Algorithms for Triangles, Cones & Peaks

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

von der KIT-Fakultät für Informatik des

Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Daniel Funke, M.Sc.

aus Altenburg

Gemäß der Promotionsordnung vom 12. Januar 2017

Tag der mündlichen Prüfung: 30. November 2023

1. Referent:	Prof. Dr. Peter Sanders Karlsruher Institut für Technologie Deutschland
2. Referent:	Prof. Dr. Stefan Funke Universität Stuttgart Deutschland

To my wife Margit and our children Nora and Oskar.

Abstract

Summary: In this dissertation we present efficient algorithms addressing three different geometric objects: triangles, cones and peaks. We devise a novel, parallel divide-and-conquer algorithm to construct Delaunay triangulations in shared and distributed memory. We present the first implementation of an optimal $\mathcal{O}(n \log n)$-time sweepline algorithm to construct the Yao graph—a cone-based geometric spanner. Furthermore, we address the problem of efficiently calculating the isolation of mountain peaks by presenting a novel sweep-plane algorithm.

Three different geometric objects are at the center of this dissertation: triangles, cones and peaks. In computational geometry, *triangles* are the most basic shape for planar subdivisions. Particularly, Delaunay triangulations (DTs) are a widely used for manifold applications in engineering, geographic information systems, telecommunication networks etc. We present two novel parallel algorithms to construct the Delaunay triangulation of a given point set. Yao graphs are geometric spanners that connect each point of a given set to its nearest neighbor in each of k *cones* drawn around it. They are used to aid the construction of Euclidean minimum spanning trees or in wireless networks for topology control and routing. We present the first implementation of an optimal $\mathcal{O}(n \log n)$-time sweepline algorithm to construct Yao graphs. One metric to quantify the importance of a mountain *peak* is its isolation. Isolation measures the distance between a peak and the closest point of higher elevation. Computing this metric from high-resolution digital elevation models (DEMs) requires efficient algorithms. We present a novel sweep-plane algorithm that can calculate the isolation of all peaks on Earth in mere minutes.

In the following, we provide more details on our contributions:

Delaunay Triangulations. We present a novel divide-and-conquer (D&C) algorithm that lends itself equally well to shared- and distributed-memory parallelism. While previous D&C algorithms generally suffer from a complex—often sequential—merge or divide step, we reduce the merging of two partial triangulations to re-triangulating a small subset of their vertices—the *border* vertices—using the same parallel algorithm and combining the three triangulations via parallel hash table lookups. The input point division should yield roughly equal-sized partitions for good load balancing and also result in a small number of border vertices for fast merging. As a further refinement of our algorithm, we devise a data-sensitive divide-step that partitions the input based upon information gained from triangulating a small sample of the input points.

We present a second algorithm specifically designed for application in network generators. Network generators are used to create massive synthetic graphs for algorithm development, testing and benchmarking as well as network analysis. We develop a communication-free Delaunay graph generator that exploits the structure of DTs of uniformly distributed points to minimize the number of redundantly generated points on distributed processing elements. The resulting generator has a near optimal scaling behavior and allows the analysis of Delaunay graphs on an unprecedented scale.

Yao Graphs. An optimal $\mathcal{O}(n \log n)$-time algorithm to construct the Yao graph for a given point set has been proposed by Chang et al. in 1990. Due to its complexity and the numerous required geometric predicates and constructions it has—to the best of our knowledge—never been implemented. Instead, algorithms with a quadratic complexity are used in popular packages to construct Yao graphs. We present the first implementation of Chang et al.'s optimal Yao graph algorithm. We develop and tune the data structures required to achieve the $\mathcal{O}(n \log n)$-time bound and detail algorithmic adaptions necessary to take the original algorithm from theory to practice. Additionally, we propose a new, easy to implement Yao graph construction algorithm based on a uniform grid data structure that outperforms Chang et al.'s algorithm for medium-sized inputs.

Mountain Isolation. With the availability of worldwide digital elevation model (DEM), the isolation of all mountain peaks on Earth can be computed by algorithms. Hitherto, algorithms with a worst-case time bound that is quadratic in the DEM size are used for this, which scale poorly to high-resolution DEMs. We present a novel sweep-plane algorithm that runs in time $\mathcal{O}(n \log n + pT_{\mathrm{NN}})$ where n is the number of sample points in the DEM, p the number of considered peaks and T_{NN} the time for a two-dimensional nearest neighbor query in an appropriate geometric search tree. We refine this algorithm to a two-level approach that has high locality and good parallel scalability. Our algorithmic improvements allow for the use of higher-precision computations to increase the accuracy of the computed isolations. We reduce the time needed to calculate the isolation of every peak on Earth from hours to minutes.

Acknowledgements

First and foremost I would like to thank Peter Sanders for giving me the opportunity to work in his research group. Over the years I could benefit from his vast knowledge and experience; without his guidance this dissertation would not have been possible. I am also grateful to Stefan Funke for being the second referee of this dissertation.

I would like to give special thanks to my former and current colleagues in Peter's research group, not only for the many fruitful discussions but also for the non-fruitful ones in and outside the office: Yaroslav Akhremtsev, Michael Axtmann, Tomáš Balyo, Timo Bingmann, Simon Gog, Lars Gottesbüren, Demian Hespe, Tobias Heuer, Lukas Hübner, Markus Iser, Florian Kurpicz, Moritz Laupichler, Hans-Peter Lehmann, Tobias Maier, Dennis Schieferdecker, Matthias Schimek, Sebastian Schlag, Dominik Schreiber, Christian Schulz, Daniel Seemaier, Jochen Speck, Darren Strash, Tim Niklas Uhl, Marvin Williams and Sascha Witt. In particular, I would like to thank Julian Arz and Lorenz Hübschle for not only being great colleagues but also dear friends.

Furthermore, I thank my students over the years: Jan Ebbing, Damir Ferizovic, David Merkel, Friedrich Schroedter, Philip Zander and particularly Nicolai Hüning and Vincent Winkler for their contributions to this dissertation.

I would like to thank my coauthors of the publications included in this dissertation: Nicolai Hüning, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, Moritz von Looz and Vincent Winkler.

Additionally, I thank Demian Hespe, Daniel Seemaier, Tim Niklas Uhl and Marvin Williams for proofreading parts of this dissertation.

Last but not least, I would like to thank my parents Manuel and Stefanie for their lifelong support of my pursuits and my wife Margit for her support, patience and encouragement over the years. I dedicate this dissertation to her and our children Nora and Oskar.

Table of Contents

1 Introduction

In the physical world around us, triangles, cones and peaks can be found everywhere. *Triangles* have already been studied by Euclid in 300 BC in his treatise *Elements* [Euc56]. In civil engineering, they are often labeled as "the strongest shape" for their rigidity and are used to construct bridges, roofs, and other structures. Triangles are the basic building block to model complex three-dimensional objects. These objects can be visualized and animated in computer graphics or used for simulations in computer-aided engineering. In computational geometry, they are the most basic shape to represent planar subdivisions.

Euclid also already studied another geometrical shape: *cones*. Cones have manifold applications in engineering and construction, including optical lenses, aerodynamic air- and spacecraft design or flow management in fluid mechanics. In computational geometry, they are used to define the class of cone-based spanning graphs. Cones are also found in nature in the form of volcanoes, hills and mountains.

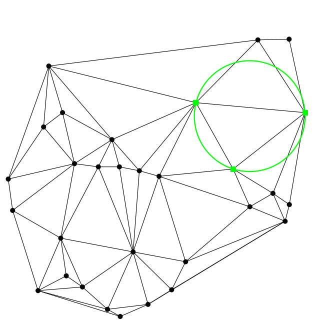
The high-point of these terrain forms—their *peaks*—have fascinated humans since the beginning of time. Many cultures consider them the home of the gods, e.g., the Greek Mount Olympus or the Tibetan Mount Kailash. Whereas people went to the mountains for religious reasons for millennia, it was not until the 18th century that the first mountaineers started to climb mountain peaks for sport. Today, hiking and mountaineering are popular sports and mountain peaks around the world are climbed by thousands of people every year.

In this dissertation we present geometric algorithms addressing each of these three objects. We devise a novel algorithm to compute the *Delaunay triangulation* of a point set in arbitrary dimension. Furthermore, we provide the first implementation of an optimal algorithm to construct the *Yao graph*—a cone-based spanner—of a point set in the plane. Finally, we design a new algorithm to compute the *topographic isolation* of mountain peaks in digital elevation models.

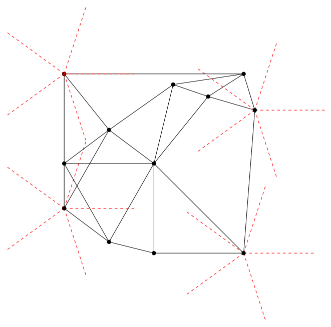
1.1 Overview & Contributions

This dissertation addresses geometric algorithms to compute Delaunay triangulations, Yao graphs and the topographic isolation. In the following we will briefly define each addressed problem and present the contributions of this dissertation.

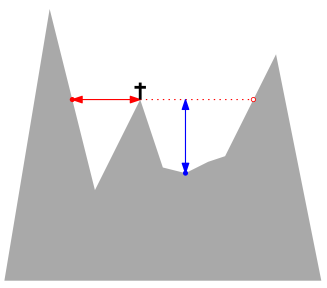
Delaunay Triangulations (Chapter 2). Given a two-dimensional point set $\mathbf{P} = \{p_1, ..., p_n\}$, a triangulation $T(\mathbf{P})$ is a subdivision of the convex hull of $\mathbf{P}$ into triangles, such that the set of the vertices of $T(\mathbf{P})$ coincides with $\mathbf{P}$ and any two triangles of T intersect in a common edge or not at all. The union of all triangles in $T(\mathbf{P})$ is the convex hull of point set $\mathbf{P}$. A triangulation is called *Delaunay triangulation* $DT(\mathbf{P})$ if it satisfies the *empty circle property*: for any triangle $t \in DT(\mathbf{P})$ the circumcircle of t does not contain any point of $\mathbf{P}$ in its interior [Del34], refer to Figure 1.1a for an example. These definitions can be generalized to higher dimensions. Delaunay triangulations are the most widely used triangulations in computational geometry and computer graphics [CDS12]. They are used for the surface reconstruction of complex objects, to compute Voronoi diagrams, to solve partial differential equations and many other applications [KKŽ05].



(a) Delaunay triangulation with exemplary empty circumcircle in green.



(b) Yao graph for ten points and $k = 5$ cones. The five cones are illustrated as red dashed lines around four example points.



(c) Illustration of a mountain's isolation (red) and prominence (blue).

Figure 1.1 Examples of the three geometric objects addressed in this dissertation.

Contribution. In this dissertation we present a novel, parallel, divide-and-conquer (D&C) algorithm to compute the Delaunay triangulation (DT) of a point set in arbitrary dimension. Whereas previous D&C algorithms suffered from either a sequential divide or merge step, all operations of our algorithm are fully parallelized. We combine two partial triangulations by recursively re-triangulating a small subset of their vertices—the *border* vertices—and combining the three triangulations using parallel hash table operations. Our algorithm can use shared-memory or distributed-memory parallelism. The algorithm requires equally-sized partitions for good load balancing and benefits from small border triangulation sizes. We propose multiple partitioning schemes ranging from recursive median splitting to more advanced techniques based on partitioning the triangulation of a small sample of the input points. Our implementation outperforms its competitors on shared-memory machines and scales to up to 2048 cores on distributed-memory ones. Furthermore, we present a new DT construction algorithm specifically designed for uniformly distributed random points stemming from network generators. The results were published in the proceedings of ALENEX, IPDPS and Euro-Par [FS17; FSW19a; Fun+18], in the *Journal of Parallel Distributed Computing* [Fun+19] as well as in accompanying technical reports [FSW19b; Fun+17].

Yao Graphs (Chapter 3). Given a set $\mathbf{P} = \{p_1, ..., p_n\}$ of points in two-dimensional Euclidean space and an integer parameter $k > 1$, the Yao graph $Y_k = (\mathbf{P}, \mathbf{E})$ is a directed graph, connecting every point $p \in \mathbf{P}$ with its nearest neighbor in each of k equiangular cones with apex at p [Yao82], refer to Figure 1.1b for an example. Yao graphs are a cone-based geometric spanner [Dam18]. A t-spanner is a weighted graph, where for any pair of vertices there exists a t-path between them, which is a path with weight at most t times their spatial distance. The parameter t is known as the *stretch factor* of the spanner and can be bounded for Yao graphs with $k \geq 4$ cones [Bar+15]. Yao graphs were originally introduced to construct Euclidean minimum spanning trees [Yao82], but have since been applied to, e.g., wireless networks for topology control and routing [Si+14; SVZ07]. Yao graphs can be generalized to higher dimensions [DGM09], however we focus on two-dimensional graphs in this dissertation.

Contribution. An optimal $\mathcal{O}(n \log n)$-time construction algorithm for Yao graphs was first presented by Chang et al. [CHT90] in 1990. However, due to its intricate nature and its reliance on many geometric primitives, the algorithm has never been implemented and algorithms with an inferior $\mathcal{O}(n^2)$-time bound are used in practice, e.g., in CGAL's cone-based spanners package [NS07; STP22]. We present the first implementation of Chang et al.'s algorithm and take their algorithm from theory to practice. We engineer the data structures required to achieve the $\mathcal{O}(n \log n)$-time bound and provide detailed descriptions of all operations of the algorithm that are missing in the original paper. Despite the algorithm's complexity, we show that it outperforms the state-of-the-art for large inputs. For medium-sized inputs, we propose a new Yao graph construction algorithm based on a uniform grid data structure, that scales better than competing algorithms. The results were published in the proceedings of SEA [FS23b] and in the accompanying technical report [FS23a].

Topographic Isolation (Chapter 4). In order to measure the "significance" of a mountain, two established metrics are the *topographic prominence* and *topographic isolation* of its peak [Gri04]. The isolation specifies the horizontal distance between a peak and the closet point of higher elevation, called isolation limit point (ILP). Peaks with high isolation dominate their surroundings and provide a nice all-round view from the top. Prominence measures the minimum difference in elevation between a peak and the lowest point on a path to reach higher ground. Figure 1.1c visualizes these definitions. Whereas both measures had to be determined by the meticulous inspection of topographic maps in the past, with the advent of digital elevation models (DEMs) they can now be computed algorithmically.

Contribution. In this dissertation we present a novel sweep-plane algorithm to compute the topographic isolation of mountain peaks in DEMs. Our algorithm is highly parallelizable and can calculate the isolation of all peaks on Earth in 4 min, as opposed to ≈10 h required by prior algorithms [KdF17]. We furthermore adapt known geometric search trees to spherical and ellipsoidal surfaces. The results were published in the proceedings of ESA [FHS23a] and the accompanying technical report [FHS23b].

Proximity problems. All problems considered in this dissertation address some form of proximity problem. The nearest neighbor graph (NNG), connecting each point p of a given point set $\mathbf{P}$ with its nearest neighbor in $\mathbf{P} \setminus \{p\}$, is a special case of the Yao graph Y_k with $k = 1$. For $k \geq 1$, the NNG is a subgraph of Y_k [Rah+15]. Furthermore, the NNG is a subgraph of the Euclidean minimum spanning tree (EMST) which is also a subgraph of Y_k [Yao82]. As the EMST is a subgraph of the Delaunay triangulation (DT) as well [dBer+08, Section 9.6], the following inclusions hold:

$$NNG \subseteq EMST \subset \begin{cases} DT \\ Y_k \end{cases} .$$

For the relationship between DTs and Yao graphs, it holds that $DT \cap Y_k \supseteq EMST$. Beyond this, relationship results have only been established for specialized versions of both graphs [Bon+10], with the general case still being an open question.

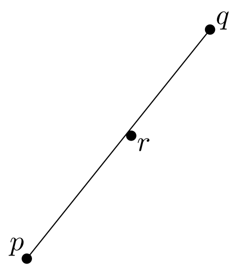
There is no formal relation between any of these graphs with the topographic isolation of a mountain. However, the ILP is defined as the *closest* point of higher elevation to a given peak. Thus, we can define a topographic isolation graph (TIG) with the peaks and ILPs as vertices and edges between peaks and their respective ILP. This graph could be interpreted as a NNG with the additional restriction that for an edge $e = (p, i)$ the ILP i must be of higher elevation than peak p.

1.2 Challenges of Geometric Algorithms

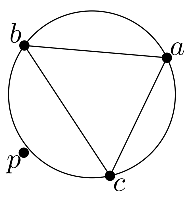
Computational geometry is a branch of computer science that studies algorithms to solve geometric problems [dBer+08]. The studied problems stem from a wide range of applications, including computer graphics, geographic information systems, robotics, computer-aided design and many more. Concrete problems include point location and nearest neighbor searches, line segment intersection, convex hulls, Voronoi diagrams, triangulations, etc. The design of geometric algorithm not only focuses on efficiency in terms of time and space usage, but also on the robustness and the treatment of degenerate cases.

Robustness. Robustness needs to address the limited precision of floating-point arithmetic in computers [She97]. For instance, consider the three nearly collinear points p, q and r pictured in Figure 1.2a. Due to the limited precision of floating-point arithmetic, a test designed to determine the orientation of r with respect to the line through p and q—either left, right or collinear—might return an incorrect result. Such tests are called *predicates* and are basic building block of many geometric algorithms. Figure 1.2 shows examples of predicates used in geometric algorithms, such as the aforementioned ORIENT predicate to determine the orientation of a point with respect to a line, the IN-CIRCLE predicate to test whether a point lies inside the circumcircle of a triangle, and the INTERSECTS predicate to test whether two line segments intersect. In order to ensure robustness, predicates need to be designed such that they always return the correct result. This can be achieved by using exact arithmetic, provided by libraries, such as the GNU Multiple Precision Arithmetic Library (GMP) in conjunction with the GNU Multiple Precision Floating-Point Reliably Library (MPFR) [Fou+07] or the Library of Efficient Data types and Algorithms (LEDA) [MN89]. However, exact arithmetic comes at a significant computational cost. Alternatively, predicates can be designed to use adaptive precision arithmetic, where the precision is increased until the result is guaranteed to be correct [DP02; She97]. Predicates are often designed such that they need to compute the sign of a determinant or polynomial. For adaptive precision predicates, an error bound on the sign of the predicate is calculated in conjunction with the sign itself. If the error is too large to guarantee the correctness of the result, the precision is increased and the test is repeated. The more precise evaluation of the predicate should take advantage of previous results in order to speed up the computation [She97].

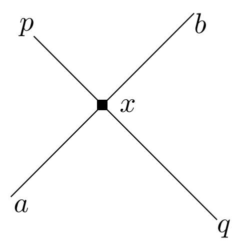
When designing geometric algorithms, the choice of geometric primitives can have a significant impact on the robustness and performance of the algorithm. The DT algorithm presented in Chapter 2 requires merely two geometric predicates: ORIENT and IN-CIRCLE. The Yao graph construction algorithm not only requires the INTERSECTS predicate, but also



(a) ORIENT predicate.



(b) IN-CIRCLE predicate.



(c) INTERSECTS predicate or construction.

Figure 1.2 Examples of predicates used in geometric algorithms.

needs to actually compute the intersection point of the two line segments. Such *constructions* necessarily need to use exact arithmetic to ensure robustness, as the required precision is not known at construction time. We examine the computational impact of these constructions in Chapter 3. The computation of the topographic isolation requires only a distance comparison predicate. However, computing the geodesic distance between points along Earth's surface is computationally expensive. Therefore, our adaptive predicates presented in Chapter 4 use increasingly accurate approximations of Earth to speed up the computation depending on the distance between the input points.

Degeneracies. Degeneracies are special cases of the input for geometric problems that require special treatment [dBer+08]. The kind of degeneracies that need to be addressed depend on the geometric problem and even the algorithm used. For instance, collinear points need to be handled correctly in convex hull algorithms, whereas four cocircular points require consistent tie-breaking rules in DT algorithms. In the theoretical design of a geometric algorithm, degeneracies are often ignored by assuming a *general position* of the input. Real-world data, however, is often far from being in general position. Degeneracies can either be addressed by general mechanisms such as a small perturbation of the input [dBer+08] or by integrating their handling into the algorithm itself. In Chapter 3 we lift the general position assumption of Chang et al.'s Yao graph construction algorithm, by correctly addressing degenerate cases in the algorithm.

1.3 Techniques & Data Structures

There are well-established algorithmic design patterns and data structures that can give rise to efficient and elegant algorithms. In the following we will briefly introduce the techniques and data structures relevant to the algorithms presented in this dissertation.

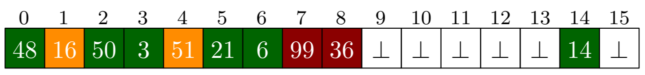
1.3.1 Data Structures

Data structures are fundamental building blocks of algorithms. Often, the choice of data structure has a significant impact on the theoretical complexity and practical performance of an algorithm. In the following we review data structures that are common to several algorithms presented in this dissertation. Figure 1.3 shows examples of the data structures discussed in this section.

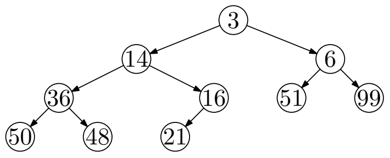
We differentiate between abstract data types (ADTs) and concrete data structures. An ADT defines the operations and behavior on data from a user's viewpoint. A concrete data structure implements an ADT and defines the internal representation of the data and the algorithms used for the ADT's operations.

Hash Tables. Hash tables are one way to implement the *dictionary* ADT, which stores a set of n elements $\mathbf{E}$ and supports the operations INSERT, DELETE and FIND [San+19, Chapter 4]. Each element $e \in \mathbf{E}$ is associated with a *key* $k(e) \in \mathbb{U}$, for some key universe $\mathbb{U}$. If $|\mathbb{U}| \sim n$, the dictionary can be implemented as an array of size $|\mathbb{U}|$. However, often $|\mathbb{U}| \gg n$, which would lead to a prohibitively large array and a waste of memory, due to many unused entries. A hash table is a data structure that stores the elements of $\mathbf{E}$ in an array of size $m \sim n$, where each element e is stored at index $h(k(e))$ for a hash function $h : \mathbb{U} \to [0..m-1]$. For brevity, we use $h(e)$ instead of $h(k(e))$. Elements $e_1, e_2 \in \mathbf{E}$, $e_1 \neq e_2$, might be stored at the same index $h(e_1) = h(e_2)$, which is called a *collision*. There are two main approaches to resolve collisions: *closed* and *open addressing*. In closed addressing, each array slot i is the

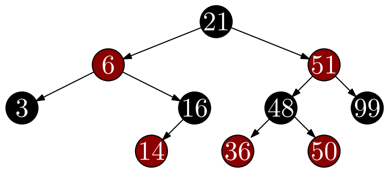
	x	$h(x)$	$d(x)$
1st	21	5	0
2nd	3	3	0
3rd	51	3	1
4th	48	0	0
5th	16	0	1
6th	6	6	0
7th	99	3	4
8th	50	2	0
9th	36	4	4
10th	14	14	0



(a) Linear probing hash table.



(b) Min-heap.



(c) Red-black tree.

Figure 1.3 Examples of the data structures discussed in Section 1.3.1. The elements arrive in the order shown in the table on the left. The hash function is given by $h(x) = x \bmod 16$ and $d(x)$ denotes the distance between $h(x)$ and x's final position in the hash table. The red-black tree stores elements not only in the leaves, but also in the inner nodes.

start of a linked list that stores all elements e with the same hash value $h(e) = i$, a method called *chaining*. In open addressing, each array slot stores exactly one element e. If the array slot at $h(e)$ is already occupied, e is stored in the next free slot of the array. There are several strategies to find the next free slot, including linear probing, quadratic probing and double hashing [Cor+09, Chapter 11].

The performance of hash tables depends on the choice of the hash function h, the load factor $\alpha = {}^{n}\!/_{m}$ of the array and the collision resolution strategy. With a careful implementation and suitable h, both open and closed addressing hash tables can achieve expected $\mathcal{O}(1)$ time for all operations. Both kind of hash tables can be parallelized, though the state-of-the-art shared-memory parallel hash table uses linear probing and carefully designed atomic operations [MSD16].

