 1.10 | 8.10x |
| *Dynamic kd-tree Delete (5d)* | 29.20 | 1.20 | 23.90x |
| *WSPD (2d)* | 6.72 | 0.24 | 27.63x |
| *EMST (2d)* | 33.02 | 1.58 | 20.86x |
| *Convex Hull (2d)* | 0.38 | 0.0088 | 43.13x |
| *Convex Hull (3d)* | 2.36 | 0.097 | 24.36x |
| *Smallest Enclosing Ball (2d)* | 0.053 | 0.0033 | 16.30x |
| *Smallest Enclosing Ball (5d)* | 0.13 | 0.014 | 9.54x |
| *Closest Pair (2d)* | 10.35 | 0.52 | 19.90x |
| *Closest Pair (3d)* | 28.00 | 2.32 | 12.07x |
| *k-NN Graph (2d)* | 37.89 | 1.46 | 25.99x |
| *Delaunay Graph (2d)* | 55.91 | 2.03 | 27.53x |
| *Gabriel Graph (2d)* | 59.61 | 1.99 | 29.99x |
| *$\beta$-skeleton Graph (2d)* | 113.27 | 3.20 | 35.37x |
| *Spanner (2d)* | 27.19 | 2.15 | 12.67x |

Table 11: Runtimes (seconds) and parallel speedups ($T_1/T_{36h}$) for ParGeo implementations on uniform hypercube data sets of varying dimensions and 10 million points. $T_1$ and $T_{36h}$ denote the single-threaded and the 36-core hyper-threaded times, respectively. For dynamic $k$d-tree updates, each batch contains 10% of the data set.

speedup against the best existing sequential implementation for $\mathbb{R}^2$, and up to 24.9x self-relative speedup and up to 124x speedup against the best existing sequential implementation for $\mathbb{R}^3$. Our sampling-based smallest enclosing ball algorithm achieves up to 27.1x self-relative speedup and up to 178x speedup against the best existing sequential implementation for $\mathbb{R}^2$ and $\mathbb{R}^3$. Finally, across all implementations in ParGeo, we achieve self-relative parallel speedup of 8.1–46.61x (on average 23.15x). As shown in Table 11, on a machine with 36 cores with two-way hyper-threading, ParGeo achieves self-relative parallel speedups of 8.1–46.61x (on average 23.15x) on a uniformly distributed data set, across all the benchmarks.

## 9.5 An API for Graph Processing on Geometric Data

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 [187, 81, 28, 96], 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 [239, 244] and motion planning [84].

This thesis 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 [11] with graph algorithms and data formats from the Graph Based Benchmark Suite [88, 89]. 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 $\beta$-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

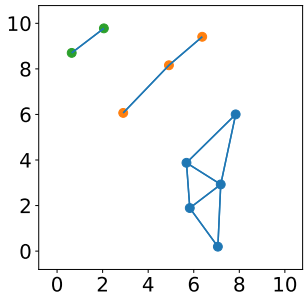



Figure 42: Example of running connected components on the 3-NN graph of a 2-dimensional point data set, where edges with weight greater than 3.2 are filtered out. The vertices of each color correspond to a connected component.

users would have to do with existing tools. We also compare with Higra [188], an existing library that supports graph algorithms and geometric graph construction using SciPy [233] and scikit-learn [187]. 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`.

### Examples of using the API

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 42.

We also consider applying hierarchical graph clustering algorithms to an input weighted graph. The output of these algorithms is usually a dendrogram representing

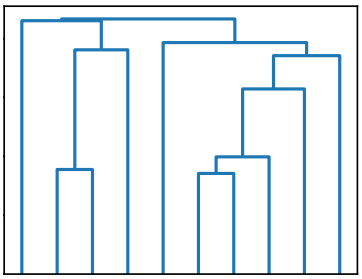


Figure 43: Example of running complete-linkage clustering on the 3-NN graph of a 2-dimensional point data set. We show the output, a corresponding dendrogram.

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 9.3. By using an efficient graph-based hierarchical clustering method, like a hierarchical version of the SCAN algorithm [228], or a graph-based agglomerative clustering algorithm, we can potentially significantly outperform classic approaches that only use the input point set [174, 147]. 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 43.

A Euclidean minimum spanning tree (EMST) on a point data set has various applications, including being used in single-linkage clustering [113], network placement optimization [237], and approximating the Euclidean traveling salesman problem [231]. A well-known fact is that the EMST is a subset of the Delaunay triangulation of a graph [142]. 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 44.

Finally, computing shortest paths on transportation and infrastructure networks is commonly used for planning [32]. These networks can be generated by constructing spatial networks on geometric data. We consider running the $\Delta$-stepping single-source shortest paths algorithm [172] on the $\beta$-skeleton of a point set. The following code shows running the $\Delta$-stepping algorithm with source vertex 0 and $\Delta = 0.01$ on

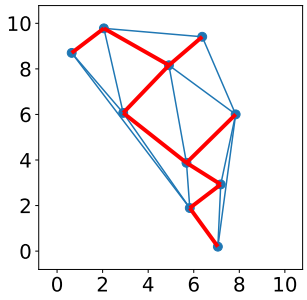



Figure 44: Example of running minimum spanning tree on the Delaunay triangulation graph of a 2-dimensional point data set. The edges of the minimum spanning tree are shown in red.

a $\beta$-skeleton graph with $\beta = 2$ (the relative neighborhood graph).