In this dissertation. Parallel hash tables are at the core of our D&C DT construction algorithms presented in Chapter 2. They enable us to efficiently merge partial triangulations in parallel. Our Yao graph construction algorithm uses a hash table to cache expensive geometric constructions (Chapter 3).

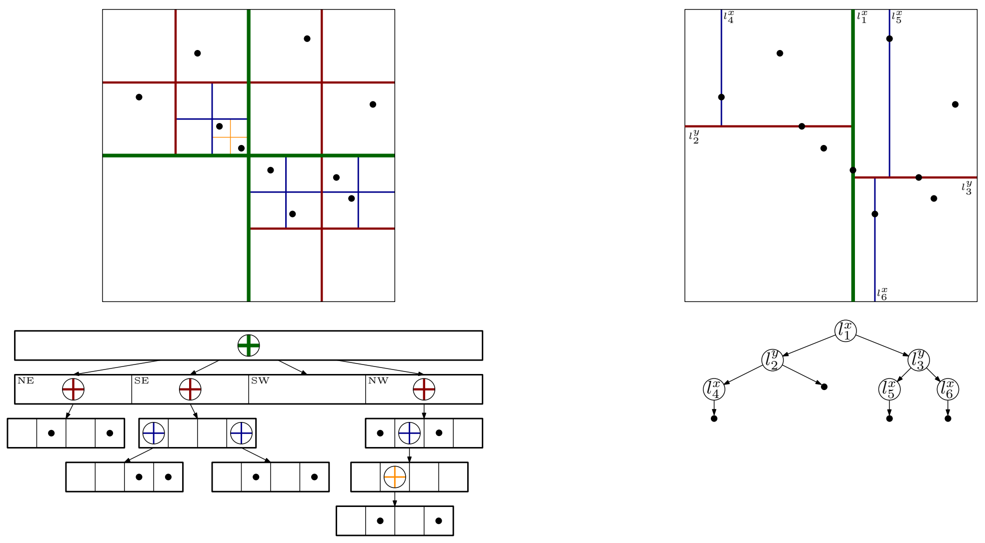
Priority Queues. A priority queue (PQ) is an ADT that stores a set of n elements $\mathbf{E}$ and supports the operations INSERT, TOP and POP [San+19, Chapter 6]. An element $e \in \mathbf{E}$ is inserted into the PQ with an associated *priority* $p(e) \in \mathbb{R}$. The TOP operation returns the element $e \in \mathbf{E}$ with the highest priority $p(e)$ or $\perp$ if the PQ is empty. For a non-empty PQ, POP removes the highest priority element from the queue. An *addressable* PQ additionally supports the operations REMOVE and UPDATE to remove an arbitrary element or change its priority. PQs are often implemented as binary heaps, which are binary trees that satisfy the *heap property*: for any node v in the tree, the priority of v is larger than the priority of its children. With binary heaps, the TOP operation takes $\mathcal{O}(1)$ time, whereas all other operations take $\mathcal{O}(\log n)$ time. However, PQs can be implemented in many ways, e.g., as pairing heaps, binomial heaps or Fibonacci heaps [San+19, Chapter 6].

In this dissertation. Priority queues are a key component of *sweepline* algorithms, which are discussed in more detail in Section 1.3.2. Sweepline algorithms use a PQ to store the elements that should be processed by the algorithm in the order in which they are swept over—called *events*. For our Yao graph construction algorithm in Chapter 3, we present a two-part priority queue that separates input elements and dynamically created events into an array and an addressable binary heap, respectively, thus reducing the cost of INSERT and REMOVE operations on the heap. In our sweep-plane algorithm to compute mountain isolations, all events are known prior to execution, thus a sorted array can serve as PQ (Chapter 4).

Search Trees. A search tree is an ADT that stores a set of n elements $\mathbf{E}$ and supports the operations INSERT, DELETE and FIND [San+19, Chapter 7]. Again, each element $e \in \mathbf{E}$ is associated with a key $k(e)$. The FIND operation requires a key k and returns the element $e = \arg\min_{x \in \mathbf{E}} (k(x) \geq k)$, i.e., the element e with the smallest key $k(e) \geq k$. These operations could be implemented using a sorted array, yielding $\mathcal{O}(\log n)$-time queries and $\mathcal{O}(n)$-time insertions and deletions. However, by using a tree data structure an (amortized) $\mathcal{O}(\log n)$-time bound can be achieved for all operations. The elements of $\mathbf{E}$ are stored in the leaves of the tree and each inner node v stores a key k_v and two pointers to its left and right child. For every element e_l of the left subtree $k(e_l) \leq k_v$ whereas for all e_r of the right subtree $k(e_r) > k_v$. At each node v, the FIND operation needs to compare the search key k with k_v and descent into the left or right subtree accordingly until the leaf with element e is found. To INSERT an element e, first the appropriate leaf for its key $k(e)$ is found using the FIND operation. Then, the leaf is replaced by an inner node with e as its left child and the original leaf as its right child. The time complexity of all tree operations therefore depends on the height of the tree. With a naive INSERT implementation, the tree could degenerate into a linked list and the FIND operation would take $\Theta(n)$ time. Numerous data structures have been proposed that ensure a *balanced* tree of logarithmic height after insertions and removals, including AVL trees [AL62], red-black trees [GS78], and B-trees [BM70]. The schemes vary in the strictness of their balancing criteria and the required overhead for rebalancing the tree in terms of time and space.

In this dissertation. The block-wise triangulation data structure for our D&C DT construction algorithm employs a balanced binary search tree to efficiently navigate to individual blocks of the triangulation (Chapter 2). Our Yao graph construction algorithm uses a balanced binary search tree to represent the current state of the sweepline (Chapter 3). As the comparison of keys requires the evaluation of an expensive geometric predicate, we present several optimizations of the tree operations specific to our algorithm.

Geometric Search Trees. In geometric search trees, the key of an element e represents its coordinates in a D-dimensional (Euclidean) space, $k(e) \in \mathbb{R}^D$ [dBer+08, Chapter 5]. We therefore refer to the elements of the tree as *points*. Since the semantics of the FIND operation are not well-defined for higher dimensions, it is replaced by the RANGE operation that reports all points within a given query range and the NEAREST operation that reports the point closest to a given query point. Various geometric search trees have been proposed in the literature, including k-D trees [Ben75], quad-trees [FB74], range trees [Ben79], and R-trees [Gut84]. All of these data structures recursively partition the input space into smaller subspaces in order to answer queries efficiently. Nevertheless, while range queries can be answered in $\mathcal{O}(\log n)$ worst-case time, nearest neighbor queries require $\mathcal{O}(n)$ time in the worst case—even in balanced trees—and only achieve a $\mathcal{O}(\log n)$-time bound on average.



(a) Quad-tree. **(b)** k-D tree.

Figure 1.4 Two-dimensional examples of the geometric search trees discussed in Section 1.3.1. The quad-tree is unbalanced even if all input points are known a priori. The depicted k-D tree stores the splitting line as well as the splitting element in each inner node.

Quad-trees recursively partition the input space into four quadrants. The inner nodes store pointers to their four children and leaves store the subset of input points that lie within their quadrant. As the splitting into quadrants does not take the distribution of the input points into account, the tree can become unbalanced if the input points are not sufficiently uniformly distributed, as shown in Figure 1.4a. In fact, the size and depth of a quad-tree cannot be bounded in terms of the number of input points but only regarding the minimum distance between any two points and the size of the input space [dBer+08, Lemma 14.1]. The technique can be generalized to $D > 2$ dimensions, where the input space is recursively partitioned into 2^D hypercubes at each inner node the tree. These generalized trees are referred to as oct-trees in three dimensions [Mea82] and hyperoct-trees for $D > 3$ [PY85]. However, they suffer from the *curse of dimensionality*, as the number of hypercubes grows exponentially with the dimensionality D [KM17].

In k-D trees, the input space is recursively partitioned into two half-spaces. Each node stores a splitting hyperplane and pointers to its two children. The dimension of the hyperplane is chosen in a cyclic manner among the k dimensions of the input space. The median point of the input regarding the splitting dimension is chosen as the splitting element and partitions the input into two equally-sized subsets. If all input points are known at construction time, this results in a balanced tree with logarithmic height. However, if the input points are inserted one-by-one, the tree can degenerate and rebalancing could require the reorganization of large parts of the tree [Sam90, Chapter 2]. For high-dimensional spaces, k-D trees also suffer from the curse of dimensionality, with an efficiency no better than exhaustive search if the number of elements n is not $n \gg 2^k$ [TOG17, Chapter 39].

In this dissertation. Our sample-based load balancing strategies for DT construction use a static k-D tree for input point distribution (Section 2.5). The sweep-plane data structure for the topographic isolation algorithm employs a dynamic k-D tree with the top-most levels pre-built in a quad-tree-like manner (Chapter 4).

1.3.2 Techniques

The two main design principles used for the algorithms in this dissertation are *divide-and-conquer* and *sweepline* algorithms. In the following we will briefly introduce these techniques and their application to geometric problems. Both techniques are illustrated in Figure 1.5 for a two-dimensional convex hull algorithm.

Divide-and-Conquer Algorithms. Divide-and-conquer (D&C) is a fundamental paradigm in algorithm design. Already Gauss used a D&C algorithm to compute the coefficients of a finite Fourier series in 1805 [HJB85]. John von Neumann was the first to develop and properly analyze a D&C algorithm specifically for a computer—merge sort in 1945 [ORe13].

In a D&C algorithm, a problem is recursively divided into smaller subproblems until a base case is reached that can be solved directly [Smi85]. The solutions to the subproblems are then combined to solve the original problem. The D&C paradigm can lead to efficient and elegant algorithms and has been applied to manifold problems, e.g., sorting, matrix multiplication and fast Fourier transform. In computational geometry, D&C algorithms have been proposed for a wide range of problems. Bentley [Ben80] presents a general framework for geometric D&C algorithms that can be applied to range searching, closest pair and all nearest neighbor problems. Prior to the recursive calls in a D&C algorithm, the current state of the algorithm needs to be stored on a stack. Therefore, the extra space required by the algorithm depends on the depth of the recursion tree. Bose et al. [Bos+07] present space-efficient geometric D&C algorithms with $\mathcal{O}(1)$ extra space for the closet pair and bichromatic closest pair problems. Aggarwal et al. [ACG88] are the first to propose a D&C algorithm to compute the Delaunay triangulation.

The divide-and-conquer paradigm lends itself well to the design of parallel algorithms, as the subproblems can be solved independently. However, for an efficient parallelization, the combination of the subproblems also needs to be parallelized. Otherwise, it becomes a sequential bottleneck, especially in the upper parts of the recursion tree, where the subproblems might be large.

In this dissertation. In Chapter 2, we present a novel D&C algorithm to compute the DT where all operations are fully parallelized. Our DT algorithm for network generators also follows the D&C paradigm on a global scale. As the algorithm produces a distributed triangulation it can avoid the ascent of the recursion tree by computing the final local DT directly on the bottom of the tree. The multi-pass algorithm to compute the topographic isolation proposed in Chapter 4 can also be considered a D&C algorithm in the wider sense: the data is already divided in the form of the tiles of the DEM, and the conquer step generates a global view on the mountain peaks relevant to each tile for further processing.

Sweepline Algorithms. Many geometric problems can be solved efficiently using a *sweepline* approach [dBer+08, Chapter 2]. The technique employs a conceptual line that sweeps over the input space and processes the input elements in the order in which they are swept. A sweepline data structure stores all elements that are currently *active* in some sense with respect to the sweepline. This approach effectively reduces the dimensionality of the problem. For two-dimensional problems the sweepline data structure is often a one-dimensional (geometric) search tree. For $D > 2$ dimensions, the sweep-*plane* data structure is $D - 1$-dimensional.

A sweepline algorithm uses two main components: the *event queue* and the *sweepline data structure*. An event could be an input point, the beginning or end of a line segment, an intersection, etc. The event queue stores all events in the order in which they are swept

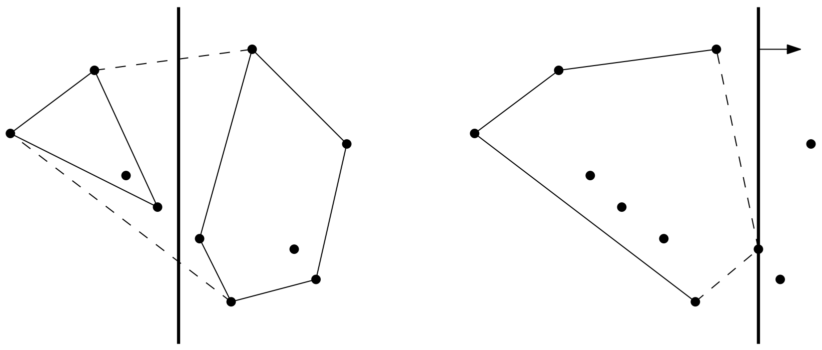

(a) Divide-and-conquer algorithm. **(b)** Sweepline algorithm.

Figure 1.5 Illustration of a D&C and a sweepline algorithm to construct the convex hull of a two-dimensional point set. For the D&C algorithm, the dashed edges are determined during the merging of the left and right subproblems. For the sweepline algorithm, the dashed edges are added by the currently processed input point.

over by the sweepline. Often the sweepline moves either along the x or y axis, but it could also follow an arbitrary direction or a circular path. If all events are known prior to the execution of the algorithm, the event queue can be implemented as a sorted array. Otherwise, if events are dynamically created and/or removed during the execution, an (addressable) PQ is used. The sweepline data structures stores currently *active* events, e.g., line segments that currently intersect the sweepline, input points within a specific range or unfinished cells. The processing of an event may involve queries to the sweepline data structure, insertion or removal of elements from it, and/or the creation of new events.

The sweepline paradigm originates in the *scanline* algorithm for rendering images in computer graphics [Wyl+67]. Shamos and Hoey [SH76] were the first to combine the scanline technique with efficient data structures to address geometric problems. They present a $\mathcal{O}(n \log n)$-time sweepline algorithm to check for intersections of n line segments using a balanced binary search tree. A similar $\mathcal{O}((n+k) \log n)$-time sweepline algorithm to report all k intersections of n given line segments is presented by Bentley and Ottmann [BO79]. Since its inception, the sweepline paradigm as been applied to compute Voronoi diagrams [For87], Delaunay triangulations [Žal05] and convex hulls [Bor19]. The technique has been generalized to higher dimensions [AGP90], e.g., it is applied to three-dimensional DT and convex hull construction by Sinclair [Sin16].

In this dissertation. We implement a sweepline algorithm to construct Yao graphs in Chapter 3, which was originally proposed by Chang et al. [CHT90] but has not been implemented before. In Chapter 4, we use the sweepline paradigm to design a novel sweep-plane algorithm to compute the topographic isolation of mountain peaks in DEMs.

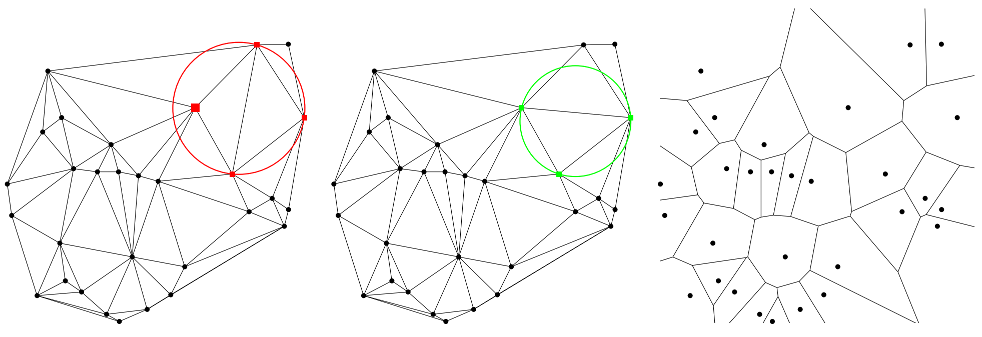
2 Parallel Construction of Delaunay Triangulations

Summary: Computing the Delaunay triangulation (DT) of a given point set in $\mathbb{R}^D$ is one of the fundamental operations in computational geometry. We present a novel divide-and-conquer (D&C) algorithm that lends itself equally well to shared- and distributed-memory parallelism. While previous D&C algorithms generally suffer from a complex—often sequential—merge or divide step, we reduce the merging of two partial triangulations to re-triangulating a small subset of their vertices—the border *vertices—using the same parallel algorithm and combining the three triangulations via parallel hash table lookups. The algorithm is sensitive to the quality of the input point partitioning in the divide step. The input point division should yield roughly equal-sized partitions for good load balancing and also result in a small number of border vertices for fast merging. We present an advanced divide-step based on partitioning the triangulation of a small sample of the input points. Additionally, we present a specialized DT construction algorithm for network generator applications, that exploits the uniform distribution of the generated points.*

Attribution: This chapter is based on three publications [FS17; FSW19a; Fun+19]. The author of this dissertation was the main author and contributor of [FS17; FSW19a], on the parallel DT construction algorithm and the sample-based partitioning scheme. The Sections 2.4 and 2.5 are mainly based on these publications or the corresponding technical report [FSW19b]. Both sections also contain further results, in particular the analysis in Section 2.4.1.2 and the point assignment strategies in Section 2.5.1. Vincent Winkler provided the implementation of the sample-based partitioning scheme and performed experiments for his Bachelor thesis [Win18]. Peter Sanders provided editing for both publications. Sebastian Lamm is the main author and contributor of the network generator KaGen. His Master thesis [Lam17] lead to the two publications [Fun+18; Fun+19], with contributions from Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash and Moritz von Looz. The author of this dissertation contributed the Delaunay network generator, which is one among several classes of graph generators implemented in KaGen. The description of the DT construction algorithm in Section 2.6.2 is largely based on [Fun+17; Fun+19].

2.1 Introduction

A triangulation $T(\mathbf{P})$ of a given point set $\mathbf{P}$ in Euclidean space $\mathbb{R}^2$ is the subdivision of $\mathbf{P}$'s convex hull into triangles. A Delaunay triangulation $DT(\mathbf{P})$ of $\mathbf{P}$ requires that no point of $\mathbf{P}$ is inside the circumcircle of any triangle in $DT(\mathbf{P})$ [Del34]. The DT is the dual graph of the Voronoi diagram $VD(\mathbf{P})$ [Vor08] and has numerous applications in computer graphics, data visualization, terrain modeling, pattern recognition and as mesh for the finite element method [CDS12; KKŽ05]. Figure 2.1 illustrates these three subdivisions. Delaunay triangulations can be generalized to D-dimensional space [Del34]. Computing the DT of a



(a) $T(\mathbf{P})$. **(b)** $DT(\mathbf{P})$. **(c)** $VD(\mathbf{P})$.

Figure 2.1 Non-Delaunay and Delaunay triangulation as well as a Voronoi diagram of a point set $\mathbf{P} \in \mathbb{R}^2$. The red point in Figure (a) violates the Delaunay criterion. In Figure (b), the circumcircles of the flipped triangles do not contain any other point of $\mathbf{P}$.

point set is thus one of the fundamental operations in computational geometry. Therefore, many algorithms to efficiently compute the DT have been proposed as surveyed by Su and Drysdale [SD95] and well implemented codes exist [HS15; She96]. With ever-increasing input sizes, research interest has shifted from sequential algorithms towards parallel ones [Bat+10; Ble+99; CG12; CMS98; FLP14; KKŽ05], with shared-memory parallelism for algorithms in two dimensions receiving most of the attention. Distributed-memory algorithms however—as studied by Cignoni et al. [CMS98] and Lee et al. [LPP01]—are required to cope with massive triangulations exceeding the memory limitations of a single machine.

In this chapter we present a novel divide-and-conquer (D&C) DT construction algorithm for arbitrary dimension that lends itself equally well to shared- and distributed-memory parallelism and thus hybrid parallelization. Previous D&C DT algorithms suffer from a complex—often sequential—divide or merge step [CMS98; LPP01]. We reduce the merging of two partial triangulations to re-triangulating a small subset of their vertices—so-called *border* vertices—using the same parallel algorithm and combining the three triangulations via hash table lookups. All steps required for the merging—identification of relevant vertices, triangulation and combining the partial DTs—are performed in parallel. Only minor modifications are required to adapt our algorithm from a shared-memory machine to a message-based distributed-memory cluster.

The division of the input points in the divide-step needs to address a twofold sensitivity to the point distribution: the partitions need to be approximately equal-sized for good load balancing, while the number of *border vertices* needs to be minimized for fast merging. This requires partitions that have many internal Delaunay edges but only few external ones, i.e., a graph partitioning of the DT graph. We therefore propose a sample-based divide-step that approximates this graph partitioning by triangulating and partitioning a small sample of the input points, and divides the original input point set based upon it.

Additionally, we address Delaunay triangulations for network generators. Network generators provide large-scale synthetic graph instances with controllable parameters, e.g., for the design and analysis of scalable algorithms. Lamm [Lam17] presents a novel technique to generate graphs on a massive scale. By making use of pseudorandomization and divide-and-conquer schemes, their generators follow a communication-free paradigm. We present a Delaunay graph generator that makes use of their technique and exploits the structure of DTs of uniformly distributed points. The resulting generator has a near optimal scaling behavior and allows the analysis of Delaunay graphs on an unprecedented scale.

Outline. This chapter is structured as follows: we present the problem definition and a survey of related work in Sections 2.2 and 2.3. Subsequently, our novel D&C DT construction algorithm is described in Section 2.4. Section 2.5 introduces and evaluates our advanced sample-based divide-step. Lastly, a special DT construction algorithm for uniformly distributed input points, as used in network generators, is presented in Section 2.6.

2.2 Definitions

D-simplices are a generalization of triangles ($D = 2$) to D-dimensional space. A D-simplex s is a D-dimensional polytope, i.e., the convex hull of $D + 1$ points. The convex hull of a subset of size $m + 1$ of these $D + 1$ points is called an m-face of s. Specifically, the 0-faces are the vertices of s and the $(D - 1)$-faces are its facets.

▶ **Definition 1** (Triangulation). *Given a D-dimensional point set $\mathbf{P} = \{p_1, p_2, ..., p_n\}$ with $p_i \in \mathbb{R}^D$ for all $i \in [1..n]$, a triangulation $T(\mathbf{P})$ is a subdivision of the convex hull of $\mathbf{P}$ into D-simplices, such that the set of the vertices of $T(\mathbf{P})$ coincides with $\mathbf{P}$ and any two simplices of T intersect in a common $D - 1$ facet or not at all. The union of all simplices in $T(\mathbf{P})$ is the convex hull of point set $\mathbf{P}$.*

▶ **Definition 2** (Delaunay Triangulation [Del34]). *A Delaunay triangulation $DT(\mathbf{P})$ is a triangulation of $\mathbf{P}$ such that no point of $\mathbf{P}$ is inside the circumhypersphere of any simplex in $DT(\mathbf{P})$.*

The DT of n points contains $\mathcal{O}(n^{\lceil \frac{D}{2} \rceil})$ simplices [Sei95] and can be computed in $\mathcal{O}(n \log n)$ time for $D = 2$ [For87] and $\mathcal{O}(n^{\lceil \frac{D}{2} \rceil})$ time for $D \geq 3$ [CDS12]. In the plane, the DT maximizes the minimum angle found in any triangle of $DT(\mathbf{P})$, however it does neither minimize the maximum angle nor the edge length [ETW92]. A DT is the supergraph of the Euclidean minimum spanning tree (EMST) [GR69], relative neighborhood graph (RNG) [Tou80], nearest neighbor graph (NNG) [EPY97] and Gabriel graph [GS69] of the same point set [dBer+08]. DTs are geometric spanners. A t-spanner is a weighted graph, where for any pair of vertices there exists a t-path between them, which is a path with weight at most t times their spatial distance. The parameter t is known as the *stretch factor* of the spanner. Upper bounds on the stretch factor of DTs have been the subject of extensive research. Currently, the best known bound for planar DTs is $t < 1.83$ [TSJ19].

▶ **Theorem 3** (Uniqueness of DTs [Del34]). *If the points of $\mathbf{P}$ are in* general position*, i.e., no $D + 2$ points lie on a common D-hypersphere, $DT(\mathbf{P})$ is unique.*

▶ **Definition 4** (Locally Delaunay [CDS12; Del34; Joe91]). *Let f be a facet of triangulation $DT(\mathbf{P})$. Facet f is said to be* locally Delaunay *if and only if*

a) f is only part of one simplex $s \in DT(\mathbf{P})$, or
b) f is shared by simplices s_1 and s_2 of $DT(\mathbf{P})$ and has an open circumhypersphere containing no vertex of simplices s_1 or s_2.

▶ **Lemma 5** (Delaunay Lemma [CDS12; Del34]). *Let $T(\mathbf{P})$ be triangulation of point set $\mathbf{P}$. The following statements are equivalent:*

a) T is Delaunay;
b) every simplex of T is Delaunay;
c) every facet of T is Delaunay;
d) every facet of T is locally Delaunay.

Table 2.1 Properties of DT construction algorithms in this dissertation and reviewed related work. The speedup given is the maximum reported by the authors for uniformly distributed points.

Algorithm	3-D	*d*-D	Shar. mem.	Dist. mem.	Speedup
	incremental insertion				
Kohout et al. [KKŽ05]	✓	✗	✓	✗	3.7 (4 PEs)
Batista et al. [Bat+10]	✓	✗	✓	✗	7 (8 PEs)
Lo [Lo12]	✓	✗	✓	✗	10 (12 PEs)
	divide-and-conquer				
Aggarwal et al. [ACG88]	✗	✗	✓	✗	theory
Cignoni et al. [Cig+93; CMS98]	✓	✓	✗	✓	3.4 (16 PEs)
Chen [Che10]	✓	✗	✗	✓	4.5 (8 PEs)
Lee et al. [LPP01]	✗	✗	✗	✓	12 (32 PEs)
Fuetterling et al. [FLP14]	✗	✗	✓	✗	13 (16 PEs)
Chen and Gotsman [CG12]	✗	✗	✓	✗	7.5 (8 PEs)
in this dissertation	✓	✓	✓	✓	260 (2048 PEs)

Notation. We refer to the set of vertices of simplex s by vertices(s); the individual i-th vertex is denoted $\text{vertices}_i(s)$. We employ a similar notation for the set of neighboring simplices of s—neighbors(s)—and an individual neighbor i—$\text{neighbors}_i(s)$.

2.3 Related Work

A survey of parallel DT construction algorithms in two and three dimensions for shared memory is given by Kohout et al. [KKŽ05]. The proposed algorithms are either based on parallel incremental insertion or a D&C approach.

Incremental Insertion Algorithms. Parallel incremental insertion algorithms are generally bootstrapped with a sequentially obtained initial triangulation of a subset of the input points. Subsequently, the rest of the points can be inserted in parallel by identifying the surrounding simplex for each point, removing it and re-triangulating the resulting cavity with the inserted point and the facets of the surrounding simplices [Bat+10; CS96; KKŽ05]. The Delaunay property of the re-triangulated region is ensured by performing local flips [Joe91; KKŽ05]. To avoid simultaneous access to the same simplex during re-triangulation, locks need to be employed. Various locking strategies are studied in [Bat+10; KKŽ05]. The algorithm of Batista et al. is the basis for the parallel DT construction algorithm found in the Computational Geometry Algorithms Library (CGAL) [HS15]. This potential for contention renders parallel incremental insertion very sensitive to the input point distribution and limits the attainable speedup. Moreover, it requires fine-grained communication, which is prohibitive in the distributed-memory setting. To avoid communication, Lo [Lo12] partitions the input points into zones. Each zone is further subdivided into cells. Neighboring zones exchange the points of their adjacent cells, in order to be able to construct the triangulation without synchronization. They report a speedup of ≈ 10 for 12 cores with shared memory for uniformly distributed points, however, no comparison with other implementations is made.

Divide-and-conquer Algorithms. A parallel D&C algorithm for DTs was first proposed by Aggarwal et al. [ACG88]. The input points are partitioned along a vertical line into blocks, which are triangulated in parallel. These partial triangulations are stitched together in an expensive merge step, which can only be performed by one processing element, thus limiting speedup. As non-Delaunay simplices might be introduced to the triangulation during stitching, corrective steps are required to restore the Delaunay property. In the worst case, the necessary corrections can spread throughout the entire triangulation [KKŽ05]. The authors do not address load balancing or prescribe how to determine the location of the splitting line.

A different approach is pursued by Cignoni et al. for three- and arbitrary-dimensional DTs [Cig+93; CMS98]. They divide the input by cutting (hyper)-planes and firstly construct the simplices of the triangulation crossing those planes. The algorithm continues to build the triangulation in the divided regions in parallel, no further merging is necessary. However, their division step is expensive and sequential and thus limits scalability. The authors mention that the regions should be of roughly equal cardinality for load balancing, but do not go into the details of the partitioning.

Chen [Che10] improves on Cignoni et al.'s work by calculating the *affected zone*, comprising the set of simplices that are indeterminate, i.e., a point outside the convex hull of the sub-triangulation may still influence them. The merging of two partial triangulations can then be reduced to merging the affected zones. Using a distributed-memory setting, they report speedups of up to 4.5 for clusters of 8 processing elements (PEs) and uniformly distributed points. Further studies on distributed-memory machines are presented by Lee et al. [LPP01]. They partition the input according to paths of Delaunay edges obtained from a lower convex hull projection [Ble+99]. The individual partitions can then be triangulated without further merging. They report a speedup of ≈ 12 for a machine with 32 PEs and a uniform distribution of input points. Both, Chen [Che10] and Lee et al. [LPP01] explicitly require splitting along the median of the input points for load balancing.

To the best of our knowledge no algorithm has been shown to scale well to clusters with hundreds of PEs. Table 2.1 provides an overview of the discussed literature and compares some properties of the proposed algorithms.

Further Related Work. Chen and Gotsman [CG12] propose an entirely different approach compared to the previously discussed algorithms to compute the DT. They *localize* the computation of the DT by computing the Delaunay neighbors for each point individually. This affords for almost linear speedup. It remains to be seen whether their approach generalizes to three and higher dimensions.

Fuetterling et al. [FLP14] present a novel data structure for D&C-based DT algorithms, the *linear floating point quad-tree* (LFQT) based on the Morton codes of the input point coordinates. The geometrical structure of the quad-tree allows for efficient subdivision of the input during the recursive descent and its numerical structure minimizes the need for exact arithmetic. Although their data structure should generalize to arbitrary dimension, they only report—very favorable—results for single threaded as well as multithreaded performance for computing the DT in two dimensions.

The subject of input partitioning has received more attention in the meshing community than for DT construction algorithms. A *mesh* of a point set $\mathbf{P}$ is a triangulation of every point in $\mathbf{P}$ and possibly more—so-called *Steiner points*—to refine the triangulation [CDS12]. Chrisochoides [Chr06] surveys algorithms for parallel mesh generation and differentiates between continuous domain decomposition—using quad- or oct-trees—and discrete domain decomposition using an initial coarse mesh that is partitioned into submeshes, trying to minimize the surface-to-volume ratio of the submeshes. Chrisochoides and Nave [CN00] propose an algorithm that meshes the subproblems via incremental insertion using the Bowyer-Watson algorithm [Bow81; Wat81].

Cao et al. [Cao+14] present an algorithm that uses a general-purpose graphics processing unit (GPGPU) to accelerate the construction of DTs for large point sets. Their algorithm constructs a near-Delaunay triangulation on the GPGPU using an incremental insertion algorithm and subsequently repairs non-Delaunay simplices on the CPU using a star splaying approach [She05].

Recently, machine learning techniques have been applied to the problem of DT construction. Sharp and Ovsjanikov [SO20] present neural networks to construct a general triangulation from a set of points. Their approach seems to be adaptable to produce triangulations fulfilling the Delaunay criterion. Rakotosaona et al. [Rak+21] use machine learning to reconstruct two-dimensional Delaunay surface elements from three-dimensional point clouds.

2.4 Parallel Divide-and-Conquer Algorithms

In this section we present our novel DT construction algorithm. We first describe the shared-memory algorithm in Section 2.4.1. The modifications necessary to adapt our algorithm to a distributed-memory setting are presented in Section 2.4.2. Section 2.4.3 highlights some technical details of our implementation. We evaluate our algorithms in Section 2.4.4.

2.4.1 Shared-Memory Algorithm

The main novelty of our D&C DT algorithm is its fully parallelizable merge step. The merging of two partial triangulations relies on re-triangulating a small subset of *border* points of both triangulations with the same parallel DT algorithm. Border points are vertices of simplices that might violate the Delaunay property for some point of the other partition and hence need to be re-triangulated for a valid DT of the combined point set. All steps necessary to identify these points and to combine the two partial triangulations with their border triangulation are fully parallelized.

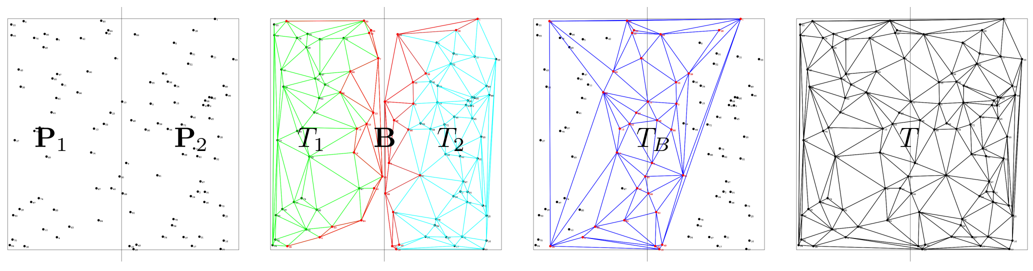
2.4.1.1 Algorithm

This section describes the operations of our D&C DT algorithm in detail. All line numbers given in the following refer to Algorithm 2.1. Figure 2.2 gives a two-dimensional example execution of our algorithm.

Partitioning. Given the set of input points $\mathbf{P} = \{p_1, ..., p_n\}$ and a recursion level r, if the number of points is below a certain threshold or a recursion depth of $\log P$ for P processors has been reached, an efficient sequential DT algorithm is used to solve the base case. Otherwise, our recursive divide-and-conquer algorithm is employed. Firstly, the splitting dimension k is determined following one of various schemes:

- constant, predetermined splitting dimension—similar to Aggarwal et al. [ACG88];
- cyclic choice of the splitting dimension—similar to k-D trees [Ben75]; or
- the dimension with largest extend of the bounding box of the input points.

The input points are then partitioned across the selected dimension k according to the median point. More sophisticated partitioning schemes are presented in Section 2.5. Both partitions are recursively triangulated in parallel, yielding triangulations T_1 and T_2 (Line 5).



(a) partitioning **(b)** partial DTs **(c)** border DT **(d)** final DT

Figure 2.2 Example of a two-dimensional triangulation. Infinite simplices are omitted for clarity.

Algorithm 2.1 Delaunay($\mathbf{P}, r$): shared-memory parallel D&C algorithm.

```
Input:   Points $\mathbf{P} = \{p_1, ..., p_n\}$ with $p_i \in \mathbb{R}^D$, recursion level $r$
Output:  Delaunay triangulation $T$
 1: if $n < N \vee r = \log P$ then                                    ▷ for $P$ processors
 2:     return sequentialDelaunay($\mathbf{P}$)                           ▷ base case
 3: $k \leftarrow$ splittingDimension($\mathbf{P}$)
 4: $(\mathbf{P}_1 \;\; \mathbf{P}_2) = (\{p_1, ..., p_s\} \;\; \{p_{s+1}, ..., p_n\}) \leftarrow$ divide($\mathbf{P}, k$)   ▷ partition points in dim. $k$
 5: $\mathbf{T} = (T_1 \;\; T_2) \leftarrow$ (Delaunay($\mathbf{P}_1, r+1$)  Delaunay($\mathbf{P}_2, r+1$))   ▷ in parallel
Border triangulation:
 6: $\mathbf{B} \leftarrow \varnothing$                                    ▷ empty set of border simplices
 7: $\mathbf{Q} \leftarrow$ convexHull($T_1$) $\cup$ convexHull($T_2$)      ▷ initialize work queue with convex hull
 8: parfor $s_{i,x} \in \mathbf{Q}$ do                       ▷ simplex originating from triangulation $T_i$
 9:     mark($s_{i,x}$)                                   ▷ only process each simplex once
10:     if circumsphere($s_{i,x}$) $\cap$ boundingBox($T_j$) $\neq \varnothing$, with $i \neq j$ then
11:         $\mathbf{B} \cup= \{s_{i,x}\}$ ▷ circumsphere intersects other partition $\Rightarrow s_{i,x}$ is a border simplex
12:         for $s_{i,y} \in$ neighbors($s_{i,x}$) $\wedge \neg$marked($s_{i,y}$) do        ▷ process all neighbors
13:             mark($s_{i,y}$);   $\mathbf{Q} \cup= \{s_{i,y}\}$
14: $T_B \leftarrow$ Delaunay(vertices($\mathbf{B}$), $r+1$)             ▷ triangulate points of border simplices
Merging:
15: $T \leftarrow (T_1 \cup T_2) \setminus \mathbf{B}$;   $\mathbf{Q} \leftarrow \varnothing$        ▷ merge partial triangulations stripped from border
16: parfor $s_b \in T_B$ do                              ▷ merge simplices from border triangulation
17:     if vertices($s_b$) $\not\subset \mathbf{P}_1 \wedge$ vertices($s_b$) $\not\subset \mathbf{P}_2$ then
18:         $T \cup= \{s_b\}$;   $\mathbf{Q} \cup= \{s_b\}$                        ▷ $s_b$ spans both partitions
19:     else
20:         if $\exists s \in \mathbf{B}$ : vertices($s$) = vertices($s_b$) then
21:             $T \cup= \{s_b\}$;   $\mathbf{Q} \cup= \{s_b\}$                  ▷ $s_b$ replaces border simplex
Neighborhood update:
22: parfor $s_x \in \mathbf{Q}$ do                            ▷ update neighbors of inserted simplices
23:     mark($s_x$)                                       ▷ only process each simplex once
24:     for $d \in [1..D+1]$ do
25:         if neighbors$_d$($s_x$) $\notin T$ then              ▷ neighbor not in triangulation anymore
26:             $C \leftarrow \{s_c : f_d(s_x) = f_d(s_c)\}$                ▷ candidates with same facet hash
27:             for $s_c \in C$ do
28:                 if |vertices($s_x$) $\cap$ vertices($s_c$)| $= D$ then
29:                     neighbors$_d$($s_x$) $\leftarrow s_c$                        ▷ $s_c$ is neighbor of $s_x$
30:                     if $\neg$marked($s_c$) then
31:                         mark($s_c$);   $\mathbf{Q} \cup= \{s_c\}$          ▷ process $s_c$ if not already marked
32: return $T$
```

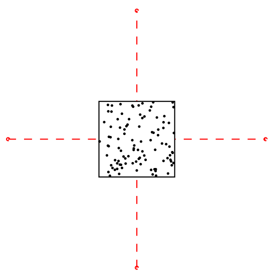


Figure 2.3 Placement of infinite vertices (red) for each face of the bounding box of the input points $\mathbf{P}$, with an offset larger than the diameter of $\mathbf{P}$.

Border Detection. Subsequently, the search for the border simplices $\mathbf{B}$ of both triangulations starts from the convex hull of T_1 and T_2. To identify the convex hull of a triangulation efficiently, Shewchuk [She96] introduces a *vertex at infinity* that forms an *infinite* simplex with every facet of the convex hull. We extend this concept by introducing a vertex at infinity for each face of the axis-aligned bounding box of point set $\mathbf{P}$. The infinite vertices are placed at a multiple of $\mathbf{P}$'s diameter outside the bounding box, as shown in Figure 2.3. This allows for meaningful intersection tests of infinite simplices with another partition's bounding box.

The search employs a parallel work queue initialized with the infinite simplices of T_1 and T_2 (Line 7). A simplex s belongs to the border of triangulation T_i if its circumsphere intersects with the bounding box of the other triangulation T_j, $i \neq j$, i.e., s might still be influenced by a point in partition j. In that case, s is added to $\mathbf{B}$ and all its neighbors are enqueued for processing. A lock-free marking scheme is used to ensure every simplex is processed at most once (Lines 10–13). After completion of the algorithm, all simplices of T_1 and T_2 *not* in $\mathbf{B}$ are completely inside their respective partition and are hence not susceptible to change due to points of the other partition—refer to Section 2.4.1.2. The same criterion is used by Isenburg et al. [Ise+06] and later Wu et al. [WGG11] to define *finalized* triangles of a partition in a streaming computation setting, by Chen [Che10] to determine the *affected zone* and by Lo [Lo12] to determine cells of a zone to be exchanged with neighboring zones. The vertices of all border simplices are collected and recursively triangulated using our D&C algorithm, yielding border triangulation T_B (Line 14).

Merging. The combined triangulation T is composed of simplices from the partial triangulations T_1 and T_2 as well as the border triangulation T_B. Non-border simplices of T_1 and T_2 can be immediately added to T, as no point of the other partition can lie within their circumsphere. The border simplices $\mathbf{B}$ are discarded, since they potentially violate the Delaunay property for some point of the opposite partition. For a simplex $s_B \in T_B$ to be added to T, one of two conditions needs to be fulfilled (Lines 17–20):

a) s_B consists of vertices from both partitions, or
b) s_B is contained within one partition but replaces a previously found border simplex.

The first condition treats simplices crossing the border of T_1 and T_2, which could not have been found before. As s_B fulfills the Delaunay property with respect to the border point set, it also fulfills it with respect to both partition point sets [Che10]. The second condition re-adds simplices of the border that have been confirmed to not only fulfill the Delaunay property with respect to their own partition but also with respect to the border point set and hence the other partition [Lo12]. If all vertices of s_B are fully contained in one partition but no simplex with equal vertices has been previously found in the respective partial triangulation,

s_B is discarded, as it must violate the Delaunay property for a point of that partition not belonging to the border point set, following the uniqueness of the DT for a point set in general position [Del34]. Refer to Section 2.4.1.2 for further analysis of these operations.

Simplices with equal vertices can be efficiently found by using a hash table of the discarded border simplices. The lookup key is a *simplex hash* $h_s(s)$—the exclusive or of a *vertex hash* $h_v(\cdot)$ of each vertex of s,

$$h_s(s) := \bigoplus_{i<D+1} h_v(\text{vertices}_i(s)).$$

For a suitable $h_v(\cdot)$, $h_s(\cdot)$ is efficiently computable, commutative and contributions of individual vertices can be easily extracted. The latter two properties are important for the subsequent neighborhood update. Refer to Section 2.4.3 for details about the choice of $h_v(\cdot)$.

Neighborhood Update. Finally, the neighborhood-relations of the newly inserted simplices need to be established and the relations of some of the previously existing simplices need to be updated. For each neighbor $d \in [1..D+1]$ of a simplex s it is determined whether the currently designated neighbor is valid—i.e., is not some placeholder value and still part of triangulation T—or needs updating. In the latter case, the simplex $s_n \in T$ is determined that shares the facet opposite of $\text{vertices}_d(s)$ with s. Simplex s_n is set as the new neighbor and enqueued for updating itself (Lines 25–31). To efficiently find candidates for neighboring simplices of a given simplex s we employ a *facet hash*—denoted $f_i(s)$ for the facet opposite the i-th vertex. The facet hash must be independent of the order of vertices in s and should be efficiently computable from $h_s(s)$. Thus, we exploit the commutativity of $h_s(s)$ and the involutionarity of exclusive or and let

$$f_i(s) := h_s(s) \oplus h_v(\text{vertices}_i(s)).$$

As only simplices of T_1 and T_2 neighboring the border simplices $\mathbf{B}$ as well as simplices added to T from T_B need to update their neighborhood, we can efficiently maintain a facet lookup table during border detection and merging. The convex hull of T is composed of the convex hull of T_1 and T_2 without the simplices belonging to the border plus the simplices of the convex hull of T_B that have been added to T. The necessary data structures can also be efficiently maintained during merging.

2.4.1.2 Analysis

In the following we prove the correctness of our presented algorithm. Furthermore, we analyze its scaling behavior and compare it to that of sequential algorithms. Our proof of correctness partially relies on reducing Algorithm 2.1 to the Bowyer-Watson incremental insertion algorithm (BW algorithm) [Bow81; Wat81]. We will briefly outline the BW algorithm in the following, refer to [CDS12, Section 3.3] for details. The BW algorithm proceeds in three steps to insert a new vertex v into an existing DT T:

1. Point location: find simplex $s \in T$ with v inside s and remove s from T.
2. Cavity search: check all $s_n \in \text{neighbors}(s)$ whether s_n's circumsphere contains v. If so, delete s_n and recursively process all neighbors of s_n. This step results in a star-shaped polyhedral cavity in T.
3. Re-triangulation: for each facet xy of the cavity add simplex $s^* = \{x, y, v\}$ to T.

Correctness. We show that our D&C algorithm produces a valid DT for a point set $\mathbf{P}$ in general position. In the following we consider point sets $\mathbf{P}_1 \cup \mathbf{P}_2 = \mathbf{P}$, the set of border simplices $\mathbf{B}$ and triangulation T obtained by merging T_1, T_2 and T_B according to Algorithm 2.1. Furthermore, let $DT(\mathbf{P})$ denote the true Delaunay triangulation of point set $\mathbf{P}$. Moreover, let $i, j \in \{1, 2\}$ and $i \neq j$. Geometric arguments are given for arbitrary dimensions but purely combinatorial arguments are presented for two dimensions for brevity.

▶ **Corollary 6.** *Given the partitioned point sets $\mathbf{P}_1$ and $\mathbf{P}_2$, T_1 and T_2 are valid DTs for their respective point set. T_B is a valid DT for the points of* $\text{vertices}(\mathbf{B})$.

▶ **Lemma 7.** *$\mathbf{B}$ contains all simplices of T_i whose circumsphere intersects the bounding box of triangulation T_j.*

Proof. The border detection algorithm is equivalent to the depth-first cavity search of the BW algorithm, with imaginary points inserted on the separation (hyper)-plane between T_i and T_j. ◀

▶ **Lemma 8.** *For $i \in \{1, 2\}$, $\forall s \in T_i \setminus \mathbf{B} : s \in DT(\mathbf{P})$, i.e., all non-border simplices of T_i belong to $DT(\mathbf{P})$.*

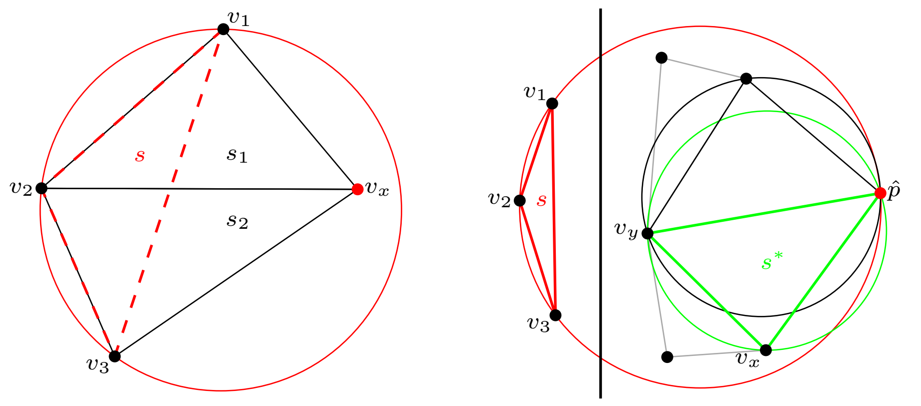
Proof. Following Lemma 7, $\mathbf{B}$ contains all simplices of T_i whose circumsphere intersects the bounding box of T_j. No point of T_j can be closer to any point in T_i than the imaginary points on the separation (hyper)-plane between T_i and T_j, therefore no simplex in $T_i \setminus \mathbf{B}$ would be affected by the cavity search step of the BW algorithm for any point in T_j. Thus, $T_i \setminus \mathbf{B}$ is the unique DT of $\mathbf{P}_i \setminus \text{vertices}(\mathbf{B})$. ◀

▶ **Lemma 9.** *For all simplices $s \in T_B$,*

a) if $\text{vertices}(s) \subset \mathbf{P}_i$ and there exists no *simplex $s' \in T_i$ with $\text{vertices}(s) = \text{vertices}(s')$ then $s \notin DT(\mathbf{P})$;*
b) with s as above, if there is a simplex s' with $\text{vertices}(s) = \text{vertices}(s')$ then $s \in DT(\mathbf{P})$;
c) if $\text{vertices}(s) \not\subset \mathbf{P}_i$ and $\text{vertices}(s) \not\subset \mathbf{P}_j$, then $s \in DT(\mathbf{P})$.