### Benchmarking

In this section, we benchmark the performance of GeoGraph on the four examples in section 9.5. Our implementations are all parallel, except for complete-linkage clustering, whose parallelization is a work in progress. We compare with Higra [188] (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 [233] and scikit-learn [187] 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 × 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 [187] generator, which produces samples from isotropic Gaussian 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 45, we show the running times of the methods on the four examples.

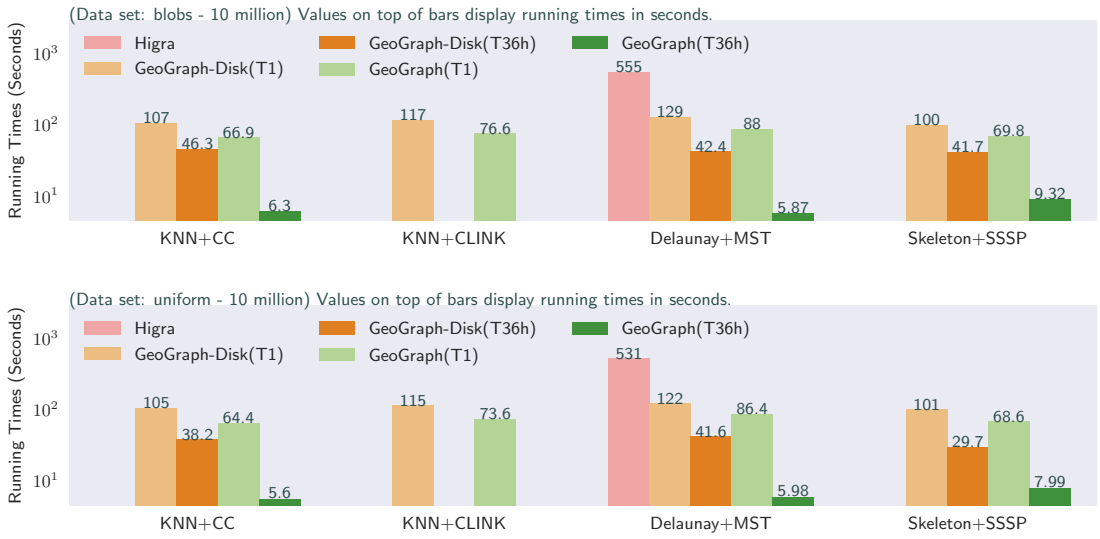



Figure 45: Comparison between GeoGraph, GeoGraph with disk I/O, and Higra. T1 corresponds to the serial time and T36h corresponds to the parallel time on 36 cores with hyper-threading. KNN+CC is connected components on the 3-NN graph; KNN+CLINK is complete-linkage clustering on the 3-NN graph; Delaunay+MST 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.

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.[8]

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

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

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.

# Part IV

# Conclusion and Future Work

# 10 Conclusion

This thesis has demonstrated the effectiveness of parallel shared-memory multi-core algorithms, implementations, and frameworks in efficiently processing large-scale spatial clustering and computational geometry problems both in theory and practice. We have designed and implemented a variety of parallel algorithms for spatial clustering, including exact and approximate DBSCAN as well as HDBSCAN* and EMST. Our experimental results showed that our proposed algorithms achieve significant speedup over existing sequential and parallel algorithms, enabling efficient processing of large-scale datasets.

In addition to spatial clustering, we have also contributed to several computational geometry problems, such as convex hull, smallest enclosing ball, and dynamic closest pair. These contributions further showcased the potential for parallel shared-memory multi-core algorithms to improve performance in various areas of computational geometry.

Through working on this thesis, we encountered challenges associated with bridging the gap between the theory and practice. In our proposed algorithms, oftentimes the algorithm derived from theory suffers from sub-optimal performance in practice. To address this, we proposed practical optimizations to improve these algorithms' speed and memory efficiency. While we found that theoretical analysis does not always capture all the complexities of a real machine, it pointed us in the right direction where we were able to derive efficient implementations.

In addition to algorithms, we have introduced the ParGeo library for parallel computational geometry. This library provides a comprehensive set of parallel algorithms for geometric problems and data structures, along with a collection of geometric graph generators. ParGeo serves as a valuable resource for researchers working with geometric data across various domains. We have also introduced a parallel grid clustering framework, which enables efficient implementation of various grid-based clustering algorithms.

In conclusion, this thesis has demonstrated that parallel shared-memory multi-core machines offer an ideal balance between programmability and efficiency for tackling large-scale spatial data processing challenges in both spatial clustering and computational geometry domains. The presented algorithms, implementations, and frameworks can significantly improve computational performance while maintaining theoretical efficiency. As the demand for faster processing speeds continues to grow

alongside increasing data set sizes, our work provides valuable insights into harnessing the power of parallel shared-memory multi-core machines to address these challenges effectively.

# 11 Future Work

This thesis has demonstrated the effectiveness of parallel shared-memory multi-core algorithms, implementations, and frameworks in efficiently processing large-scale spatial clustering and computational geometry problems. However, there remain several avenues for future work to further enhance the capabilities of these systems.