Proof. a) Let $\text{vertices}(s) = \{v_1, v_2, v_3\} \subset \mathbf{P}_i$. As there is no simplex $s' \in T_i$ with vertices $\{v_1, v_2, v_3\}$, there must be, due to Definition 1 and Corollary 6, simplices s_1 and s_2 in T_i with $\text{vertices}(s_1) = \{v_a, v_b, v_x\}$ and $\text{vertices}(s_2) = \{v_a, v_c, v_x\}$ for some $v_x \in \mathbf{P}_i \setminus \{v_1, v_2, v_3\}$ and $v_a \neq v_b \neq v_c \in \{v_1, v_2, v_3\}$, as shown in Figure 2.4a. As T_i is the unique DT for point set $\mathbf{P}_i$ and $\text{vertices}(T_B) \cap \mathbf{P}_i \subset \mathbf{P}_i$, s must violate the Delaunay property for some point $p \in \mathbf{P}_i \setminus \text{vertices}(T_B)$ and therefore cannot be in $DT(\mathbf{P})$, since s_1 and s_2 are valid simplices of $DT(\mathbf{P}_i)$.

b) Simplex s with $\text{vertices}(s) = \{v_1, v_2, v_3\} \subset \mathbf{P}_i$ fulfills the Delaunay property for point set $\mathbf{P}_i$ and point set $\mathbf{P}_i \cup \text{vertices}(\mathbf{B}) \supset \mathbf{P}_i$, as $s \in T_i$ and $s \in T_B$ (Corollary 6). Assume s would violate the Delaunay property for some point $\hat{p} \in \mathbf{P}_j \setminus \text{vertices}(\mathbf{B})$. The circumsphere of any simplex $s^* \in T_j$ with $\hat{p} \in \text{vertices}(s^*)$ cannot intersect the bounding box of T_i, as otherwise s^* would be in $\mathbf{B}$ and $\hat{p}$ in $\text{vertices}(\mathbf{B})$ (Lemma 7). Furthermore, s^* cannot belong to the convex hull of T_j, as all simplices of the convex hull are in $\mathbf{B}$. Consider $\text{vertices}(s^*) = \{v_x, v_y, \hat{p}\}$ with $v_x, v_y \in \mathbf{P}_j$ and $v_x, v_y \in \text{vertices}(\mathbf{B})$, i.e., $\hat{p}$ is the point of s^* farthest removed from T_j's border, as v_x and v_y also belong to some border simplex. As $\text{vertices}(s) \in \mathbf{P}_i$, the circumsphere of s would need to intersect T_j's bounding box in order for s to violate the Delaunay criterion for $\hat{p}$, therefore it would also need to violate the Delaunay property for v_x and/or v_y, refer to Figure 2.4b. Thus, s would not be in T_B (Corollary 6). Ergo, $\hat{p}$ cannot exist and $s \in DT(\mathbf{P})$.



(a) Lemma 9 a): simplex s (red, dashed) violates the DT criterion for v_x and cannot be in $DT(\mathbf{P})$
.

(b) Lemma 9 b): the circumcircle of simplex s (red) needs to contain v_y in order for it to contain $\hat{p}$.

Figure 2.4 Illustrations for the proof of Lemma 9.

c) Consider simplex s with vertices$(s) = \{v_1, v_2, v_3\}$ partitioned into two disjunct blocks vertices$(s) = \mathbf{v}_i \cup \mathbf{v}_j$ with $\mathbf{v}_i \subset \mathbf{P}_i$ and $\mathbf{v}_j \subset \mathbf{P}_j$. As s contains vertices from both partitions, s cannot be in either T_i or T_j. By Corollary 6, s fulfills the Delaunay property in T_B. Following Lemma 7, s cannot violate the Delaunay property in $\mathbf{P}_i \setminus$ vertices$(\mathbf{B})$ as all simplices that would be affected by any point from T_j, in particular from $\mathbf{v}_j$, are in $\mathbf{B}$. The same holds for $\mathbf{P}_j \setminus$ vertices$(\mathbf{B})$. Following the same uniqueness argument as above, there are no simplices $s_1, s_2 \in DT($vertices$(\mathbf{B}))$ with vertices$(s_1) = \{v_a, v_b, v_x\}$ and vertices$(s_2) = \{v_a, v_c, v_x\}$ for some $v_x \in$ vertices$(\mathbf{B}) \setminus \{v_1, v_2, v_3\}$ and $v_a \neq v_b \neq v_c \in \{v_1, v_2, v_3\}$, therefore, $s \in DT($vertices$(\mathbf{B}))$ and thus $s \in DT(\mathbf{P})$, as there can be no simplices $s'_1, s'_2 \in DT(\mathbf{P})$, analogously.

◀

▶ **Theorem 10.** *For a given point set* $\mathbf{P}$*, the triangulation* $T(\mathbf{P})$ *computed by Algorithm 2.1 is the Delaunay triangulation* $DT(\mathbf{P})$ *of* $\mathbf{P}$*.*

Proof. The theorem follows from Corollary 6 and Lemmas 7–9. ◀

Runtime Analysis. We now analyze the runtime of our algorithm. Let $T_P(n)$ denote the time required to process n input points with P processing elements (PEs):

$$T_P(n) = \overbrace{\mathcal{O}\left(\frac{n}{P}\right)}^{(\mathbf{P}_1 \quad \mathbf{P}_2)} + \overbrace{2T_{\frac{P}{2}}\left(\frac{n}{2}\right)}^{(T_1 \quad T_2)} + \overbrace{\mathcal{O}\left(\frac{|DT(n)|}{P}\right)}^{\text{border simplices } \mathbf{B}} + \overbrace{T_P\left(|\text{vertices}(\mathbf{B})|\right)}^{\text{border DT } T_B} + \overbrace{\mathcal{O}\left(\frac{|T_B|}{P}\right)}^{\substack{\text{merging \&} \\ \text{neighbor update}}},$$

with $|\cdot|$ denoting the number of simplices in the triangulation. Recall, that the triangulation size is bounded by $\mathcal{O}(n^{\lceil \frac{D}{2} \rceil})$ [Sei95]. Thus, $T_P(n)$ can be simplified to

$$T_P(n) = \mathcal{O}\left(\frac{n^{\lceil \frac{D}{2} \rceil}}{P}\right) + 2T_{\frac{P}{2}}\left(\frac{n}{2}\right) + T_P(\alpha n), \tag{2.1}$$

with $\alpha \in [0, 1]$ denoting the fraction of border vertices.

We will analyze Equation (2.1) in the work-span model [SV82]. In this model, work is defined as the total number of operations performed by the algorithm and span as the longest chain of dependent operations. Alternatively, work can be defined as $T_1(n)$, i.e., the time the algorithm takes with a single PE, and span as $T_\infty(n)$, i.e., the time the algorithm takes with an infinite number of PEs.

Work. To derive a bound for the work $T_1(n)$, let us first, conservatively, assume that half of the vertices belong to the border, i.e., $\alpha = \frac{1}{2}$. This allows us to solve the recurrence with the Master Theorem [BHS80], yielding

$$\begin{aligned} T_1(n) &= \mathcal{O}\left(n^{\lceil \frac{D}{2} \rceil}\right) + 3T_1\left(\frac{n}{2}\right) \qquad\qquad \text{with } c_{\text{crit}} = \log_2 3 \approx 1.58^{\dagger} \\ &\in \begin{cases} \Theta\left(n^{\log_2 3}\right) & \text{for } D = 2 \\ \Theta\left(n^{\lceil \frac{D}{2} \rceil}\right) & \text{for } D \geq 3. \end{cases} \end{aligned}$$

For $D \geq 3$, this work bound matches the bound for sequential algorithms [CDS12], however for $D = 2$, it is worse than the sequential bound of $\mathcal{O}(n \log n)$ [For87].

Bounding $T_1(n)$ for arbitrary α is more involved and requires the use of the Akra-Bazzi theorem [AB98]. It states that the asymptotic behavior of a recurrence of the form

$$T(n) = g(n) + \sum_{i=1}^{k} a_i T\left(b_i n\right),$$

with $a_i \in (0, \infty)$ and $b_i \in (0, 1)$ for $i \in [1..k]$, is bounded by

$$T(n) \in \Theta\left(n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}} du\right)\right),$$

with p being the unique solution of the equation $\sum_{i=1}^{k} a_i b_i^p = 1$. Applying this theorem to Equation (2.1) yields

$$\begin{aligned} T_1(n) &\in \Theta\left(n^p \left(1 + \int_1^n \frac{u^{\lceil \frac{D}{2} \rceil}}{u^{p+1}} du\right)\right) \\ &= \Theta\left(\frac{1}{\lceil \frac{D}{2} \rceil - p} n^{\lceil \frac{D}{2} \rceil} + \frac{\lceil \frac{D}{2} \rceil - p - 1}{\lceil \frac{D}{2} \rceil - p} n^p\right) \in \begin{cases} \Theta\left(n^{\lceil \frac{D}{2} \rceil}\right) & \text{for } p \leq \lceil \frac{D}{2} \rceil \\ \Theta(n^p) & \text{for } p > \lceil \frac{D}{2} \rceil \end{cases} \end{aligned}$$

with $2\left(\frac{1}{2}\right)^p + \alpha^p = 1$. As a closed-form expression for p is elusive, we resort to numerical methods to determine p for varying values of α. Figure 2.5 shows the resulting p for $\alpha \in [0, 1]$. We call a value of α *critical* for a given dimension D, if the resulting p is equal to $\lceil \frac{D}{2} \rceil$. As shown in Table 2.2, for two dimensions, the critical value for α is zero, meaning that we can not achieve the work bound of the sequential algorithm. However, for three and four dimensions, the critical value for α is $\frac{1}{\sqrt{2}}$, resulting in the same work bound as the sequential algorithm if less than 70 % of the vertices belong to the border. Except for worst-case inputs, the fraction of border vertices is usually much smaller than this. Table 2.2 also shows the resulting p for different values of α. Setting $\alpha = \frac{1}{2}$, results in $p = \log_2 3$, validating our previous analysis with the Master Theorem.

[†] For recurrences of the form $T(n) = g(n) + aT(\frac{n}{b})$, c_{crit} is defined as $\log_b a$ [BHS80].

Table 2.2 Critical α for different dimensions D and resulting p for varying α.

D	Critical α
2	0.0
3	$\frac{1}{\sqrt{2}} \approx 0.7$
4	$\frac{1}{\sqrt{2}} \approx 0.7$
5	≈ 0.9

α	p
1	∞
$1/2$	$\log_2 3 \approx 1.58$
$1/4$	$\log_2\left(1+\sqrt{2}\right) \approx 1.27$
$1/10$	≈ 1.12

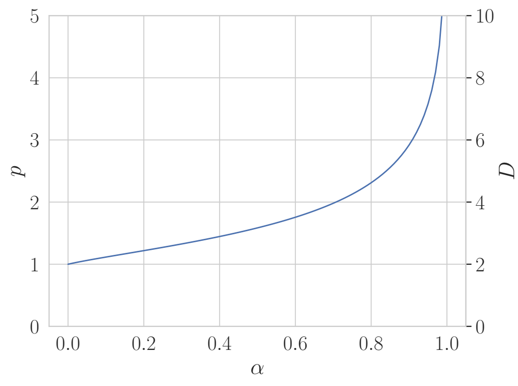


Figure 2.5 Plot of $2\left(\frac{1}{2}\right)^p + \alpha^p = 1$ for $\alpha \in [0,1]$ and resulting p. The right axis shows the critical α for different dimensions D.

Span. Equation (2.1) has two base cases:

$$T_P(n = N) = T_{\text{Seq}}(N) \qquad\qquad T_{P=1}(n) = T_{\text{Seq}}(n)$$

for some fixed N. In order to bound the span $T_\infty(n)$, only the first base case is relevant, as $P = \infty$. We need to consider

$$\frac{n}{2^a} = N \qquad\qquad n\alpha^b = N$$

$$a = \log_2 \frac{n}{N} \qquad\qquad b = \log_{\alpha^{-1}} \frac{n}{N}$$

with $T_\infty(n) = \max(a, b)$. Thus, for $\alpha \leq \frac{1}{2}$, the span is bounded by $\log_2 n$ and by $\log_{\alpha^{-1}} n$ otherwise. In practice, the span is dominated by the first recursive descent which usually terminates in the second base case, while the second descent terminates in the first base case after only a few levels of recursion, if any, for most inputs. Therefore, in practice, the span is bounded by $\log_2 P$.

2.4.2 Distributed-Memory Algorithm

The previously presented divide-and-conquer algorithm can be applied to a distributed-memory model with explicit message-based communication. The general idea of the approach remains unchanged, only slight modifications are required to account for the incomplete information each processing element (PE) has about the input points and hence the resulting global triangulation.

Each of the P PEs holds a portion of the input points, $\mathbf{P} = \{\mathbf{P}_1 \cup \ldots \cup \mathbf{P}_P\}$ with $\mathbf{P}_i = \{p_{i,1}, \ldots, p_{i,n_i}\}$. In the following, we assume that a PE i belongs to exactly one partition

Algorithm 2.2 Delaunay($\mathbf{P}, \mathbf{C}$): distributed-memory parallel D&C algorithm.

```
Input:  Point set $\mathbf{P} = \{p_1, ..., p_n\}$, PEs of partition $\mathbf{C}$
Output: local view $T$ of Delaunay triangulation $DT(\bigcup_{j \in \mathbf{C}} \mathbf{P}@j)$
 1: if $\Sigma_{j \in \mathbf{C}} n@j < N \vee |\mathbf{C}| = 1$ then                ▷ base case
 2:     return DelaunayBase($\mathbf{P}, \mathbf{C}$)
 3: $S \leftarrow$ localVertexStatistics($\mathbf{P}$)                        ▷ local min, max and median
 4: $S_{\text{all}} \leftarrow$ allReduce($S, \mathbf{C}$)                       ▷ global min, max and median
 5: $k \leftarrow$ splittingDimension($S_{\text{all}}$)
 6: $p = \text{median}(S_k) \geq \text{median}(S_{\text{all},k})$          ▷ PE's side of splitting plane in dim. $k$
 7: $\mathbf{C}' \leftarrow \{j : p@j = p \quad \forall j \in \mathbf{C}\}$          ▷ set of all PEs on same side of splitting plane
 8: $T \leftarrow$ Delaunay($\mathbf{P}, \mathbf{C}'$)                          ▷ triangulate own partition
 9: $\mathbf{B} \leftarrow$ borderSimplices($T, \mathbf{C}, \mathbf{C}'$)                ▷ simplices across splitting plane
10: $\mathbf{B} \leftarrow$ sparseAllToAll($\mathbf{B}, \mathbf{C}'$)              ▷ receive border simplices from neighboring PEs
11: $\mathbf{C}_B \leftarrow \{j : \mathbf{B}@j \neq \varnothing \quad \forall j \in \mathbf{C}\}$          ▷ PEs with non-empty border
12: if $\mathbf{B} \neq \varnothing$ then                                  ▷ PE belongs to border
13:     $T_B \leftarrow$ Delaunay(vertices($\mathbf{B}$), $\mathbf{C}_B$)          ▷ triangulate border vertices
14:     $T \leftarrow$ merge($T, T_B, \mathbf{B}, \mathbf{P}, \mathbf{C}'$)       ▷ merge triangulations and update neighborhood
15: return $T$
```

in each partitioning step, i.e., all its points $\mathbf{P}_i$ are either on one side of the splitting plane or the other. This holds naturally for e.g., geospatial data read from pre-tiled files. Data not adhering to this assumption require *one* additional all-to-all communication in the first recursive descent to move the input points to their respective PE.

Algorithm 2.2 presents the modified D&C algorithm from the viewpoint of PE i, with $x@j$ denoting the value of x at PE j. A partition of nodes is represented by set $\mathbf{C}$, e.g., at recursion level zero $\mathbf{C} = \{1, ..., P\}$. At a given recursion level, if the number of input points in the current partition $\mathbf{C}$ lies below a certain threshold or there is only one PE left in the partition, the base case Algorithm 2.6—described at the end of this section—is invoked to compute the Delaunay triangulation of the points of $\mathbf{C}$. In the recursive case, the local minimum, maximum and median of the input points are computed. These statistics are gathered by all PEs of the partition to compute the global values. The splitting dimension is determined according to the global statistics, following the same schemes as for the shared-memory implementation. The partition $\mathbf{C}$ is reduced to $\mathbf{C}'$, containing only PEs on the same side of the splitting plane as PE i. The recursive call with partition $\mathbf{C}'$ yields PE i's local view T on the triangulation of the points in $\bigcup_{j \in \mathbf{C}'} \mathbf{P}_j$—denoted $DT(\mathbf{C}')$. It holds that $T = \{s \in DT(\mathbf{C}') : |\,\text{vertices}(s) \cap \mathbf{P}_i| \geq 1\}$, i.e., PE i stores every simplex of $DT(\mathbf{C}')$ that contains at least one vertex from the input points of PE i.

Subsequently, the border simplices of the local triangulation T are determined. Algorithm 2.3 follows the same principle as described in the previous section. A simplex is added to the local border simplex set $\mathbf{B}$ if its circumsphere intersects with the bounding box of the other partition $\mathbf{C} \setminus \mathbf{C}'$, which can be computed without additional communication. Since PE i only has a local view T on $DT(\mathbf{C}')$ and the border detection algorithm starts its search from the convex hull of $DT(\mathbf{C}')$, there might be a simplex $\hat{s} \in T$ which belongs to the border of $DT(\mathbf{C}')$ but is only reachable from the convex hull of $DT(\mathbf{C}')$ via a simplex $\hat{s}'$ of the convex hull only stored at PE j. That is the case if vertices($\hat{s}$) contains only one vertex from $\mathbf{P}_i$ and vertices($\hat{s}'$) is fully contained in $\mathbf{P}_j$ and hence $\hat{s}' \notin T$. As $\hat{s}$ and $\hat{s}'$ are neighbors, at least one vertex of $\hat{s}$ is in $\mathbf{P}_j$ and therefore $\hat{s} \in T$ at PE j. Thus, $\hat{s}$ will be identified

Algorithm 2.3 borderSimplices($T, \mathbf{C}, \mathbf{C}'$): border simplex detection.

```
Input:  Triangulation $T$, PEs of partition $\mathbf{C}$ and sub-partition $\mathbf{C}'$
Output: border simplices $\mathbf{B}$
 1: $\bar{B} \leftarrow$ boundingBox($\mathbf{C}$) − boundingBox($\mathbf{C}'$)   ▷ compute bounds of other partition
 2: $\mathbf{Q} \leftarrow$ convexHull($T$)                                      ▷ initialize work queue
 3: $\mathbf{B} = \varnothing$
 4: parfor $s \in \mathbf{Q}$ do
 5:     mark($s$)                                                            ▷ only process each simplex once
 6:     if circumsphere($s$) $\cap \bar{B} \neq \varnothing$ then
 7:         $\mathbf{B} \cup \{s\}$                                          ▷ circumsphere intersects other partition
 8:         for $s_n \in$ neighbors($s$) do                                   ▷ process all neighbors
 9:             if $\neg$ marked($s_n$) then
10:                 mark($s_n$);   $\mathbf{Q} \cup \{s_n\}$
11: return $\mathbf{B}$
```

Algorithm 2.4 merge($T, T_B, \mathbf{B}, \mathbf{P}, \mathbf{C}$): merge border simplices into T.

```
Input:  Triangulation $T$, border triangulation $T_B$
        border simplices $\mathbf{B}$, points $\mathbf{P}$, PEs of partition $\mathbf{C}$
Output: merged triangulation $T$
 1: $T \leftarrow T \setminus \mathbf{B}$                                    ▷ strip border simplices
 2: $\mathbf{Q} \leftarrow \varnothing$                                      ▷ work queue for neighbor updates
 3: parfor $s_{B,x} \in T_B$ do
 4:     if $\exists k$ : vertices$_k(s_{B,x}) \in \mathbf{P}$ then           ▷ simplex touches own points
 5:         if $\exists k, l$ : vertices$_k(s_{B,x}) \in \mathbf{C} \wedge$ vertices$_l(s_{B,x}) \notin \mathbf{C}$) then
 6:             $T \cup \{s_{B,x}\}$   $\mathbf{Q} \cup \{s_{B,x}\}$          ▷ simplex spans both partitions
 7:         else
 8:             if $\exists s \in \mathbf{B}$ : vertices($s$) = vertices($s_{B,x}$) then
 9:                 $T \cup \{s_{B,x}\}$   $\mathbf{Q} \cup \{s_{B,x}\}$      ▷ previously stripped simplex
10: updateNeighbors($T, \mathbf{Q}, \mathbf{C}$)
11: return $T$
```

as border simplex by PE j. To ensure every PE is aware of all of its border simplices, a sparse all-to-all communication within partition $\mathbf{C}'$ is required, yielding the updated set $\mathbf{B} = (\cup_{j \in \mathbf{C}'} \mathbf{B}@j) \cap T$.

Only PEs with nonempty set $\mathbf{B}$ need to participate in the border triangulation. The border triangulation follows the same algorithm as the main triangulation with the reduced PE set $\mathbf{C}_B$. The merging of T and T_B is extended by the additional condition that a simplex $s_B \in T_B$ is only considered for addition to T if at least one of its vertices lies in $\mathbf{P}_i$. The further conditions are the same as in the shared-memory case, as seen in Algorithm 2.4.

The determination of the neighborhood relations of the newly inserted simplices is also identical to the shared-memory case. However, as each PE only possesses a partial view on the triangulation $DT(\mathbf{C})$, not all neighbors of simplex $s \in T$ can be determined by PE i alone. Particularly, if vertices(s) contains only one vertex from $\mathbf{P}_i$, at least one of the neighbors of s will not be stored at PE i. Therefore, each PE keeps track of all the updates it performs. In a sparse all-to-all communication, information about the updates to a simplex s is sent to every PE that contains s in its local triangulation and can then perform the updates locally, as seen in Algorithm 2.5.

Algorithm 2.5 updateNeighbors($T, \mathbf{Q}, \mathbf{C}$): update neighbor relations of simplices in T.

```
Input:  Triangulation $T$, work queue $\mathbf{Q}$, PEs of partition $\mathbf{C}$
Output: updated triangulation $T$
 1: $\mathbf{U} \leftarrow \varnothing$                                          ▷ list of performed updates
 2: parfor $s_x \in \mathbf{Q}$ do
 3:     mark($s_x$)                                       ▷ only process each simplex once
 4:     for $k \in [1..D+1]$ do                              ▷ process all neighbors
 5:         $s_y \leftarrow \text{neighbors}_k(s_x)$
 6:         if $s_y = \infty \lor s_y \notin T$ then                     ▷ $s_y$ not valid neighbor
 7:             $s_y \leftarrow \infty$                                    ▷ update needed, reset
 8:             $C \leftarrow \{s_c : f_k(s_x) = f_k(s_c)\}$             ▷ candidates with same facet hash
 9:             for $s_c \in C$ do
10:                 if $|\text{vertices}(s_x) \cap \text{vertices}(s_c)| = D \land \text{vertices}_k(s_x) \notin s_c$ then
11:                     $s_y \leftarrow s_c$                              ▷ $s_c$ is neighbor of $s_x$
12:                     if $\neg\,\text{marked}(s_c)$ then
13:                         mark($s_c$);   $\mathbf{Q} \cup \{s_c\}$        ▷ also update neighbors of $s_c$
14:                         $\mathbf{U} \cup \{(s_x \quad k \quad s_c)\}$        ▷ keep track of update
15: $\mathbf{U} \leftarrow \text{allGather}(\mathbf{U}, \mathbf{C})$                           ▷ exchange neighbor updates
16: parfor $(s_x \quad k \quad n) \in \mathbf{U}$ do
17:     if $s_x \in T$ then $\text{neighbors}_k(s_x) \leftarrow n$          ▷ apply updates from other PEs
18: return $T$
```

Algorithm 2.6 DelaunayBase($\mathbf{P}, \mathbf{C}$): base case for distributed DT algorithm.

```
Input:  Point subset $\mathbf{P} = \{p_1, ..., p_n\}$, PEs of partition $\mathbf{C}$
Output: Delaunay triangulation $T$ of $\mathbf{P}$
 1: if $|\mathbf{C}| = 1$ then                                              ▷ base case
 2:     return Delaunay($\mathbf{P}$)                         ▷ shared-memory DT algorithm
 3: if PE $i = \min \mathbf{C}$ then
 4:     $\mathbf{P}' \leftarrow \text{gather}(\mathbf{P}, \mathbf{C})$                         ▷ gather points from neighbors
 5:     $T' \leftarrow \text{Delaunay}(\mathbf{P}')$   broadcast($T'$)        ▷ shared-memory DT algorithm
 6: else                                                       ▷ all other PEs
 7:     send($\mathbf{P}$)   receive($T'$)
 8: $T \leftarrow \{s \in T' : |\text{vertices}(s) \cap \mathbf{P}| \geq 1\}$                 ▷ filter simplices
 9: return $T$
```

Distributed Base Case: Algorithm 2.6 details the treatment of the base case in the distributed setting. If there is only PE i left in partition $\mathbf{C}$ the points $\mathbf{P}_i$ are triangulated on PE i using a sequential or shared-memory parallel algorithm. By setting the base case threshold N in Algorithm 2.2 greater than $\max_{j\in\{1,\dots,P\}} |\mathbf{P}_j|$, $|\mathbf{C}|$ will always be one in the first recursive descent and each PE will triangulate its own input points locally. Only in the recursive calls of delaunay(...) for border triangulations can $\mathbf{C}$ contain more than one PE. In that case, the PE in $\mathbf{C}$ with the lowest index is chosen as master and receives the input points of all other PEs in $\mathbf{C}$. The master triangulates the gathered points and broadcasts the simplices among the PEs of the partition. The PEs then discard all simplices with no vertex in their respective input point sets.

2.4.3 Implementation Details

This section highlights some aspects of our implementation of the previously proposed algorithms.[1] While our algorithms are implemented for arbitrary dimension, we have only included base case algorithms for two- and three-dimensional DTs at the moment.

The input points $\mathbf{P} = \{p_1, ..., p_n\}$ with $p_i \in \mathbb{R}^D$ are stored in an array with their D coordinates. A partition of points consists of a list of indices into this array. To ensure globally unique point indices in the distributed setting, each PE i stores the global offset $o_{v,i}$ of its point array; $o_{v,i} = \Sigma_{j<i}|\mathbf{P}_j|$. In addition to the main point array, an auxiliary hash table of points is stored at each PE, that holds the vertices of simplices not entirely contained in $\mathbf{P}_i$ and points received in Algorithm 2.6. In shared memory, the data structure of Fuetterling et al. [FLP14] seems to be applicable to speedup division of the input points. However, due to unavailability of source code its inclusion remains for future work.

Our triangulation data structure is extended from Shewchuk [She96]. Simplices are stored in an array, where each simplex s consists of an ID, the $D+1$ indices of its points in the point array—vertices(s)—and $D+1$ indices to its neighboring simplices—neighbors(s). The vertices of a simplex are sorted by index; neighbors are stored such that neighbor i intersects s at the facet opposite vertex i. In the distributed case, a PE i sets the upper $\log P$ bits of its simplex IDs to i to obtain globally unique identifiers.[2] Our triangulation data structure furthermore stores the indices of the simplices of its convex hull, as these serve as starting point for the border detection algorithm. To allow fast merging of two partial triangulations, simplices are stored in *blocks*. A base case triangulation results in a single block of simplices with consecutive, globally unique IDs. When merging T_1 and T_2 into a combined triangulation T, T stores two pointers to the data blocks of T_1 and T_2, along with their respective minimum and maximum simplex ID in a binary search tree. After k merge steps, this allows for random access to a simplex in $\mathcal{O}(\log k)$ time; scanning is still in $\mathcal{O}(1)$.

The vertex hash function $h_v(\cdot)$ required for the simplex and facet hash function needs to minimize the expected number of collisions while being efficiently computable. In our experiments we found that setting $h_v(v)$ equal to the point index rotated by its lowest byte value to suit our needs. That is,

$$h_v(v) := v \lll \mathrm{lsb}(v).^3$$

Whether our algorithm would benefit from a more sophisticated hash function, i.e., a provable universal hash function, remains for future work. Concurrent hash table operations

[1] Source code available at `https://github.com/dfunke/ParDeTria`.
[2] Simplex ID offset $o_{s,i} = i \cdot 2^{8w-\log P}$, w = machine word size.
[3] $\lll$ bit rotation, lsb least significant byte.

are at the heart of the merging and neighborhood update algorithms. The efficient, growable, concurrent hash table by Maier et al. [MSD16] is used in our implementation. We extended their implementation to multisets for facet hash lookups during neighborhood updates.

Intel's Threading Building Blocks (TBB) library[4] is used for shared-memory parallelization. Particularly, its concurrent work queue is employed in the border simplex detection and neighborhood update algorithms.

Geometric algorithms need to address the limited floating-point precision of current hardware. Our proposed D&C scheme relies on combinatorial computations on hash values except for the detection of the border simplices of two partial triangulations. We use the fast sphere-box overlap test of Larsson et al. [LAL07] to determine if the (hyper)-circumsphere of a given simplex intersects with the bounding box of the opposite partial triangulation. The test does not suffer from floating-point inaccuracies like the ORIENT- and IN-SPHERE-tests required by the base case algorithm [She97].

2.4.4 Evaluation

Batista et al. [Bat+10] propose three input point distributions to evaluate the performance of their DT algorithm: N points distributed uniformly

a) in the unit cube;
b) on the surface of an ellipsoid; and
c) on skewed lines.

Additionally, Lee et al. [LPP01] suggest normally distributed input points around

d) the center of the unit cube; and
e) several points within the unit cube—called "bubbles".

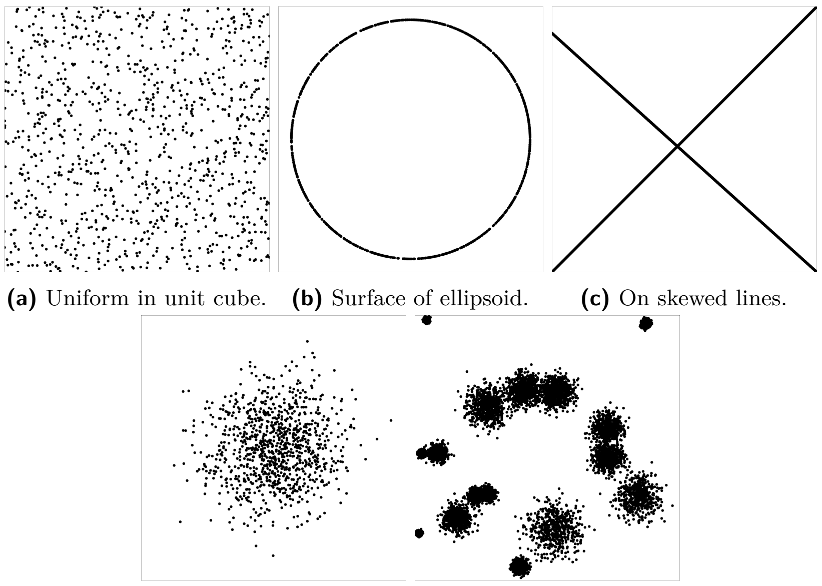
All experiments are performed in three-dimensional space, Figure 2.6 gives two-dimensional examples of the studied input point sets. Points along skewed lines are among the worst-case inputs for DT construction algorithms, as they generate triangulations with a quadratic number of simplices [Bat+10]. The bubble distribution can result in a large border triangulation if the dense parts of the bubbles are cut and poses a challenge for load balancing [LPP01].

We furthermore test our algorithm with two real-world datasets from materials science, where Voronoi analysis is used in simulation studies of liquids, glasses and solids to explore their atomic structure, e.g., the characteristic arrangement of near neighbors of an atom [Stu12]. Amorphous Copper-Zirconium (CuZr) alloys are used as benchmark compound in the field [Wan+04]; we evaluate two 50/50 Copper/Zirconium system consisting of four million and 100 million atoms.

Table 2.3 gives an overview of all evaluated point sets, along with the size of their resulting triangulation and required runtime of our algorithm. It shows that the ellipsoid and particularly the skewed lines distribution result in a large number of simplices and require a long time to compute, as further discussed in Section 2.4.4.1.

The shared-memory algorithm was evaluated on a machine with dual Intel Xeon E5-2683 16-core processors and 512 GiB of main memory. We use the sequential Delaunay triangulation algorithm of CGAL 4.7 as base case in Algorithm 2.1 and compare our implementation to the parallel DT construction algorithm of CGAL [HS15]. In both cases, CGAL is configured to

[4] `https://www.threadingbuildingblocks.org/` (accessed 08-12-2023)



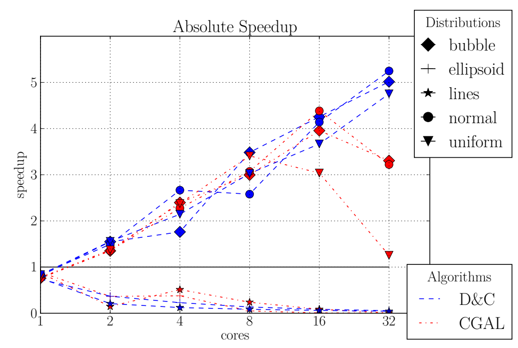
(a) Uniform in unit cube. **(b)** Surface of ellipsoid. **(c)** On skewed lines.

(d) Normal around center. **(e)** Bubbles.

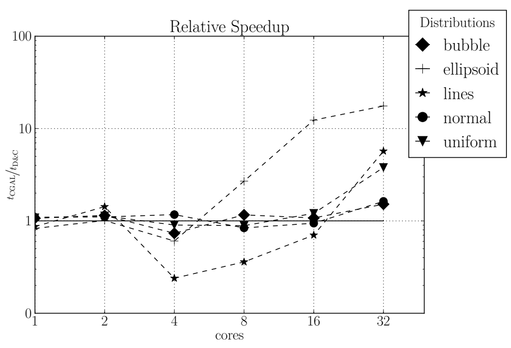
Figure 2.6 Two-dimensional examples of our synthetic input distributions for $n = 1000$ points. Note that lines cannot be skewed in two dimensions.

Table 2.3 Evaluated point sets and their resulting triangulations. Shared-memory runtimes are reported for 32 cores, distributed-memory runtimes for 2048 cores with 4 cores per MPI process.

Distribution	Points	Simplices	Runtime
	shared memory		
uniform	50 000 000	360 542 380	64 s
normal	50 000 000	361 877 812	83 s
bubble	50 000 000	361 638 812	70 s
ellipsoid	500 000	84 408 498	169 s
lines	10 000	122 396 140	292 s
CuZr	4 000 000	28 927 267	8 s
CuZr	100 000 000	634 926 984	148 s
	distributed memory		
uniform	2 048 000 000	22 112 081 080	92 s
normal	204 800 000	5 861 711 093	50 s



(a) Absolute speedup over sequential CGAL.



(b) Relative speedup over parallel CGAL.

Figure 2.7 Speedup of our shared-memory D&C algorithm and CGAL's parallel DT implementation for various point distributions.

use exact predicates for ORIENT and IN-SPHERE computations.[5] In preliminary experiments, the cyclic choice of splitting dimension proved to be the best partitioning scheme and was therefore used for all experiments.

The distributed-memory experiments were conducted on InstitutsCluster II at the Steinbuch Centre for Computing at Karlsruhe Institute of Technology. The cluster contains 480 compute nodes with dual Intel Xeon E5-2670 8-core processors and 64 GiB of main memory, connected by an InfiniBand 4X QDR interconnect. A single job may use up to 128 nodes ($\equiv$ 2048 cores). OpenMPI version 1.8.6 was used as message passing library.

2.4.4.1 Shared-memory Algorithm

Figure 2.7 shows the performance of our algorithm in comparison to CGAL's sequential and parallel DT algorithms for the aforementioned input distributions. The uniform, normal and bubble distribution show good scaling behavior. The bubble distribution has few points near the border of a partition and thus a low number of vertices in the border triangulation. CGAL's parallel incremental insertion encounters low congestion in its locking, since the vertices of one bubble are mostly handled by a single thread due to spatial sorting [Bat+10]. Uniformly distributed points have larger—but compact and even—border triangulations, resulting in good load balancing between the partitions. In contrast, normally distributed points result in larger border triangulations around the center, yet they profit from small cuts in the outer regions. When using only a single socket of our test machine, our algorithm performs on par with CGAL's implementation. For multiple sockets, however, we clearly outperform CGAL. Our algorithm adapts well to the NUMA setting, as—except for the final merge step—the entire algorithm operates exclusively on socket-local data. Contrarily, CGAL's incremental insertion algorithm requires continuous communication between threads of the two sockets.

The ellipsoid and skewed line distributions are specifically tailored to be hard inputs for both implementations. The former is hard due to its large convex hull, the latter due to the quadratic number of simplices in the input size. Both parallel implementations fall below the throughput of the sequential reference. Our implementation's performance degrades less than CGAL for the ellipsoid, as only the simplices of the convex hull whose circumspheres

[5] `CGAL::Exact_predicates_inexact_constructions_kernel`

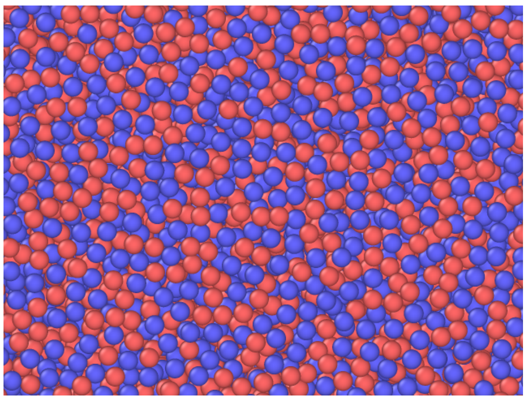


Figure 2.8 Structure of an amorphous CuZr alloy. Copper atoms are depicted in red, Zirconium atoms in blue.

intersect the splitting plane contribute to the border triangulation, while CGAL suffers from high congestion on the inner simplices of the ellipsoid. Congestion on the inner simplices also leads to CGAL's bad performance for skewed lines. Our implementation suffers even more, due to almost all simplices intersecting the splitting plane for at least one cutting dimension, resulting in large border triangulations.

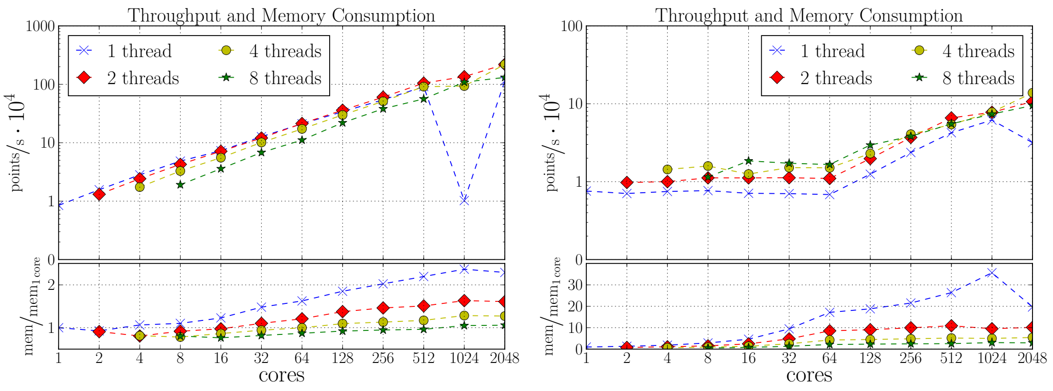
Figure 2.8 shows the fairly regular structure of the atoms of a CuZr alloy. This results in compact and even cuts between the partitions of the triangulation similar to uniformly distributed points. The scaling behavior is therefore also comparable to this point distribution.

2.4.4.2 Distributed-memory Algorithm

Figure 2.9 shows the weak scaling behavior of our distributed-memory implementation. In the experiment each core processes one million input points for the uniform distribution and 100k points for the normal distribution, due to memory limitations at the central nodes described below. We show the behavior for different configurations of hybrid parallelization, ranging from one thread per MPI process—i.e., one process per core with sequential base case—to eight cores per MPI process—i.e., one process per socket with our shared-memory parallel D&C algorithm as base case in Algorithm 2.6. We would have expected one process per socket to yield the best results, as this configuration is NUMA aware and requires the least inter-process communication. Nevertheless, for uniformly distributed points, two cores per process show the best performance with a speedup of ≈ 260 for 2048 cores.[6] For the normal distribution, we attain a more modest speedup of 18 for 2048 cores due to the lack of load balancing in our current implementation. PEs close to the center of the distribution have a much higher workload than others, preventing larger speedup gains. We address this shortcoming in Section 2.5. If the input points are not pre-partitioned, we observe a runtime overhead for the additional all-to-all communication of 10 % to 15 % on average.

With increasing number of PEs and input size the recursion depth increases. Thus, more border triangulations are required to produce the global triangulation. Furthermore, each PE needs to store more simplices that are only partially contained in its original input point set. This increases memory consumption per PE. In the uniformly distributed setting, our measurements show that while the total number of processed points grows by three orders

[6] We attribute the outlier for 1024 cores to some other activity in the cluster.



(a) Uniform distribution. **(b)** Normal distribution.

Figure 2.9 Throughput and memory overhead of our distributed-memory algorithm in a weak scaling experiment.

of magnitude going from one to 2048 PEs, memory consumption per core only increases by a factor of 2.2 and about 9 % of the input points need to be re-triangulated in a border triangulation. For the normal distribution, the memory increase exceeds a factor of 30. This is due to the large number of points processed by the central PEs. Furthermore, PEs close to the center also have to store many simplices only partially contained in their original point set. Again, load balancing would mitigate this issue. For both distributions, the benefit of hybrid parallelization is apparent, as more threads per MPI process result in fewer processes, leading to reduced recursion depth in the distributed algorithm.

2.4.5 Conclusions

In this section we presented a novel divide-and-conquer algorithm for computing the Delaunay triangulation in arbitrary dimension, that reduces the merging of two subproblems to re-triangulating a small subset of their vertices and using efficient hash table operations to combine the three triangulations into one. All steps of the merging are parallelized. We are able to perform on par with or better than CGAL's parallel three-dimensional DT implementation in shared memory and show good scalability for our approach in distributed memory up to 2048 cores and two billion input points.

Our experiments show that more advanced load balancing and work division strategies, aiming at smaller border sizes, are required to yield a more robust algorithm, capable of processing large realistic inputs from a variety of fields. We will develop such strategies in the next section.

Many real-world inputs require periodic boundary conditions, which need to be handled efficiently by our algorithm and remain for future work. The DT construction algorithm presented in Section 2.6 is capable of handling periodic boundary conditions, however it is not designed for general point sets but only for uniformly distributed points as produced by our network generator.

2.5 Sample-based Load Balancing Strategies

As seen in the previous section, the divide-step of our D&C DT algorithm needs to address a twofold sensitivity to the point distribution: the partitions need to be approximately equal-sized for good load balancing, while the number of *border vertices* needs to be minimized for fast merging. This requires partitions that have many internal Delaunay edges but only few external ones, i.e., a graph partitioning of the DT graph. In this section we propose a novel divide-step that approximates this graph partitioning by triangulating and partitioning a small sample of the input points, and divides the original input point set based upon it. This results in more complexly shaped partitions than simple axis-aligned bounding boxes, for which we present fast intersection tests. We compare our new divide-step against the partitioning schemes presented in Section 2.4.1.1.

2.5.1 Sample-based Partitioning

The underlying idea of our partitioning scheme is derived from sample sort [FM70]: gain insight into the input distribution from a (small) sample of the input. This sample is then used to partition the input points into k partitions. For this we need to generalize the original algorithm, Algorithm 2.1, to work with an arbitrary number of partitions. Algorithm 2.7 shows the adapted algorithm, with changes highlighted by red line numbers. The partitioning procedure in Line 3 is described in Algorithm 2.8. A sample $\mathbf{P}_S$ of $\eta(n)$ points is taken from the input point set $\mathbf{P}$ of size n and triangulated to obtain $DT(\mathbf{P}_S)$. A similar approach can be found in Delaunay hierarchies [Dev02], where the sample triangulation is used to speed up point location queries.

We transform the sample DT into a graph $G = (V, E, \omega)$, with V being equal to the sample point set $\mathbf{P}_S$ and E containing all edges of $DT(\mathbf{P}_S)$. The resulting graph is then partitioned into k blocks using a graph partitioning algorithm.

The choice of the weight function ω influences the quality of the resulting partitioning. As mentioned above, the D&C algorithm is sensitive to the balance of the blocks as well as the size of the border triangulation. The former is ensured by the imbalance parameter ϵ of the graph partitioning, which guarantees that for all partitions i: $|V_i| \leq (1+\epsilon)\lceil\frac{|V|}{k}\rceil$. The latter needs to be addressed by the edge weight function ω of the graph. In order to minimize the size of the border triangulation, dense regions of the input points should not be cut by the partitioning. Sparse regions of the input points result in long Delaunay edges in the sample triangulation. As graph partitioning tries to minimize the weight of the cut edges, edge weights need to be inversely related to the Euclidean length of the edge. Table 2.4 provides an overview of the edge weight functions considered, which are evaluated in Section 2.5.2.1.

Table 2.4 Possible choices for the edge weight ω, with $d(v,w) = \frac{||v-w||}{d^*}$ denoting the normalized Euclidean distance of points v and w, with d^* being the length of the maximum diagonal of the input space.