First, due to the diverse range of computing resources available today with varying 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. In particular, there are many opportunities in implementing the algorithms proposed in this thesis on general purpose GPUs.

Second, dynamic updates to input data sets (point insertions, deletions, or modifications) also presents opportunities for future work. Systems should be able to efficiently update associated data structures and algorithm outputs in parallel.

Third, we are interested in using our geometric graph processing framework to study algorithms and applications in geometry that can benefit from efficient graph algorithms internally. Examples include motion planning algorithms that require computing the shortest path on a visibility graph [84], and geometric clustering algorithms like DBSCAN [95, 239] and hierarchical spatial clustering [61, 244], which rely on underlying connected components or minimum spanning tree algorithms. Investigating the interaction between geometric data processing and graph processing within these applications highlights the need for a unified framework.

Fourth, designing efficient graph construction algorithms for high-dimensional datasets is an important research direction. Approximate $k$-NN graph construction methods have applications in data mining and information retrieval [71, 52, 221, 101, 169, 91]. Studying how different graph construction methods affect the quality of downstream tasks will provide valuable insights.

Finally, there is a challenge in developing efficient visualization techniques that present both the input point set and geometric graph realizations while illustrating algorithm outputs on both. Future systems should support parallel visualization techniques that scale to large data sets.

# Bibliography


[1] C++ implementation of the 3d quickhull algorithm. https://github.com/akuukka/quickhull.

[2] Chem dataset. https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynamic+gas+mixtures.

[3] Geolife dataset. https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/.

[4] Header only 3d quickhull in c99. https://github.com/karimnaaji/3d-quickhull.

[5] A header-only c implementation of the quickhull algorithm for building n-dimensional convex hulls and delaunay meshes. https://github.com/leomccormack/convhull_3d.

[6] Household dataset. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption.

[7] Ht dataset. https://archive.ics.uci.edu/ml/datasets/Gas+sensors+for+home+activity+monitoring.

[8] Qhull. http://www.qhull.org/.

[9] Quickhull3d: A robust 3d convex hull algorithm in java. https://www.cs.ubc.ca/~lloyd/java/quickhull3d.html.

[10] The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

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

[12] Pankaj K. Agarwal, Herbert Edelsbrunner, Otfried Schwarzkopf, and Emo Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. In *Annual Symposium on Computational Geometry*, pages 203–210, 1990.

[13] Pankaj K. Agarwal, Herbert Edelsbrunner, Otfried Schwarzkopf, and Emo Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry*, 6(3):407–422, September 1991.

[14] Pankaj K. Agarwal, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for closest pair and all nearest neighbors. *ACM Transactions on Algorithms (TALG)*, 5(1):1–37, 2008.

[15] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1):293–327, March 1988.

[16] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, page 94–105, New York, NY, USA, 1998. Association for Computing Machinery.

[17] Y. Akhremtsev and P. Sanders. Fast parallel operations on search trees. In *IEEE International Conference on High Performance Computing (HiPC)*, pages 291–300, 2016.

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

[19] Daichi Amagata and Takahiro Hara. Fast density-peaks clustering: multicore-based parallelization approach. In *Proceedings of the 2021 International Conference on Management of Data*, pages 49–61, 2021.

[20] Nancy M Amato and Franco P Preparata. The parallel 3d convex hull problem revisited. *International Journal of Computational Geometry & Applications*, 2(02):163–173, 1992.

[21] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-DBSCAN: A GPU accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369 – 378, 2013.

[22] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering points to identify the clustering structure. In *ACM International Conference on Management of Data (SIGMOD)*, pages 49–60, 1999.

[23] Antonio Cavalcante Araujo Neto, Ticiana Linhares Coelho da Silva, Victor Aguiar Evangelista de Farias, José Antonio F. Macêdo, and Javam de Castro Machado. G2P: A partitioning approach for processing DBSCAN with MapReduce. In *Web and Wireless Geographical Information Systems*, pages 191–202, 2015.

[24] Domenica Arlia and Massimo Coppola. Experiments in parallel clustering with dbscan. In *European Conference on Parallel Processing (Euro-Par)*, pages 326–331, 2001.

[25] Sunil Arya and David M. Mount. Approximate range searching. *Computational Geometry*, 17(3):135 – 152, 2000.

[26] Sunil Arya and David M. Mount. A fast and simple algorithm for computing approximate euclidean minimum spanning trees. In *ACM-SIAM Symposium on Discrete Algorithms*, page 1220–1233, 2016.

[27] Mikhail J. Atallah and Michael T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, 1986.

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

[29] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[30] Bahareh Banyassady and Wolfgang Mulzer. A simple analysis of Rabin's algorithm for finding closest pairs. *European Workshop on Computational Geometry (EuroCG)*, 2007.

[31] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

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

[33] Marc Barthelemy. January 2018.

[34] Vicente H.F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler. Parallel geometric algorithms for multi-core computers. *Computational Geometry*, 43(8):663–677, 2010.

[35] Bentley and Friedman. Fast algorithms for constructing minimal spanning trees in coordinate spaces. *IEEE Transactions on Computers*, C-27(2):97–105, February 1978.

[36] Jon L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[37] Jon L. Bentley and Michael I. Shamos. Divide-and-conquer in multidimensional space. In *ACM Symposium on Theory of Computing (STOC)*, pages 220–230, 1976.

[38] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[39] Sergei N. Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discrete & Computational Geometry*, 19(2):175–195, 1998.

[40] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[41] 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.

[42] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016.

[43] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *ACM SIGPLAN Symposium on Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 181–192, 2012.

[44] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low-depth cache oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 189–199, 2010.

[45] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 467–478, 2016.

[46] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Randomized incremental convex hull is highly parallel. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, page 103–115, 2020.

[47] Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. In *The Computer Science and Engineering Handbook*, pages 277–315. 1997.

[48] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[49] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. Density-based clustering using graphics processors. In *ACM Conference on Information and Knowledge Management*, pages 661–670, 2009.

[50] B. Borah and D. K. Bhattacharyya. An improved sampling-based DBSCAN for large spatial databases. In *International Conference on Intelligent Sensing and Information Processing*, pages 92–96, 2004.

[51] Prosenjit Bose, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Jan Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry*, 37(3):209 – 227, 2007.

[52] 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.

[53] S. Brecheisen, H. Kriegel, and M. Pfeifle. Efficient density-based clustering of complex objects. In *IEEE International Conference on Data Mining (ICDM)*, pages 43–50, 2004.

[54] Stefan Brecheisen, Hans-Peter Kriegel, and Martin Pfeifle. Parallel density-based clustering of complex objects. In *Advances in Knowledge Discovery and Data Mining (PAKDD)*, pages 179–188, 2006.

[55] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[56] 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.

[57] Paul B Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 332–340, 1993.

[58] Paul B. Callahan and S. Rao Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1993.

[59] Paul B. Callahan and S. Rao Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 263–272, 1995.

[60] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995.

[61] Ricardo Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Trans. Knowl. Discov. Data*, 10(1):5:1–5:51, July 2015.

[62] Jean Cardinal and David Eppstein. Lazy algorithms for dynamic closest pair with arbitary distance measures. In *Algorithm Engineering and Experiments (ALENEX)*, 2004.

[63] Timothy M. Chan. Geometric applications of a randomized optimization technique. *Discrete & Computational Geometry*, 22(4):547–567, 1999.

[64] Timothy M. Chan. Dynamic generalized closest pair: Revisiting Eppstein's technique. In *Symposium on Simplicity in Algorithms*, pages 33–37, 2020.

[65] Chung-I Chang, Nancy P Lin, Nien-Yi Jan, et al. An axis-shifted grid-clustering algorithm. *Journal of Applied Science and Engineering*, 12(2):183–192, 2009.

[66] Samidh Chatterjee, Michael Connor, and Piyush Kumar. Geometric minimum spanning trees with GeoFilterKruskal. In *International Symposium on Experimental Algorithms (SEA)*, volume 6049, pages 486–500, 2010.

[67] 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.

[68] Bernard Chazelle. How to search in history. *Information and control*, 64(1-3):77–99, 1985.

[69] Chun-Chieh Chen and Ming-Syan Chen. HiClus: Highly scalable density-based clustering with heterogeneous cloud. *Procedia Computer Science*, 53:149 – 157, 2015.

[70] Danny Z Chen, Michiel Smid, and Bin Xu. Geometric algorithms for density-based data clustering. *International Journal of Computational Geometry & Applications*, 15(03):239–260, 2005.

[71] 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.

[72] Xiaoming Chen, Wanquan Liu, Huining Qiu, and Jianhuang Lai. APSCAN: A parameter free algorithm for clustering. *Pattern Recognition Letters*, 32(7):973 – 986, 2011.

[73] Kenneth L Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, 17(4):830–847, 1988.

[74] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry, ii. *Discrete Comput. Geom*, 4:387–421, 1989.

[75] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, August 1988.

[76] Richard Cole, Philip N. Klein, and Robert E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 243–250, 1996.

[77] Massimo Coppola and Marco Vanneschi. High-performance data mining with skeleton-based structured parallel programming. *Parallel Comput.*, 28(5):793–813, May 2002.

[78] I. Cordova and T. Moh. DBSCAN on resilient distributed datasets. In *International Conference on High Performance Computing Simulation (HPCS)*, pages 531–540, 2015.

[79] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[80] CriteoLabs. Terabyte click logs, 2013.

[81] 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.

[82] N. Dadoun and D.G. Kirkpatrick. Parallel construction of subdivision hierarchies. *Journal of Computer and System Sciences*, 39(2):153–165, 1989.

[83] B. Dai and I. Lin. Efficient map/reduce-based DBSCAN algorithm with optimized data partition. In *IEEE International Conference on Cloud Computing*, pages 59–66, 2012.

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

[85] Mark de Berg, Ade Gunawan, and Marcel Roeloffzen. Faster DB-scan and HDB-scan in low-dimensional euclidean spaces. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 25:1–25:13, 2017.

[86] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[87] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 9(5):256–268, 1974.

[88] 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 (SPAA)*, pages 293–304, 2018.

[89] 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.

[90] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[91] 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.

[92] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[93] Y. El-Sonbaty, M. A. Ismail, and M. Farouk. An efficient density based clustering algorithm for large databases. In *IEEE International Conference on Tools with Artificial Intelligence*, pages 673–677, 2004.