Weight	$\omega(e = (v,w))$
constant	1
inverse	$\frac{1}{d(v,w)}$
logarithmic	$-\log d(v,w)$
linear	$1 - d(v,w)$

Algorithm 2.7 Delaunay($\mathbf{P}$): shared-memory parallel D&C algorithm adapted to an arbitrary number of partitions. Red line numbers indicate differences to the original Algorithm 2.1.

```
Input:   Points $\mathbf{P} = \{p_1, ..., p_n\}$ with $p_i \in \mathbb{R}^D$
Output:  Delaunay triangulation $T(\mathbf{P})$
 1: if $n < N$ then
 2:     return $sequentialDelaunay(\mathbf{P})$                                ▷ base case
 3: $\left(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k\right) \leftarrow \text{partitionPoints}(\mathbf{P}, k)$        ▷ partition points into $k$ partitions
 4: $\left(T_1 \quad \dots \quad T_k\right) \leftarrow \left(\text{Delaunay}(\mathbf{P}_1) \quad \dots \quad \text{Delaunay}(\mathbf{P}_k)\right)$   ▷ parallel triangulation
Border triangulation:
 5: $\mathbf{B} \leftarrow \varnothing$                                          ▷ empty set of border simplices
 6: $\mathbf{Q} \leftarrow \bigcup_{1 \leq i \leq k} \text{convexHull}(T_i)$                  ▷ initialize work queue with convex hull
 7: parfor $s_{i,x} \in \mathbf{Q}$ do                          ▷ simplex originating from triangulation $T_i$
 8:     mark($s_{i,x}$)                                          ▷ process each simplex only once
 9:     if intersects (circumsphere($s_{i,x}$), $T_j$), with $i \neq j$ then
10:         $\mathbf{B} \cup= \{s_{i,x}\}$     ▷ circumsphere intersects other partition $\Rightarrow s_{i,x}$ is border simplex
11:         for $s_{i,y} \in \text{neighbors}(s_{i,x}) \wedge \neg\,\text{marked}(s_{i,y})$ do         ▷ process all neighbors
12:             mark($s_{i,y}$);   $\mathbf{Q} \cup= \{s_{i,y}\}$
13: $T_B \leftarrow \text{Delaunay}(\text{vertices}(\mathbf{B}))$                   ▷ triangulate points of border simplices
Merging:
14: $T \leftarrow \left(\bigcup_{1 \leq i \leq k} T_k\right) \setminus \mathbf{B}$;   $\mathbf{Q} \leftarrow \varnothing$           ▷ merge partial triangulations, strip border
15: parfor $s_b \in T_B$ do                               ▷ merge simplices from border triangulation
16:     if vertices($s_b$) $\not\subset \mathbf{P}_i \quad \forall 1 \leq i \leq k$ then
17:         $T \cup= \{s_b\}$;   $\mathbf{Q} \cup= \{s_b\}$                        ▷ $s_b$ spans multiple partitions
18:     else
19:         if $\exists s \in \mathbf{B}$ : vertices($s$) = vertices($s_b$) then
20:             $T \cup= \{s_b\}$;   $\mathbf{Q} \cup= \{s_b\}$                    ▷ $s_b$ replaces border simplex
Neighborhood update:
21: parfor $s_x \in \mathbf{Q}$ do                           ▷ update neighbors of inserted simplices
22:     mark($s_x$)                                          ▷ only process each simplex once
23:     for $d \in [1..D+1]$ do
24:         if $\text{neighbors}_d(s_x) \notin T$ then              ▷ neighbor not in triangulation anymore
25:             $C \leftarrow \{s_c : f_d(s_x) = f_d(s_c)\}$                 ▷ candidates with same facet hash
26:             for $s_c \in C$ do
27:                 if $|\text{vertices}(s_x) \cap \text{vertices}(s_c)| = D$ then
28:                     $\text{neighbors}_d(s_x) \leftarrow s_c$                       ▷ $s_c$ is neighbor of $s_x$
29:                     if $\neg\,\text{marked}(s_c)$ then
30:                         mark($s_c$);   $\mathbf{Q} \cup= \{s_c\}$          ▷ process $s_c$ if not already marked
31: return $T$
```

Algorithm 2.8 partitionPoints$(\mathbf{P}, k)$: partition input into k partitions.

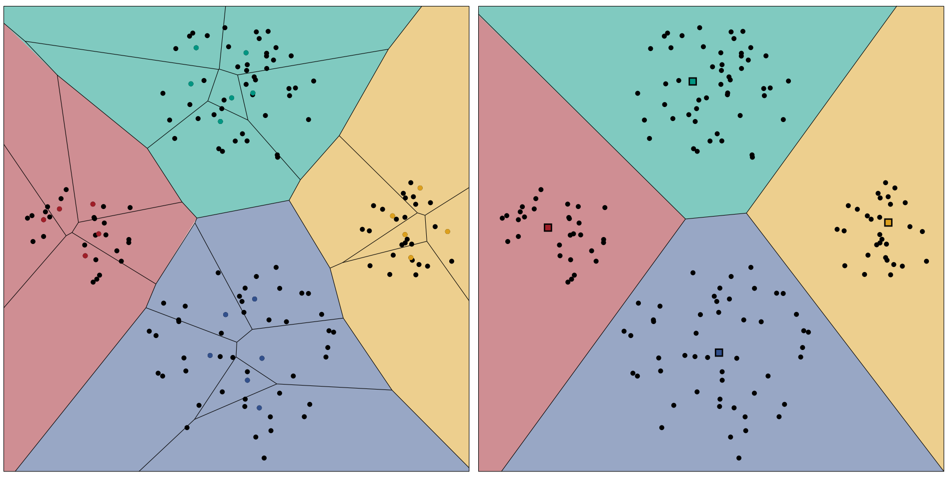
```
Input:  Points $\mathbf{P} = \{p_1, ..., p_n\}$ with $p_i \in \mathbb{R}^D$, number of partitions $k$
Output: partitioning $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k)$
1: $\mathbf{P}_S \leftarrow$ choose $\eta(n)$ from $\mathbf{P}$ uniformly at random        ▷ $\eta(n)$ sample size
2: $T \leftarrow$ Delaunay$(\mathbf{P}_S)$
3: $G = (V, E, \omega)$ with $V = \mathbf{P}_S$, $E = T$ and weight function $\omega$
4: $(V_1 \quad \dots \quad V_k) \leftarrow$ partition$(G)$        ▷ partition graph
5: $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k) \leftarrow$ extendPartitioning$(\mathbf{P}, \mathbf{P}_S, (V_1 \quad \dots \quad V_k))$
6: return $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k)$
```

Algorithm 2.9 extendPartitioning$(\mathbf{P}, \mathbf{P}_S, (V_1 \quad \dots \quad V_k))$: nearest sample point assignment.

```
Input:  Points $\mathbf{P}$, sample points $\mathbf{P}_S$, sample partitioning $(V_1 \quad \dots \quad V_k)$
Output: partitioning $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k)$
1: $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k) \leftarrow (\varnothing \quad \dots \quad \varnothing)$
2: parfor $p \in \mathbf{P}$ do
3:     $v_n \leftarrow \arg\min_{v \in \mathbf{P}_S} ||p - v||$        ▷ find the nearest sample point to $p$
4:     $\mathbf{P}_i \cup= \{p\}$ with $i \in [1..k] : v_n \in V_i$        ▷ assign $p$ to $v_n$'s partition
5: return $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k)$
```

Algorithm 2.10 extendPartitioning$(\mathbf{P}, \mathbf{P}_S, (V_1 \quad \dots \quad V_k))$: nearest partition center assignment.

```
Input:  Points $\mathbf{P}$, sample points $\mathbf{P}_S$, sample partitioning $(V_1 \quad \dots \quad V_k)$
Output: partitioning $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k)$
1: $\mathbf{C} \leftarrow \varnothing$        ▷ initialize set of centroids
2: parfor $i \in [1..k]$ do
3:     $\mathbf{C} \cup= \left\{ \frac{\sum_{p \in V_i} p}{|V_i|} \right\}$        ▷ compute centroid of $V_i$
4: $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k) \leftarrow (\varnothing \quad \dots \quad \varnothing)$
5: parfor $p \in \mathbf{P}$ do
6:     $c_i \leftarrow \arg\min_{c \in \mathbf{C}} ||p - c||$        ▷ find the nearest centroid to $p$
7:     $\mathbf{P}_i \cup= \{p\}$        ▷ assign $p$ to $c_i$'s partition
8: return $(\mathbf{P}_1 \quad \dots \quad \mathbf{P}_k)$
```



(a) Nearest sample point assignment. **(b)** Nearest partition center assignment.

Figure 2.10 Extension of the partitioning of the sample points to the entire input point set. Sample points are colored according to their partition, all other input points are black. The partition centroids are marked by colored squares.

Given the partitions of the sample vertices $(V_1 \dots V_k)$, the partitioning needs to be extended to encompass all input points. We propose two different variants to extend the partitioning: **1.** nearest sample point assignment (NSA) and **2.** nearest partition center assignment (NCA).

1. Nearest Sample Point Assignment. Each input point is assigned to the partition of its nearest sample point, refer to Figure 2.10a and Algorithm 2.9. Consider the dual of the Delaunay triangulation of the sample point set $DT(\mathbf{P}_S)$—its Voronoi diagram $VD(\mathbf{P}_S)$. Each point $p_{S,i}$ of the sample point set $\mathbf{P}_S$ is assigned to a partition V_j, with $j \in [1..k]$, by the graph partitioning of $DT(\mathbf{P}_S)$. The Voronoi cell of $p_{S,i}$ in $VD(\mathbf{P}_S)$ defines the subspace R_i of $\mathbb{R}^D$ where any point $p \in R_i$ is closer to $p_{S,i}$ than to any other sample point. The union of the Voronoi cells of all points in partition V_j therefore defines the subspace of the input space associated with partition j. Thus, in order to extend the partitioning to the entire input point set, each point $p \in \mathbf{P}$ is assigned to the partition of the Voronoi cell in $VD(\mathbf{P}_S)$ containing it, i.e., the partition of its nearest sample point.

2. Nearest Partition Center Assignment. As can be seen in Figure 2.10a, the borders of a partition can be irregular and jagged using NSA. In order to obtain a more regular partitioning, we propose to assign each input point to the partition of its nearest partition center, refer to Figure 2.10b and Algorithm 2.10. The partition center C_i is defined as the centroid of all points assigned to partition V_i, given by

$$C_i = \frac{\sum_{p \in V_i} p}{|V_i|}, \text{ with } p \in \mathbb{R}^D.$$

Each partition center C_i defines a Voronoi cell R_i in $\mathbb{R}^D$, where all input points $p \in R_i$ are closer to C_i than to any other partition center. This leads to smoother borders between partitions as can be seen in Figure 2.10b.

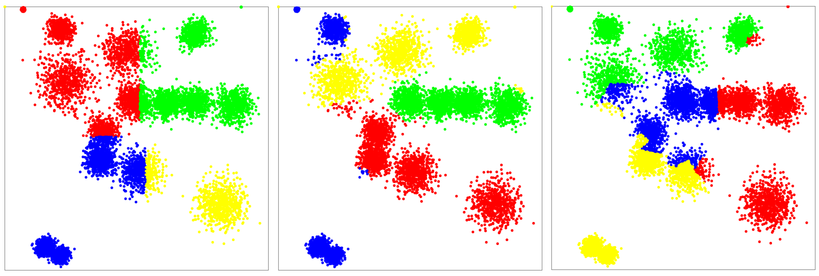

(a) Cyclic median. **(b)** Direct k-way. **(c)** Recursive bisection.

Figure 2.11 Example of a two-dimensional partitioning with four partitions for 10 000 points and a sample size of 1000. Cyclic median splitting corresponds to the original partitioning scheme presented in Section 2.4.1.1. Direct k-way and recursive bisection use our new sample-based partitioning strategy with NSA.

Parallelization. All steps in Algorithm 2.8 can be efficiently parallelized. Sanders et al. [San+18] present an efficient parallel random sampling algorithm. The triangulation of the sample point set $\mathbf{P}_S$ could be computed in parallel using our DT algorithm recursively. However, as the sample is small, a fast sequential algorithm is typically more efficient. Graph conversion is trivially done in parallel and Akhremtsev et al. [ASS18] present a state-of-the-art parallel graph partitioning algorithm. The parallelization of the assignment of input points to their respective partitions is explicitly given in Algorithms 2.9 and 2.10.

2.5.1.1 Recursive Bisection & Direct k-way Partitioning

Two possible strategies exist to obtain k partitions from a graph: direct k-way partitioning and recursive bisection. For the latter, the graph is recursively partitioned into $k' = 2$ partitions $\log k$ times. In the graph partitioning community, Simon and Teng [ST97] prove that recursive bisection can lead to arbitrarily bad partitions and Kernighan and Lin [KL70] confirm the superiority of direct k-way partitioning experimentally. However, recursive bisection is still widely—and successfully—used in practice (e.g., in METIS [KK98] and for initial partitioning in KaHIP [SS13]). Other problem domains also apply recursive bisection successfully. In hypergraph partitioning, it can lead to better partitionings in the presence of large hyperedges, i.e., edges with many vertices [Akh+17]. We therefore consider both strategies to obtain k partitions for our DT algorithm.

The partitioning schemes described in Section 2.4.1.1 can be seen as recursive bisection: the input is recursively split along the median. The splitting dimension is chosen in a cyclic fashion, similar to k-D trees. Figure 2.11a shows an example.

Similarly, Algorithm 2.8 can be applied $\log k$ times, at each step i drawing a new sample point set $\mathbf{P}_{S,i}$, triangulating and partitioning $\mathbf{P}_{S,i}$, and assigning the remaining input points to their respective partition. As in the original scheme, this leads to $k - 1$ merge steps, entailing $k - 1$ border triangulations. In the sample-based approach however, the partitioning avoids cutting dense regions of the input, which would otherwise lead to large and expensive border triangulations; refer to Figure 2.11c.

Using direct k-way partitioning, only one partitioning and one merge step is required with our generalized algorithm (Algorithm 2.7). The single border point set will be larger, with points spread throughout the entire input area. This, however, allows for efficient parallelization of the border triangulation step using our DT algorithm recursively. Figure 2.11b depicts an example partitioning.

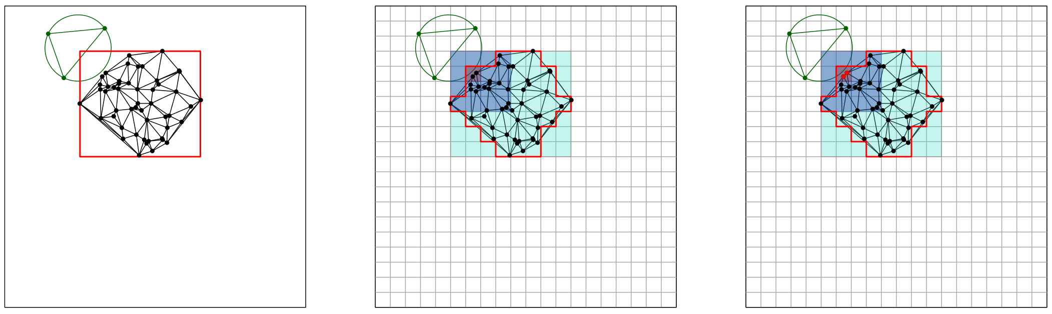

(a) Bounding box. **(b)** Grid-based. **(c)** Exact.

Figure 2.12 Partition boundary determination strategies. The path through the AABB tree to test for intersection with the circle in the upper left is marked by the colored squares. The tested points for the exact strategy are highlighted in red.

For a fair comparison, we also implemented a variant of the original cyclic partitioning scheme, where all leaf nodes of the recursive bisection tree are merged in a single k-way merge step. This allows us to determine whether any runtime gains are due to the k-way merging or due to our more sophisticated data-sensitive partitioning.

2.5.1.2 Geometric Primitives

Our D&C algorithm mostly relies on combinatorial computations on hash values except for the base case computations and the detection of the border simplices. The original partitioning schemes always result in partitions defined by axis-aligned bounding boxes. Therefore, the test whether the circumhypersphere of a simplex intersects another partition (Line 9 in Algorithm 2.7) can be performed using the fast box-sphere overlap test of Larsson et al. [LAL07]. However, using the more advanced partitioning algorithms presented in this section, this is no longer true. Therefore, the geometric primitives to determine the border simplices need to be adapted to the more complexly shaped partitions. The primitives need to balance the computational cost of the intersection test itself with the associated cost for including non-essential points in the border triangulation. We propose three different intersection tests with varying accuracies and computational demands.

Bounding Box Intersection Test. A crude approximation uses the bounding box of each partition and the fast intersection test of Larsson et al. [LAL07] to determine the simplices that belong to the border of a partition. While computationally cheap, the bounding box can overestimate the extent of a partition. Figure 2.12a provides an example.

Grid-based Intersection Test. To improve accuracy while still keeping the determination of the border simplices geometrically simple and computationally cheap, we use a uniform grid combined with an AABB tree [vdBer97]. For each partition $\mathbf{P}_k$ it is determined which cells of the uniform grid $\mathcal{G}$ are occupied by points from that partition, i.e., $\mathcal{C}_k = \{c \in \mathcal{G} : \exists p \in \mathbf{P}_k : p \in c\}$. To accelerate the intersection tests we build an AABB tree on top of each set $\mathcal{C}_k$, depicted in Figure 2.12b. The AABB tree is built once for every partition k and contains the occupied grid cells $\mathcal{C}_k$ as leaves and recursively more coarse-grained bounding boxes as inner nodes. The root node of the tree corresponds to the bounding box of the entire partition, as used in the bounding box intersection test. The grid-based intersection test allows for a more accurate test whether a given simplex s of partition i intersects with partition j using $\log |\mathcal{C}_j|$ box-sphere intersection tests [LAL07].

Exact Intersection Test. In order to only add the absolutely necessary points to the border triangulation an even more computationally expensive test is required. For a given simplex s of partition i we use the AABB intersection test from the previous approach to determine the set $\mathcal{C}'_j \subseteq \mathcal{C}_j$ of cells intersected by the circumhypersphere of s in partition j. For all points contained in these cells an adaptive precision IN-SPHERE-test [She97] is performed to determine whether s violates the Delaunay property and thus its vertices need to be added to the border triangulation.

2.5.1.3 Implementation Notes

We integrated the sample-based divide-step into our original implementation of the D&C algorithm, which is available as open source.[7] We use KaHIP [SS13] and its parallel version [ASS18] as graph partitioning tool. The triangulation of the sample point set is computed sequentially using CGAL [HS15] with exact predicates.[8] The closest sample point for a given input point, e.g., in Line 4 of Algorithm 2.9, can be found via the Voronoi diagram of the sample triangulation. However, using the lightweight k-D tree implementation *nanoflann*[9] proved to be more efficient.

2.5.2 Evaluation

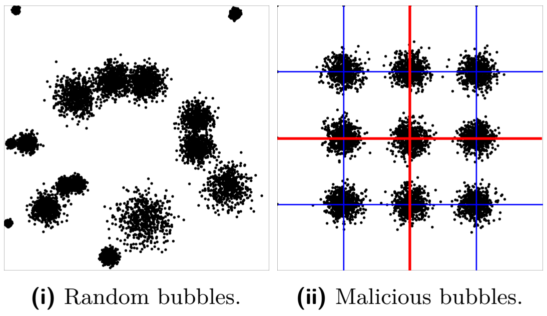
We use the same input point distributions as in Section 2.4.4 to evaluate our sample-based divide-step. Recall that we use n points distributed uniformly

a) in the unit cube;
b) on the surface of an ellipsoid; and
c) on skewed lines [Bat+10];

[7] `https://github.com/dfunke/ParDeTria`
[8] `CGAL::Exact_predicates_inexact_constructions_kernel`
[9] `https://github.com/jlblancoc/nanoflann` (accessed 08-12-2023)

Table 2.5 Input point sets and their resulting triangulations. Running times are reported for $k = t = 16$, parallel KaHIP, $\eta(n) = \sqrt{n}$, grid-based intersection test with $c_\mathcal{G} = 1$ and logarithmic edge weights.

Dataset	Points	Simplices	$\frac{\text{simplices}}{\text{point}}$	Runtime
uniform	50 000 000	386 662 755	7.73	164.6 s
normal	50 000 000	390 705 843	7.81	162.6 s
ellipsoid	500 000	23 725 276	4.74	88.6 s
lines	10 000	71 540 362	7154.04	213.3 s
bubbles	50 000 000	340 201 778	6.80	65.9 s
malicious	50 000 000	143 053 859	2.86	63.9 s
Gaia DR2	50 000 000	359 151 427	7.18	206.9 s



(i) Random bubbles. **(ii)** Malicious bubbles.

Figure 2.13 Two-dimensional examples of the two variants of bubble distribution for $n = 1000$ points. For the malicious bubbles distribution, the cuts of the cyclic median partitioner are shown.

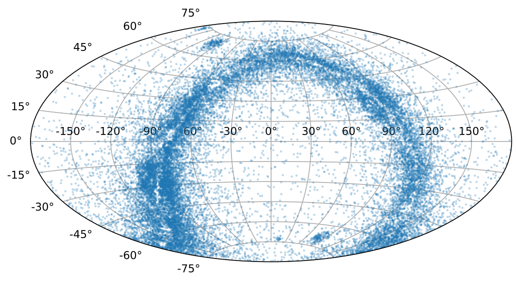


Figure 2.14 Aitoff projection of a random sample of 25 000 sources from the Gaia DR2 dataset.

as well as n normally distributed input points around

d) the center of the unit cube; and
e) several “bubble” centers within the unit cube [LPP01].

As illustrated in Figure 2.13, we study two variants of distribution e) with the bubble centers:

i) distributed uniformly at random in the unit cube;
ii) along the axes of the cycle partitioner cuts—called “malicious” distribution.

The malicious bubble distribution is designed as worst-case input for the cyclic median splitting scheme, as the dense bubble centers have to be cut on every level of the partitioning.

Voronoi tessellations are used in astronomy to analyze voids in galactic surveys in order to understand the factors that influence the expansion of the universe [Sut+15]. We therefore test our algorithm with the Gaia DR2 catalog [Gai18] that contains celestial positions and the apparent brightness for approximately 1.7 billion stars. Additionally, for 1.3 billion of those stars, parallaxes and proper motions are available, enabling the computation of three-dimensional coordinates. As Figure 2.14 shows, the data exhibits clear structure, which can be exploited by our partitioning strategy. We use a random sample of the stars to evaluate our algorithm. All experiments are performed in three-dimensional space ($D = 3$). Table 2.5 gives an overview of all input point sets, along with the size of their resulting triangulation.

The algorithm was evaluated on a machine with dual Intel Xeon E5-2683 16-core processors and 512 GiB of main memory. The machine is running Ubuntu 18.04, with GCC version 7.2 and CGAL version 4.11.

Table 2.6 Parameters of our algorithm configuration and conducted experiments.

Parameter	Values
	Algorithmic parameters
sample size $\eta(n)$	$1\,\%$, $2\,\%$, $\log n$, $\sqrt{n}$
KaHIP configuration	STRONG, ECO, FAST, PARALLEL
edge weight $\omega(e)$	constant, inverse, log, linear$^{\dagger)}$
geometric primitive	bbox, exact, grid with cell sizes $c_{\mathcal{G}} = [\frac{1}{2}, 1, 2]$
	Experimental parameters
partitions k	$1, 2, 4, \ldots, 64$
threads t	$t = k$
points n	$[1, 5, 10, 25, 50] \cdot 10^{6\,\ddagger)}$
distribution	see Table 2.5

$^{\dagger)}$ see Table 2.4 for details $^{\ddagger)}$ unless otherwise stated in Table 2.5

2.5.2.1 Parameter Studies

Table 2.6 lists the configuration parameters of our algorithm and the parameter choices of our conducted experiments. In the following we examine the configuration parameters and determine robust choices for all inputs. The parameter choice influences the quality of the partitioning with respect to partition size deviation and number of points in the border triangulation. As inferior partitioning quality will result in higher execution time, we use it as indicator for our parameter tuning. Even though choices for the parameters are correlated, we present each parameter individually for clarity. We use the uniform, normal, ellipsoid and random bubble distribution for our parameter tuning and compare against the originally proposed cyclic median partitioning scheme for reference. In all plots, k-way partitioning schemes are pictured on the left and recursive-bisection ones on the right.

2.5.2.2 Sample Size

The main goal of our divide-step is to approximate a good partitioning of the final triangulation of $DT(\mathbf{P})$. Clearly, a larger sample size $\eta(n)$ yields a better approximation at the cost of an increased runtime for the sample triangulation. On the other hand, a higher partitioning quality results in better load balancing between partitions and smaller border triangulations. Figure 2.15 shows the total triangulation time for various choices of $\eta(n)$ for a fixed choice of edge weight and KaHIP configuration. The runtime of our k-way strategies shows little dependence on the sample size, as only a single sample triangulation is performed. For recursive bisection the higher runtime for larger sample triangulations clearly outweighs any benefit gained from a better partitioning. We therefore choose $\eta(n) = \sqrt{n}$ as default for all subsequent experiments, which is a robust choice for all tested data sets and algorithms.

2.5.2.3 Partitioner Configuration

Numerous configuration parameters balance quality and runtime in graph partitioning [SS13]. KaHIP defines several presets of its parameters, each providing a good trade-off for a given runtime or quality requirement; these are, with increasing focus on runtime: STRONG, ECO and FAST [SS11]. Additionally, a set of parameters specifically tuned for social and web graphs

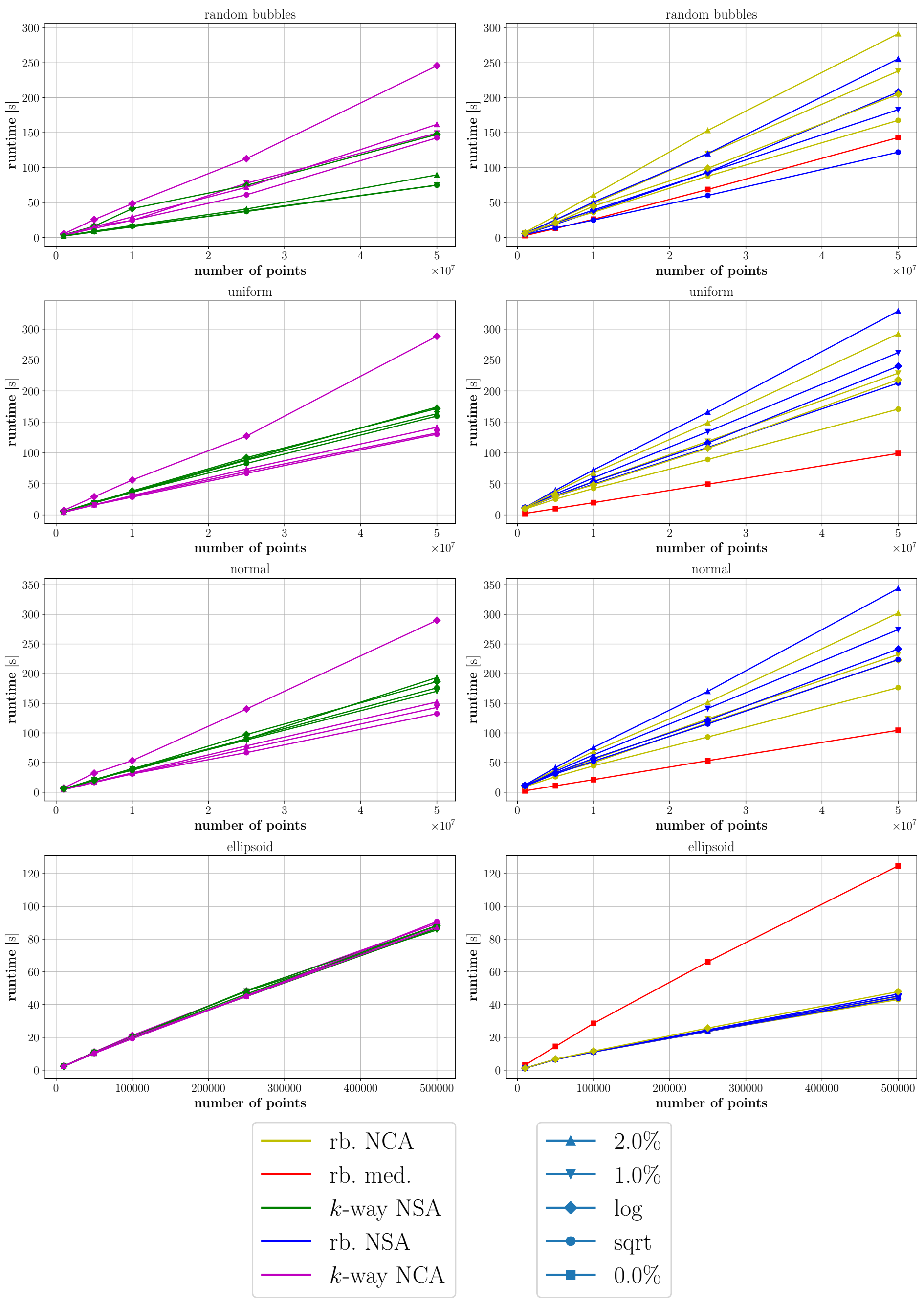


Figure 2.15 Sample size experiments with $k = t = 16$, logarithmic edge weights, grid-based intersection test with $c_{\mathcal{G}} = 1$ and parallel KaHIP.

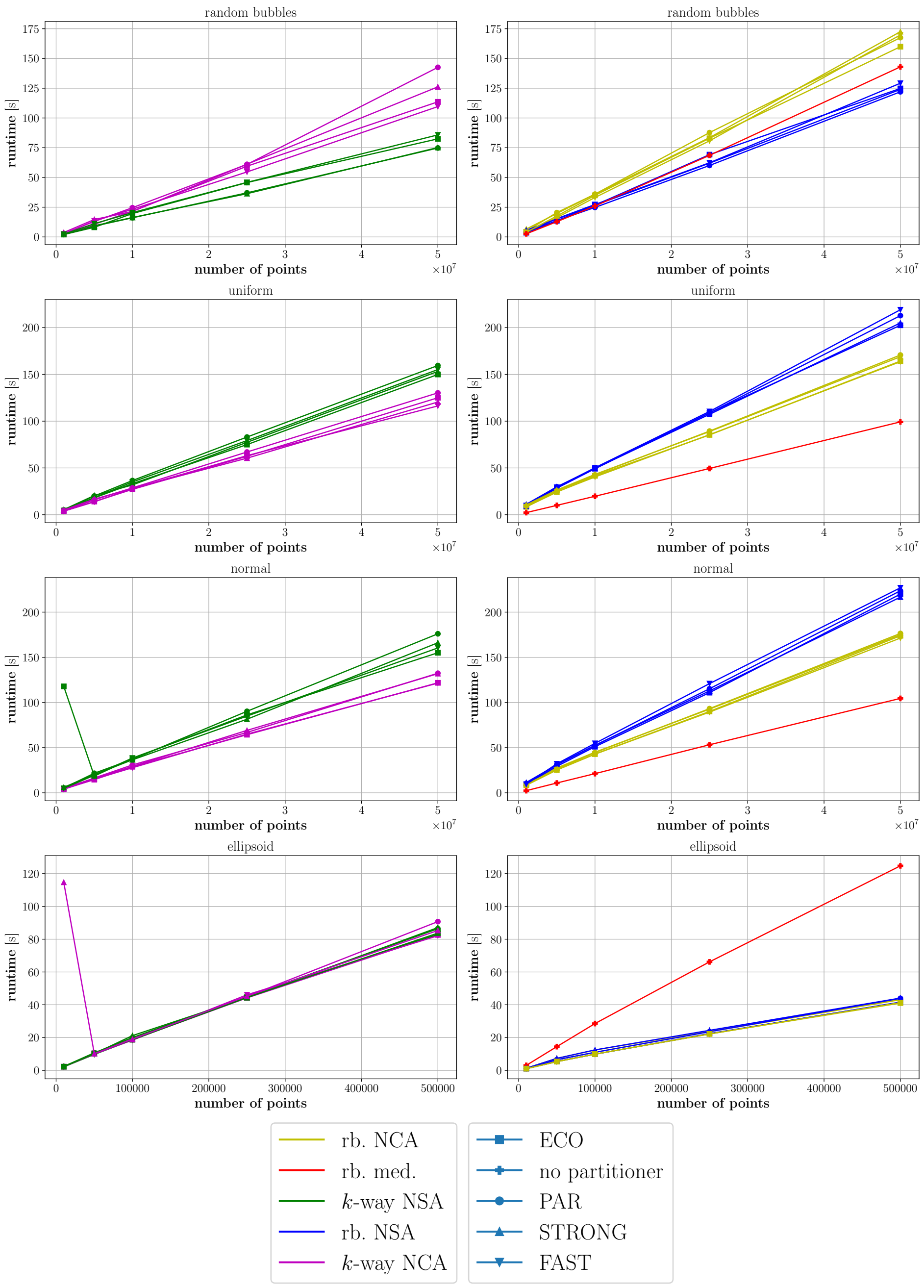


Figure 2.16 KaHIP configuration experiments with $k = t = 16$, logarithmic edge weights, grid-based intersection test with $c_{\mathcal{G}} = 1$ and $\eta(n) = \sqrt{n}$.

is provided. The shared-memory parallel version of KaHIP builds upon these configuration presets and extends them with parallel algorithms. The configuration identified as PARALLEL in our experiments corresponds to FASTSOCIALMULTITRY_PARALLEL in [ASS18]. In all experiments, we set the imbalance parameter for KaHIP to $\epsilon = 5\,\%$. Figure 2.16 shows the total triangulation time for the various KaHIP presets for a fixed choice of edge weight and sample size. In general, the time taken by the graph partitioning algorithm is very small compared to the DT computations. This can be seen by the little runtime variation for distributions that do not have an exploitable underlying structure, such as uniform and normal distribution. For the random bubble distribution, the influence of the choice of KaHIP configuration is more pronounced, as the quality of the partitioning has a larger influence on the overall runtime of the algorithm. For nearest sample point assignment (NSA), the PARALLEL preset is the best choice, whereas it is the worst choice for nearest partition center assignment (NCA), which benefits most from the ECO configuration. Overall distributions and partitioning schemes, the PARALLEL configuration presents a balanced choice and will be the default for all subsequent experiments.

2.5.2.4 Edge Weights

As discussed in Section 2.5.1, sparse regions of the input points—which are desirable as partition borders—result in long Delaunay edges in the sample triangulation. Since graph partitioning minimizes the weight of the cut edges, the edge weight needs to be inversely related to the Euclidean length of the edge, refer to Table 2.4. Figure 2.17 shows the total triangulation time for the various proposed edge weights for a fixed choice of KaHIP configuration and sample size. As dense regions of the input point set are reflected by *many* short edges in the sample triangulation, even constant edge weights result in a sensible partitioning. However, for input distributions with an exploitable structure, such as random bubbles, logarithmic edge weights are the best choice for NSA, due to the increased incentive to cut through long—ergo cheap—Delaunay edges. Again, they are the worst choice for NCA, which benefits most from linear edge weights. However, as logarithmic edge weights are a balanced choice over all inputs, we set them as default for all subsequent experiments.

2.5.2.5 Geometric Primitive

The geometric primitive used to determine the border simplices influences both the number of simplices in the border (accuracy) and the runtime required for the primitive itself. The intersection tests introduced in Section 2.5.1.2 each provide their own trade-off between accuracy and runtime. The grid-based intersection test requires the grid cell size as further configuration parameter, which introduces a trade-off between runtime—mainly memory allocation for the grid data structure—and accuracy. Figure 2.18 shows the total triangulation time for the bounding box, exact and grid-based intersection test, the latter for various choices of cell size $c_{\mathcal{G}}$. The bounding box test produces very large border triangulation and suffers from the resulting runtime penalty. On the contrary, the exact test produces the smallest border triangulation, the test itself, however, is rather expensive. The grid-based test provides a good trade-off between the two strategies. The finer grid better approximates the exact test. The impact of the finer grid on the runtime becomes apparent for the recursive bisection strategies, which need to allocate memory repeatedly. We use the grid-based intersection test with $c_{\mathcal{G}} = 1$ as default for all subsequent experiments.

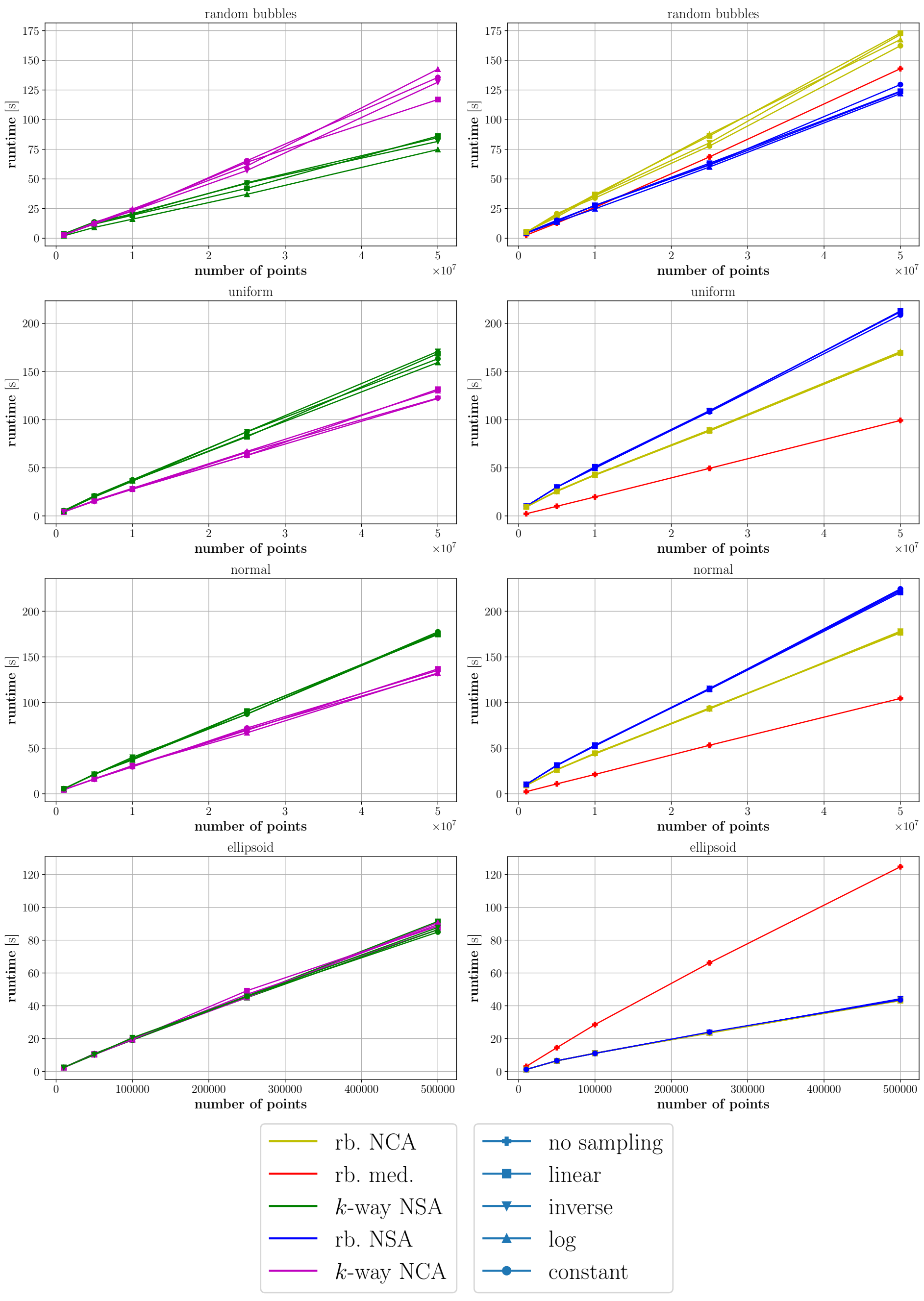


Figure 2.17 Edge weights experiments with $k = t = 16$, parallel KaHIP, grid-based intersection test with $c_{\mathcal{G}} = 1$ and $\eta(n) = \sqrt{n}$

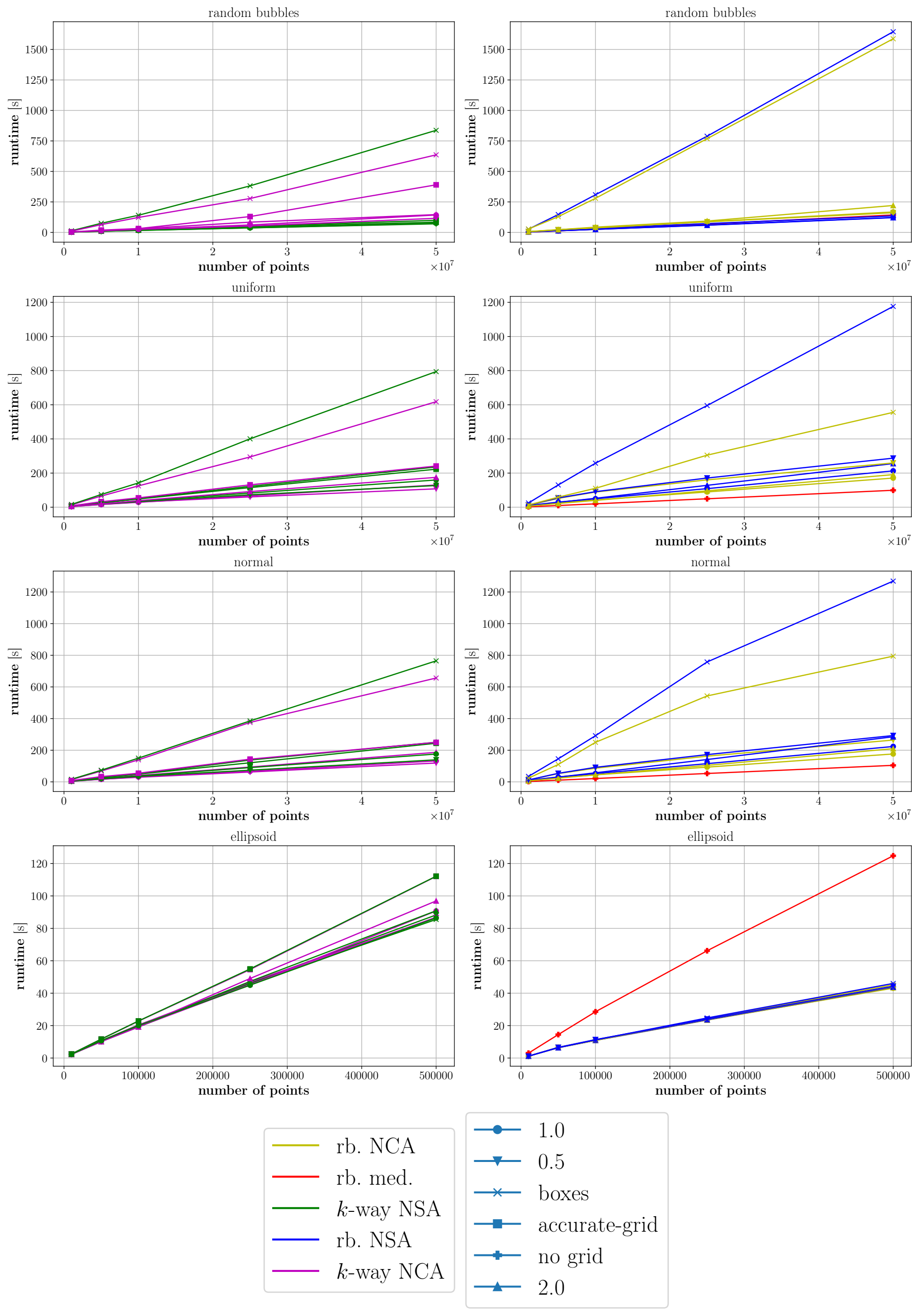


Figure 2.18 Intersection test experiments with $k = t = 16$, logarithmic edge weights, parallel KaHIP and $\eta(n) = \sqrt{n}$.

2.5.2.6 Partitioning Quality

Given a graph partitioning $(V_1 \quad \dots \quad V_k)$, its quality is defined by the weight of its cut, $\sum_{e \in C} \omega(e)$ for $C := \{e = (u,v), e \in E \text{ and } u \in V_i, v \in V_j \text{ with } i \neq j\}$. As mentioned in Section 2.5.1, the balance of the graph partitioning is ensured by the imbalance parameter ϵ, $|V_i| \leq (1+\epsilon)\lceil \frac{|V|}{k} \rceil$ for all $i \leq k$. When the partitioning of the sample triangulation is extended to the entire input set, this guarantee no longer holds. We therefore study two quality measures:

- the deviation from the ideal partition size, and
- the coefficient of variation of the partition sizes.

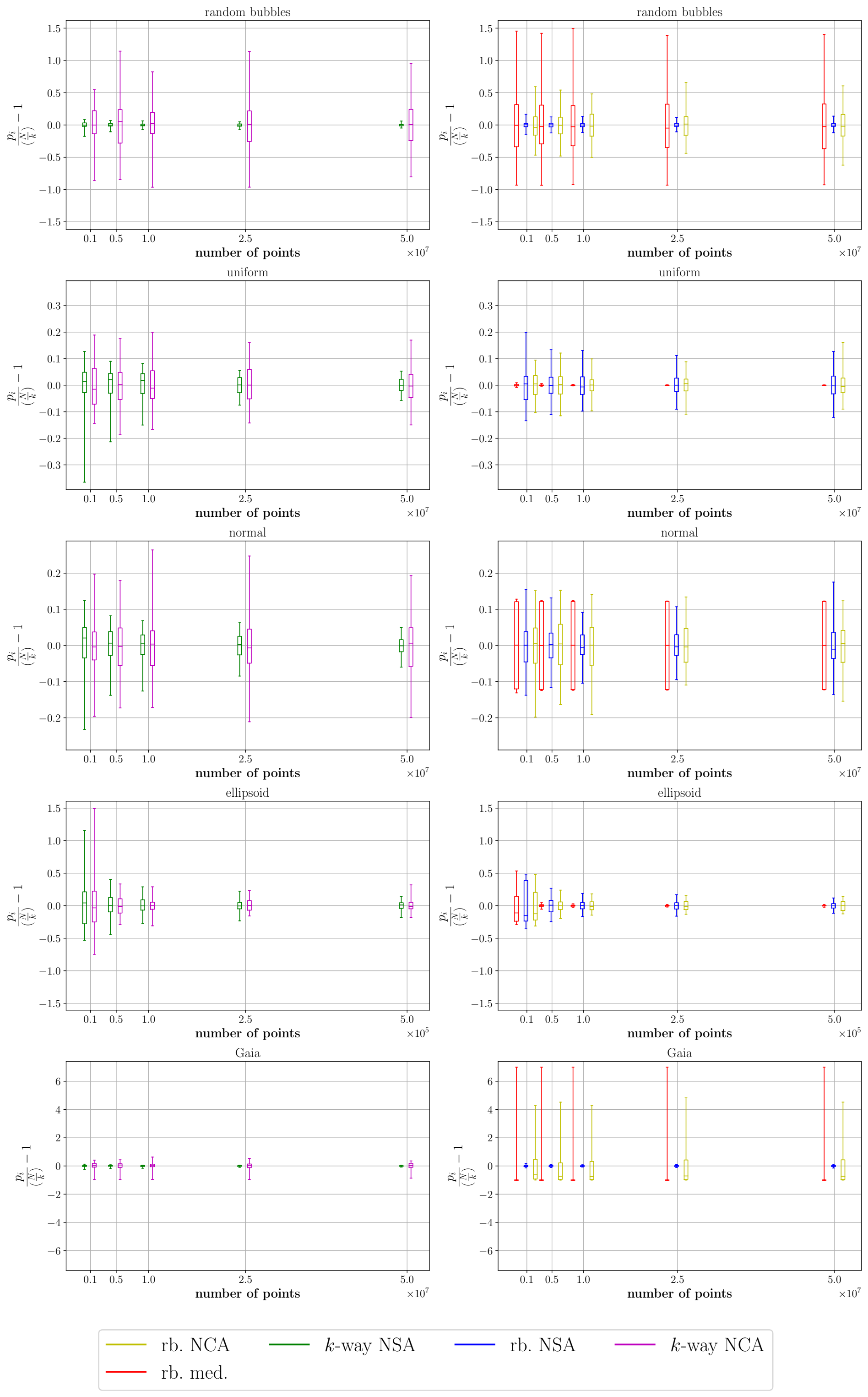
Deviation from the Ideal Partition Size. The deviation from the ideal partition size is given by $p_i / \frac{N}{k} - 1$, for k partitions with N points in total and partition sizes p_i, $i \leq k$, and is shown in Figure 2.19 for a fixed choice of KaHIP configuration, edge weights and two different sample sizes. Our sample-based approach with NSA produces almost equally sized partitions with little variance for the random bubble distribution and Gaia data sets. It clearly outperforms the cyclic median partitioning scheme for inputs with exploitable structure. NCA produces partitions of similar size, but with a higher variance for those inputs, but compares more favorable for less structured data sets. We attribute this to the more fine-grained representation of the sample triangulation by NSA. This can be mitigated by using recursive bisection with NCA, which exhibits less variance than when used with direct k-way partitioning. For both strategies, the larger sample size of $0.01n$ results in less variance compared to $\sqrt{n}$. Considering the uniform distribution, the cyclic median partitioning scheme produces perfectly balanced partitions with smooth cuts between them, whereas our new divide-step suffers from the jagged border between the partitions.