[94] David Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Experimental Algorithmics*, 5:1–es, 2000.

[95] 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 *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–231, 1996.

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

[97] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.

[98] Steve Fortune and John Hopcroft. A note on rabin's nearest-neighbor algorithm. *Information Processing Letters*, 8(1):20–23, 1979.

[99] 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.

[100] 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, July 1976.

[101] 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.

[102] Xiufen Fu, Yaguang Wang, Yanna Ge, Peiwen Chen, and Shaohua Teng. Research and application of dbscan algorithm based on hadoop platform. In *Pervasive Computing and the Networked World*, pages 73–87, 2014.

[103] H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[104] Junhao Gan and Yufei Tao. On the hardness and approximation of euclidean DBSCAN. *ACM Trans. Database Syst.*, 42(3):14:1–14:45, 2017.

[105] Junhao Gan and Yufei Tao. Fast Euclidean OPTICS with bounded precision in low dimensional space. In *ACM SIGMOD International Conference on Management of Data*, pages 1067–1082, 2018.

[106] Mingcen Gao, Thanh-Tung Cao, Ashwin Nanjappa, Tiow-Seng Tan, and Zhiyong Huang. Ghull: A gpu algorithm for 3d convex hull. *ACM Trans. Math. Softw.*, 40(1), October 2013.

[107] B. Gärtner. Fast and robust smallest enclosing balls. In *ESA*, 1999.

[108] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, December 1991.

[109] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

[110] Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. Simple randomized algorithms for closest pair problems. *Nordic J. of Computing*, 2(1):3–27, March 1995.

[111] Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. Randomized data structures for the dynamic closest-pair problem. *SIAM J. on Computing*, 27(4):1036–1072, 1998.

[112] Markus Götz, Christian Bodenstein, and Morris Riedel. HPDBSCAN: Highly parallel DBSCAN. In *Workshop on Machine Learning in High-Performance Computing Environments*, pages 2:1–2:10, 2015.

[113] 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.

[114] Jonathan S. Greenfield. A proof for a quickhull algorithm. 1990.

[115] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

[116] Joachim Gudmundsson, Mikael Hammar, and Marc van Kreveld. Higher order delaunay triangulations. *Computational Geometry*, 23(1):85–98, 2002.

[117] Ade Gunawan. A faster algorithm for DBSCAN, 2013. Master's thesis, Eindhoven University of Technology.

[118] M. Haklay and P. Weber. OpenStreetMap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, October 2008.

[119] Shay Halperin and Uri Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM (extended abstract). In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, 1994.

[120] Shay Halperin and Uri Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. *Journal of Algorithms*, 39(1):1 – 46, 2001.

[121] D. Han, A. Agrawal, W. Liao, and A. Choudhary. A novel scalable DBSCAN algorithm with Spark. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1393–1402, 2016.

[122] Nick Harvey. CPSC 536N: Randomized Algorithms, Lecture 4. University of British Columbia, 2015.

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

[124] Qing He, Hai Xia Gu, Qin Wei, and Xu Wang. A novel DBSCAN based on binary local sensitive hashing and binary-KNN representation. *Adv. in MM*, 2017:3695323:1–3695323:9, 2017.

[125] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: a scalable mapreduce-based DBSCAN algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, February 2014.

[126] William Hendrix, Md Mostofa Ali Patwary, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. Parallel hierarchical clustering on shared memory platforms. In *International Conference on High Performance Computing*, pages 1–9, 2012.

[127] Alexander Hinneburg and Daniel A. Keim. An efficient approach to clustering in large multimedia databases with noise. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, page 58–65. AAAI Press, 1998.

[128] Klaus Hinrichs, Jurg Nievergelt, and Peter Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26(5):255–261, 1988.

[129] Qi Hu, Nail A Gumerov, and Ramani Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[130] Qi Hu, Nail A Gumerov, and Ramani Duraiswami. Gpu accelerated fast multipole methods for vortex particle simulation. *Computers & Fluids*, 88:857–865, 2013.

[131] Xiaojuan Hu, Lei Liu, Ningjia Qiu, Di Yang, and Meng Li. A MapReduce-based improvement algorithm for DBSCAN. *Journal of Algorithms & Computational Technology*, 12(1):53–61, 2018.

[132] Xu Hu, Jun Huang, and Minghui Qiu. A communication efficient parallel DBSCAN algorithm based on parameter server. In *ACM on Conference on Information and Knowledge Management (CIKM)*, pages 2107–2110, 2017.

[133] Fang Huang, Qiang Zhu, Ji Zhou, Jian Tao, Xiaocheng Zhou, Du Jin, Xicheng Tan, and Lizhe Wang. Research on the parallelization of the DBSCAN clustering algorithm for spatial data mining based on the Spark platform. *Remote Sensing*, 9(12), 2017.

[134] M. Huang and F. Bian. A grid and density based fast spatial clustering algorithm. In *International Conference on Artificial Intelligence and Computational Intelligence*, volume 4, pages 260–263, 2009.

[135] Ramón Huerta, Thiago Schiavo Mosqueiro, Jordi Fonollosa, Nikolai F. Rulkov, and Irene Rodríguez-Luján. Online humidity and temperature decorrelation of chemical sensors for continuous monitoring. volume 157, pages 169–176, 2016.

[136] Intel. Cilk++ programming language, 2010. `http://software.intel.com/en-us/articles/intel-cilk`.

[137] Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/.

[138] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[139] Jennifer Jang and Heinrich Jiang. DBSCAN++: Towards fast and scalable density clustering. In *International Conference on Machine Learning (ICML)*, volume 97, pages 3019–3029, 2019.

[140] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. Dbdc: Density based distributed clustering. In *International Conference on Extending Database Technology (EDBT)*, pages 88–105, 2004.

[141] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. Scalable density-based distributed clustering. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 231–244, 2004.

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

[143] Hua Jiang, Jing Li, Shenghe Yi, Xiangyang Wang, and Xin Hu. A new hybrid method based on partitioning-based DBSCAN and ant clustering. *Expert Systems with Applications*, 38(8):9373 – 9381, 2011.

[144] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In *SIAM International Conference on Data Mining*, pages 246–256, 2004.

[145] Linus Källberg and Thomas Larsson. Accelerated computation of minimum enclosing balls by gpu parallelization and distance filtering. In *Proceedings of SIGRAD 2014, Visual Computing, June 12-13, 2014, Göteborg, Sweden*, number 106, pages 57–65. Linköping University Electronic Press, 2014.

[146] Sanjiv Kapoor and Michiel Smid. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. on Computing*, 25(4):775–796, 1996.

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

[148] Samir Khuller and Yossi Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, April 1995.

[149] Jeong-Hun Kim, Jong-Hyeok Choi, Kwan-Hee Yoo, and Aziz Nasridinov. AA-DBSCAN: an approximate adaptive DBSCAN for finding clusters with varying densities. *The Journal of Supercomputing*, 75(1):142–169, January 2019.

[150] Younghoon Kim, Kyuseok Shim, Min-Soeng Kim, and June Sup Lee. DBCURE-MR: An efficient density-based clustering algorithm for large data using mapreduce. *Information Systems*, 42:15 – 35, 2014.

[151] 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.

[152] Marzena Kryszkiewicz and Piotr Lasek. TI-DBSCAN: Clustering with DBSCAN by means of the triangle inequality. In *Rough Sets and Current Trends in Computing*, pages 60–69, 2010.

[153] YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Scientific and Statistical Database Management*, pages 132–150, 2010.

[154] Thomas Larsson, Gabriele Capannini, and Linus Källberg. Parallel computation of optimal enclosing balls by iterative orthant scan. *Computers & Graphics*, 56:1–10, 2016.

[155] Der-Tsai Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Transactions on Computers*, 100(6):478–487, 1982.

[156] Charles E Leiserson and Ilya B Mirman. How to survive the multicore software revolution (or at least survive the hype). *Cilk Arts*, 1:11, 2008.

[157] Hans-Peter Lenhof and Michiel Smid. Sequential and parallel algorithms for the $k$ closest pairs problem. *International J. of Computational Geometry & Applications*, 5(03):273–288, 1995.

[158] Kingsly Leung and Christopher Leckie. Unsupervised anomaly detection in network intrusion detection using clusters. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*, pages 333–342, 2005.

[159] B. Liu. A fast density-based clustering algorithm for large databases. In *International Conference on Machine Learning and Cybernetics*, pages 996–1000, 2006.

[160] 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.

[161] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. NG-DBSCAN: Scalable density-based clustering for arbitrary data. *Proc. VLDB Endow.*, 10(3):157–168, November 2016.

[162] G. Luo, X. Luo, T. F. Gooch, L. Tian, and K. Qin. A parallel DBSCAN algorithm based on Spark. In *IEEE International Conferences on Big Data and Cloud Computing*, pages 548–553, 2016.

[163] Eden WM Ma and Tommy WS Chow. A new shifting grid clustering algorithm. *Pattern recognition*, 37(3):503–514, 2004.

[164] Philip D. MacKenzie and Quentin F. Stout. Ultrafast expected time parallel algorithms. *J. Algorithms*, 26(1):1–33, 1998.

[165] K. Mahesh Kumar and A. Rama Mohan Reddy. A fast DBSCAN clustering algorithm by accelerating neighbor searching using groups method. *Pattern Recogn.*, 58(C):39–48, October 2016.

[166] Anil Maheshwari, Wolfgang Mulzer, and Michiel Smid. A simple randomized $O(n \log n)$–time closest-pair algorithm in doubling metrics. *arXiv preprint arXiv:2004.05883*, 2020.

[167] S. Mahran and K. Mahar. Using grid for accelerating density-based clustering. In *IEEE International Conference on Computer and Information Technology*, pages 35–40, July 2008.

[168] 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.

[169] 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.

[170] William March, Parikshit Ram, and Alexander Gray. Fast Euclidean minimum spanning tree: Algorithm, analysis, and applications. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 603–612, 2010.

[171] Leland McInnes and John Healy. Accelerated hierarchical density clustering. *arXiv preprint arXiv:1705.07321*, 2017.

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

[173] Henning Meyerhenke. Constructing higher-order Voronoi diagrams in parallel. In *European Workshop on Computational Geometry*, pages 123–126, 2005.

[174] 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.

[175] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[176] Daniel Müllner. Modern hierarchical, agglomerative clustering algorithms, 2011.

[177] S. Näher and Daniel Schmitt. A framework for multi-core implementations of divide and conquer algorithms and its application to the convex hull problem. In *CCCG*, 2008.

[178] Giri Narasimhan and Martin Zachariasen. Geometric minimum spanning trees via well-separated pair decompositions. *ACM Journal of Experimental Algorithmics*, 6:6, 2001.

[179] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313 – 1325, 1995.

[180] Vitaly Osipov, Peter Sanders, and Johannes Singler. The Filter-Kruskal minimum spanning tree algorithm. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 52–61, 2009.

[181] Mark H. Overmars. Dynamization of order decomposable set problems. *J. Algorithms*, 2(3):245–260, 1981.

[182] 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.

[183] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. k. Liao, F. Manne, and A. Choudhary. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 62:1–62:11, 2012.

[184] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. K. Liao, F. Manne, and A. Choudhary. Scalable parallel OPTICS data clustering using graph algorithmic techniques. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 49:1–49:12, 2013.

[185] Md. Mostofa Ali Patwary, Suren Byna, Nadathur Rajagopalan Satish, Narayanan Sundaram, Zarija Lukić, Vadim Roytershteyn, Michael J. Anderson, Yushu Yao, Prabhat, and Pradeep Dubey. BD-CATS: Big data clustering at trillion particle scale. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 6:1–6:12, 2015.

[186] Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Fredrik Manne, Salman Habib, and Pradeep Dubey. PARDICLE: Parallel approximate density-based clustering. In *ACM/IEEE International Conference for High*

*Performance Computing, Networking, Storage and Analysis (SC)*, pages 560–571, 2014.

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

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

[189] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002.

[190] Maria Cristina Pinotti and Geppino Pucci. Parallel algorithms for priority queue operations. *Theoretical Computer Science (TCS)*, 148(1):171–180, 1995.

[191] Madhav Poudel and Michael Gowanlock. Cuda-dclust+: Revisiting early gpu-accelerated dbscan clustering designs. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 354–363. IEEE, 2021.

[192] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, February 1977.

[193] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

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

[195] Michael O. Rabin. Probabilistic algorithms. 1976.

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

[197] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[198] John H. Reif and Sandeep Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7(1):91–117, June 1992.

[199] Alex Rodriguez and Alessandro Laio. Clustering by fast search and find of density peaks. *science*, 344(6191):1492–1496, 2014.

[200] Andrei Sorin Sabau. Survey of clustering based financial fraud detection research. *Informatica Economica*, 16(1):110, 2012.

[201] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data Mining and Knowledge Discovery*, 2(2):169–194, June 1998.

[202] Jörg Sander, Xuejie Qin, Zhiyong Lu, Nan Niu, and Alex Kovarsky. Automatic extraction of clusters from hierarchical clustering representations. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 75–87, 2003.

[203] J. Santos, T. Syed, M. Coelho Naldi, R. J. G. B. Campello, and J. Sander. Hierarchical density-based clustering using mapreduce. *IEEE Transactions on Big Data*, 2019.

[204] A. Sarma, P. Goyal, S. Kumari, A. Wani, J. S. Challa, S. Islam, and N. Goyal. $\mu$dbscan: An exact scalable dbscan algorithm for big data exploiting spatial locality. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2019.

[205] E. Schikuta. Grid-clustering: an efficient hierarchical clustering method for very large data sets. In *Proceedings of 13th International Conference on Pattern Recognition*, volume 2, pages 101–105 vol.2, 1996.

[206] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Trans. Database Syst.*, 42(3):19:1–19:21, July 2017.

[207] Christian Schwarz and Michiel Smid. An $O(n \log n \log \log n)$ algorithm for the on-line closest pair problem. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA '92, page 280–285, USA, 1992. Society for Industrial and Applied Mathematics.

[208] Christian Schwarz, Michiel Smid, and Jack Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12(1):18–29, 1994.

[209] 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.

[210] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *IEEE Symposium on Foundations of Computer Science*, page 151–162, 1975.

[211] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[212] J. Shun and G. E. Blelloch. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Transactions on Parallel Computing*, 1(1):8:1–8:20, October 2014.

[213] Julian Shun. *Shared-memory parallelism can be simple, fast, and scalable*. Morgan & Claypool, 2017.

[214] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 152–163, 2013.

[215] 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 (SPAA)*, pages 68–70, 2012.

[216] Michiel Smid. Maintaining the minimal distance of a point set in less than linear time. In *Algorithms Rev.*, pages 33–44, 1991.

[217] Michiel Smid. Maintaining the minimal distance of a point set in polylogarithmic time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–6, 1991.

[218] Hwanjun Song and Jae-Gil Lee. RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1173–1187, 2018.

[219] D Srikanth, Kishore Kothapalli, R Govindarajulu, and P Narayanan. Parallelizing two dimensional convex hull on nvidia gpu and cell be. In *International Conference on High Performance Computing (HiPC)*, pages 1–5, 2009.

[220] Ayal Stein, Eran Geva, and Jihad El-Sana. Cudahull: Fast parallel 3d convex hull on the gpu. *Computers & Graphics*, 36(4):265–271, 2012. Applications of Geometry Processing.

[221] 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.

[222] Yihan Sun and Guy E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.

[223] Kenneth J. Supowit. New techniques for some dynamic closest-point and farthest-point problems. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 84–90, 1990.

[224] Min Tang, Jie yi Zhao, Ruo feng Tong, and Dinesh Manocha. Gpu accelerated convex hull computation. *Computers & Graphics*, 36(5):498–506, 2012.

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

[226] 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.

[227] Cheng-Fa Tsai and Chien-Tsung Wu. GF-DBSCAN: A new efficient and effective data clustering technique for large databases. In *WSEAS International Conference on Multimedia Systems & Signal Processing*, pages 231–236, 2009.

[228] 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.

[229] Stanley Tzeng and John D Owens. Finding convex hulls using quickhull on the gpu. *arXiv preprint arXiv:1201.2936*, 2012.

[230] O. Uncu, W. A. Gruver, D. B. Kotak, D. Sabaz, Z. Alibhai, and C. Ng. GRIDBSCAN: Grid density-based spatial clustering of applications with noise. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 2976–2981, 2006.

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

[232] Remco C. Veltkamp. The $\gamma$-neighborhood graph. *Computational Geometry*, 1(4):227–246, 1992.

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

[234] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques, 2010.

[235] P. Viswanath and V. Suresh Babu. Rough-DBSCAN: A fast hybrid density based clustering method for large data sets. *Pattern Recognition Letters*, 30(16):1477 – 1488, 2009.

[236] P. Viswanath and R. Pinkesh. l-DBSCAN : A fast hybrid density based clustering method. In *International Conference on Pattern Recognition (ICPR)*, volume 1, pages 912–915, 2006.

[237] 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.

[238] Wei Wang, Jiong Yang, and Richard R. Muntz. Sting: A statistical information grid approach to spatial data mining. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 186–195, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[239] 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.

[240] Yiqiu Wang, Rahul Yesantharao, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. ParGeo: A Library for Parallel Computational Geometry. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms (ESA 2022)*, volume 244 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 88:1–88:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[241] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. Geograph: A framework for graph processing on geometric data. *SIGOPS Oper. Syst. Rev.*, 55(1):38–46, June 2021.

[242] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. Pargeo: a library for parallel computational geometry. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 450–452, 2022.

[243] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. A Parallel Batch-Dynamic Data Structure for the Closest Pair Problem. In Kevin Buchin and Éric Colin de Verdière, editors, *37th International Symposium on Computational Geometry (SoCG 2021)*, volume 189 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 60:1–60:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[244] 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.

[245] Benjamin Welton, Evan Samanas, and Barton P. Miller. Mr. scan: Extreme scale density-based clustering using a tree-based network of GPGPU nodes. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 84:1–84:11, 2013.

[246] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, pages 359–370, 1991.

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

[248] Eden W.M. Ma and Tommy W.S. Chow. A new shifting grid clustering algorithm. *Pattern Recognition*, 37(3):503–514, 2004.

[249] Yi-Pu Wu, Jin-Jiang Guo, and Xue-Jie Zhang. A linear DBSCAN algorithm based on LSH. In *International Conference on Machine Learning and Cybernetics*, volume 5, pages 2608–2614, August 2007.

[250] Yan Xiang Fu, Wei Zhong Zhao, and Huifang Ma. Research on parallel DBSCAN algorithm design based on MapReduce. *Advanced Materials Research*, 301-303:1133–1138, July 2011.

[251] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. A fast parallel clustering algorithm for large spatial databases. *Data Mining and Knowledge Discovery*, 3(3):263–290, September 1999.

[252] Ying Xu, Victor Olman, and Dong Xu. Minimum spanning trees for gene expression data clustering. *Genome Informatics*, 12:24–33, February 2001.

[253] Zhao Yanchang and Song Junde. Gdilc: a grid-based density-isoline clustering algorithm. In *2001 International Conferences on Info-Tech and Info-Net. Proceedings (Cat. No.01EX479)*, volume 3, pages 140–145 vol.3, 2001.

[254] Andrew Chi-Chih. Yao. On constructing minimum spanning trees in $k$-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736, 1982.

[255] Rahul Yesantharao, Yiqiu Wang, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic $k$ d-trees. *arXiv preprint arXiv:2112.06188*, 2021.

[256] Meichen Yu, Arjan Hillebrand, Prejaas Tewarie, Jil Meier, Bob van Dijk, Piet Van Mieghem, and Cornelis Jan Stam. Hierarchical clustering in minimum spanning trees. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(2):023107, 2015.

[257] Yanwei Yu, Jindong Zhao, Xiaodong Wang, Qin Wang, and Yonggang Zhang. Cludoop: An efficient distributed density-based clustering for big data using Hadoop. *Int. J. Distrib. Sen. Netw.*, 2015:2:2–2:2, January 2015.

[258] Mingwei Zhao, Yang Liu, and Rongan Jiang. Research of wavecluster algorithm in intrusion detection system. In *2008 International Conference on Computational Intelligence and Security*, volume 1, pages 259–263, 2008.

[259] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. Learning transportation mode from raw gps data for geographic applications on the web. In *International Conference on World Wide Web*, pages 247–256, 2008.