Coefficient of Variation. The coefficient of variation c_v of the partition sizes $p_i = |\mathbf{P}_i|$, $i \leq k$, is given by

$$c_v = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{\sum_{i \leq k}(p_i - \mu)^2}{k-1}}}{\frac{\sum_{i \leq k} p_i}{k}}.$$

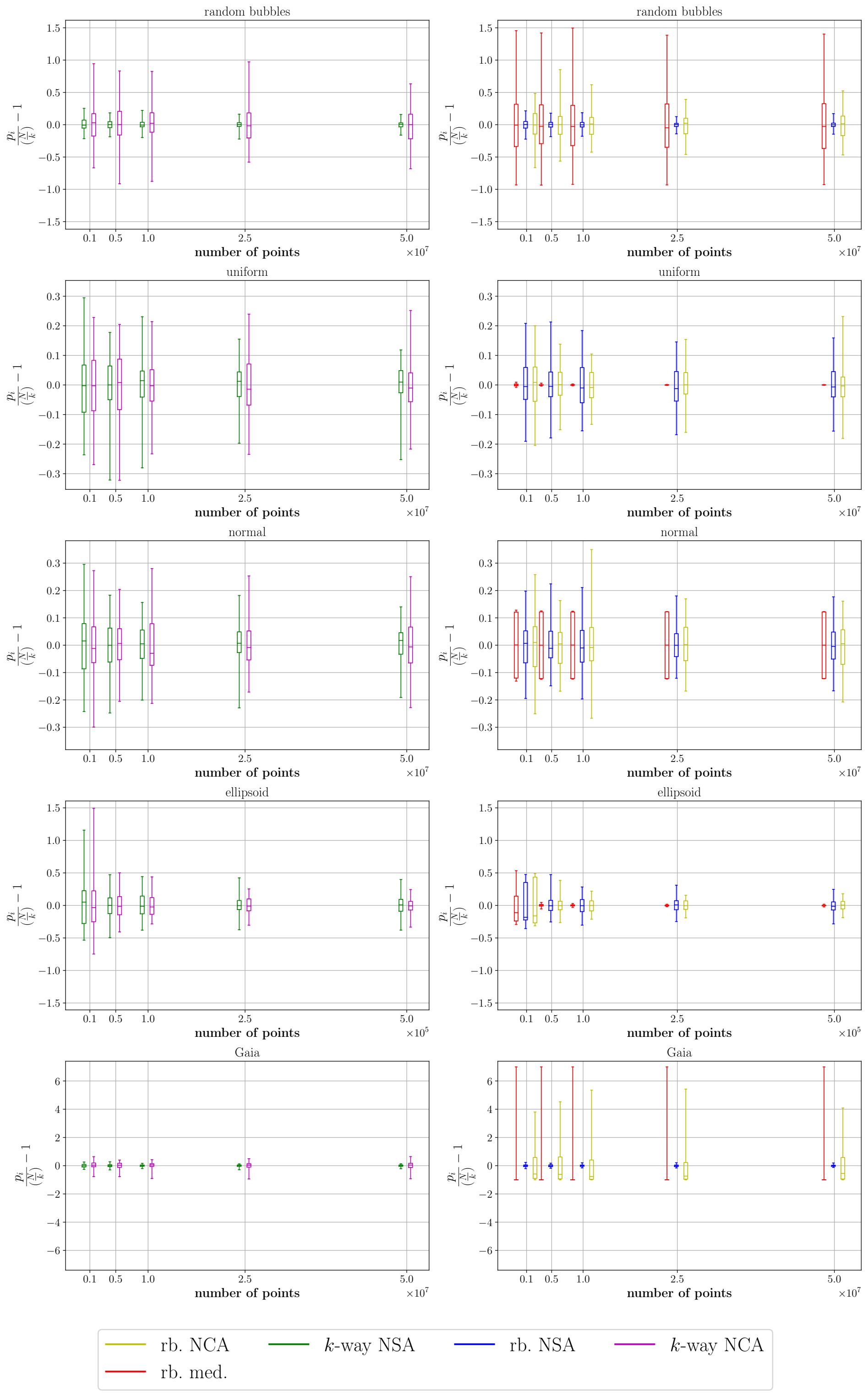
Figure 2.20 shows c_v for a fixed choice of KaHIP configuration, edge weights and two different sample sizes. For all distributions, our sample-based partitioning scheme with NSA robustly achieves a c_v of ≈6 % and ≈12 % for sample sizes $\sqrt{n}$ and $0.01n$, respectively.[10] Both lie above the chosen imbalance of the graph partitioning of $\epsilon = 5\,\%$, as expected. The larger sample size not only decreases the average imbalance but also its spread for various random seeds. Moreover, the deficits of the original cyclic median partitioning scheme become apparent: whereas it works exceptionally well for uniformly distributed points, it produces inferior partitions in the presence of an underlying structure in the input, as found for instance in the random bubble distribution. The results with NCA vary: for unstructured input it performs on par with NSA, but produces significantly more unbalanced partitions for inputs with structure. Moreover, whereas NSA achieves approximately the same balance for recursive bisection and direct k-way partitioning, NCA shows no clear trend, which partitioning scheme is preferable.

[10] We attribute the outlier for the ellipsoid distribution to the small input size.



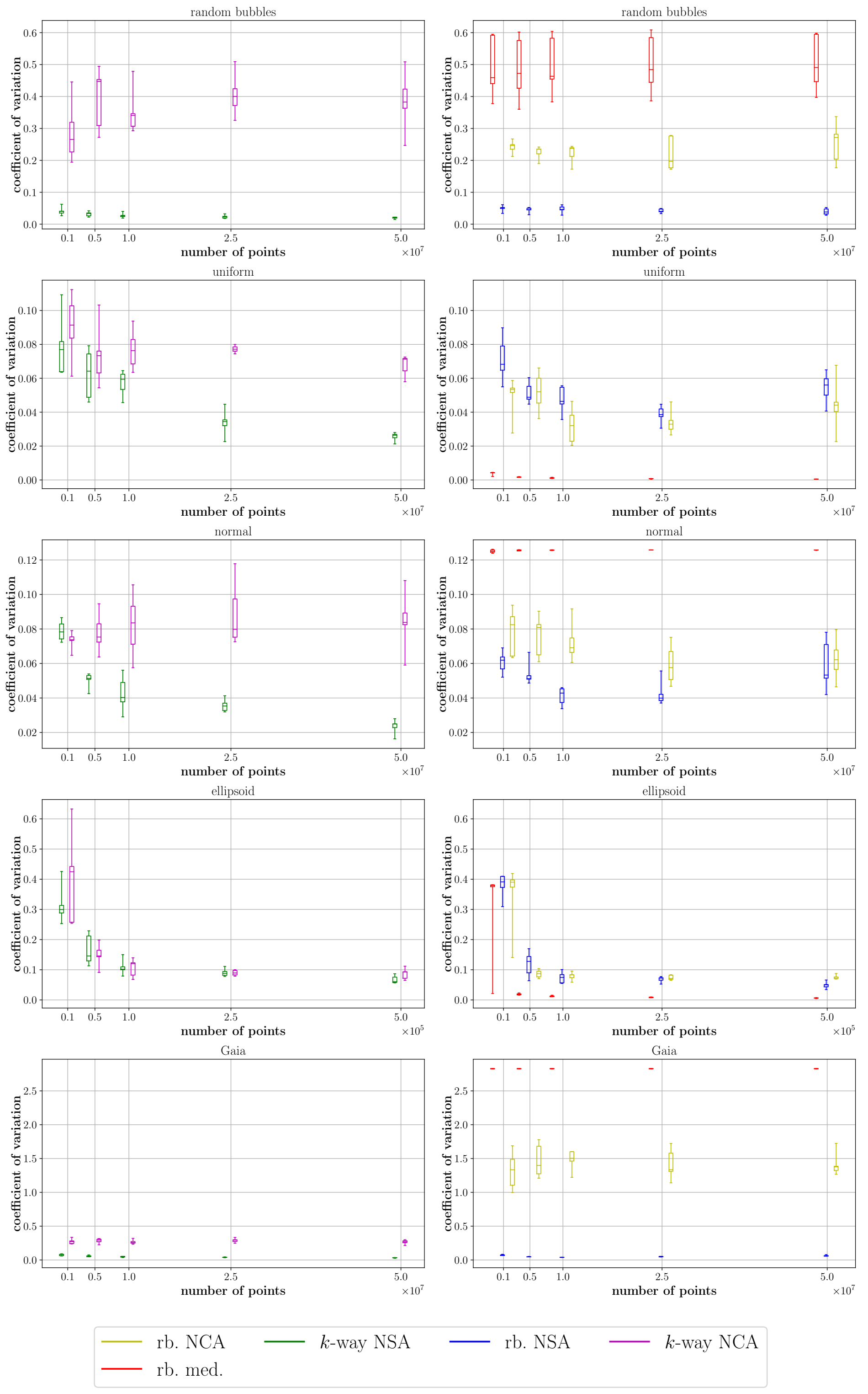
(a) Sample size $\eta(n) = 0.01n$.

Figure 2.19 Deviation from the ideal partition size for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.



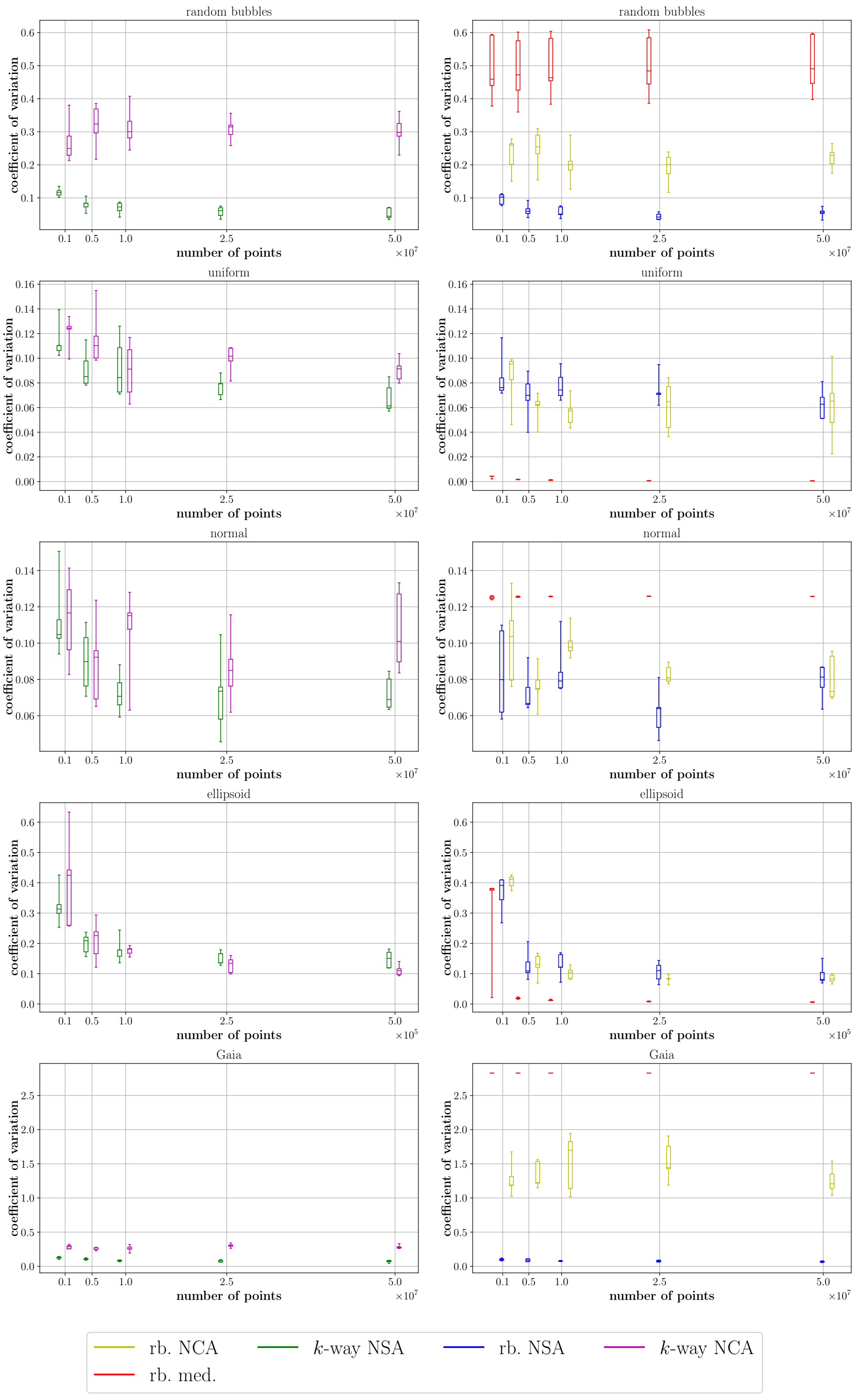
(b) Sample size $\eta(n) = \sqrt{n}$.

Figure 2.19 (cont.) Deviation from the ideal partition size for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.



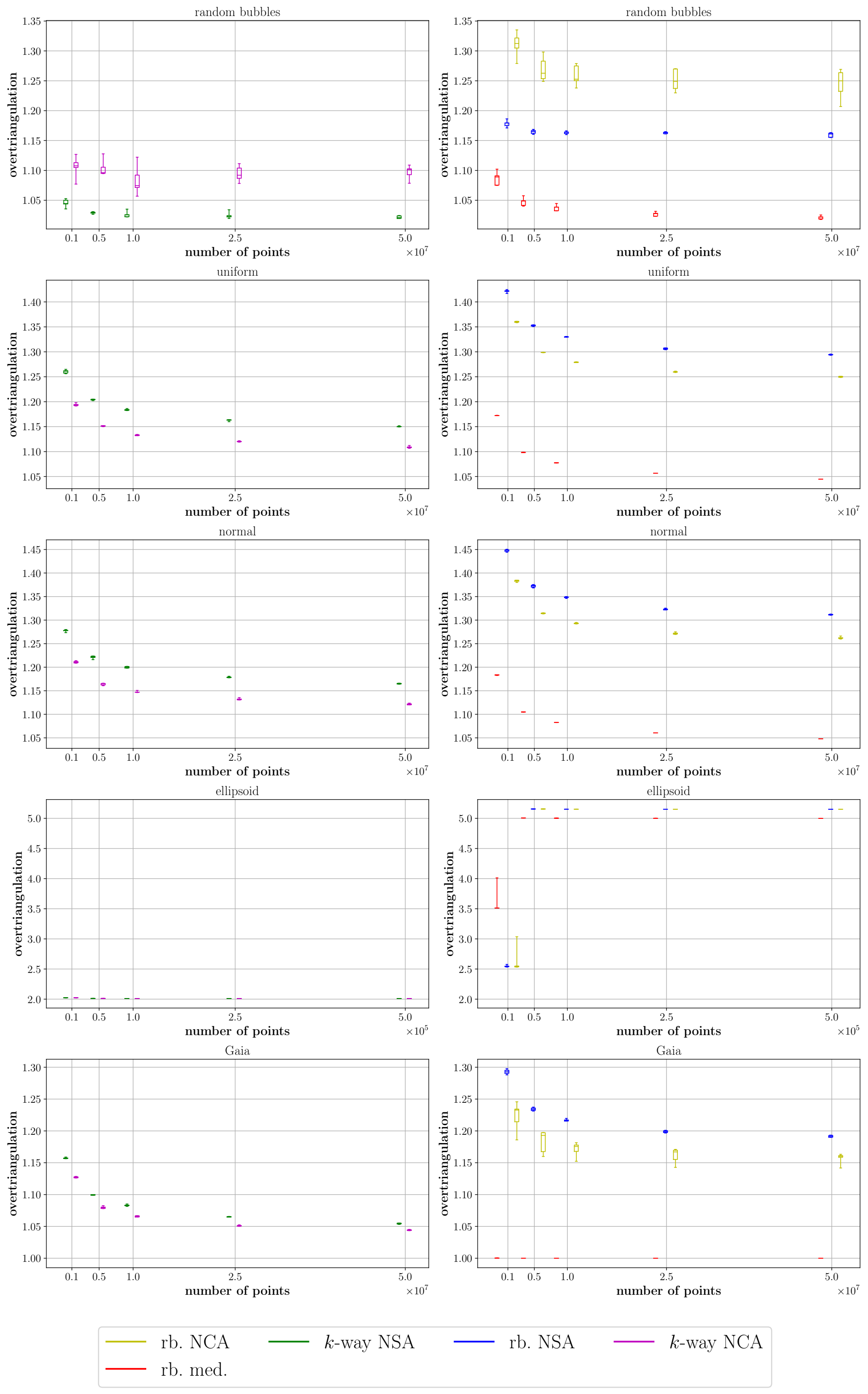
(a) Sample size $\eta(n) = 0.01n$.

Figure 2.20 Coefficient of variation of the partition sizes for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.



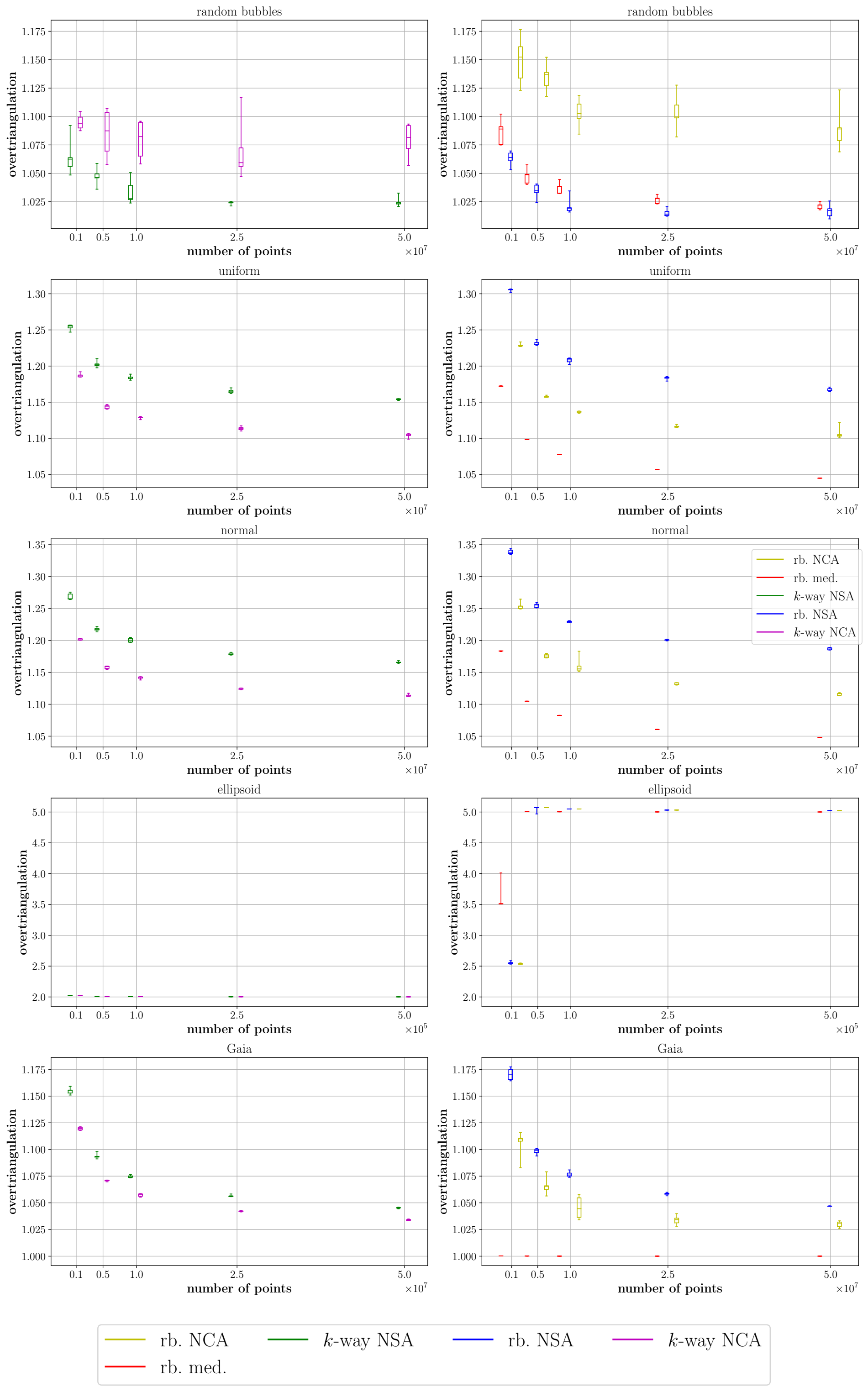
(b) Sample size $\eta(n) = \sqrt{n}$.

Figure 2.20 (cont.) Coefficient of variation of the partition sizes for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.



(a) Sample size $\eta(n) = 0.01n$.

Figure 2.21 Overtriangulation factor for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.



(b) Sample size $\eta(n) = \sqrt{n}$.

Figure 2.21 (cont.) Overtriangulation factor for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.

Overtriangulation. In total, our recursive algorithm triangulates more than the number of input points due to the triangulation of the sample points, and the triangulation(s) of the border point set(s). We quantify this in the overtriangulation factor o_{DT}, given by

$$o_{DT} := \frac{|\mathbf{P}| + \sum |\mathbf{P}_S| + \sum |\operatorname{vertices}(\mathbf{B})|}{|\mathbf{P}|}.$$

$\mathbf{B}$ is the set of border simplices, refer to Line 13 of Algorithm 2.7. For direct k-way partitioning, only one sample and one border triangulation are necessary; for recursive bisectioning there are a total of $k-1$ of each. Figure 2.21 shows the overtriangulation factor for a fixed choice of KaHIP configuration, edge weight and two different sample sizes. The smaller sample size of $\eta(n) = \sqrt{n}$ results in a lower overtriangulation factor for all recursive bisection schemes, as expected. For direct k-way partitioning, the effects of larger sample DT and more fine-grained representation of the underlying input structure—and thus reduction of the border triangulation size—approximately balance. For the random bubble distribution, the overtriangulation factor of our sample-based partitioning with NSA is on par or below that of the original cyclic median partitioning scheme, whereas NCA requires more points to be triangulated. The ellipsoid distribution is specifically tailored to be a hard input. Due to its large convex hull, almost all points are part of the border triangulation, therefore the oversampling factor is bound by the maximum recursion depth. For the normally distributed input point set, the central dense region needs to be cut multiple times for recursive bisection schemes in order to ensure balance between the partition sizes. Thus, more points are part of the border point set. For the uniform distribution, our new divide-step suffers from the jagged border between the partitions compared to the smooth cut produced by the cyclic median partitioning scheme. This results in more circumhyperspheres intersecting another partition and thus the inclusion of more points in the border triangulation, particularly for recursive bisection. Our experiments with the exact intersection test primitive confirm this notion.

2.5.2.7 Runtime Evaluation

We conclude our experiments with a study of the runtime of Algorithm 2.7 with the new sample-based divide step against the originally proposed cyclic median division strategy, its k-way variant as well as the parallel incremental insertion algorithm of CGAL. Figure 2.22 shows the total triangulation time for a fixed choice of KaHIP configuration, edge weights and sample size.

Direct k-way partitioning with NSA performs best on the random bubbles distribution, with a speedup of up to 50 % over the median partitioning schemes. Considering the k-way median partitioner, a small fraction of this speedup can be attributed to the k-way merging, however the larger fraction is due to the data sensitivity of the sample-based scheme. CGAL's parallel incremental insertion algorithm requires locking to avoid race conditions. It therefore suffers from high contention in the bubble centers, resulting in a high variance of its runtime and a 350 % speedup for our approach compared to it. For uniformly distributed points, our new divide-step falls behind the cyclic partitioning scheme as there is no structure to exploit in the input data and due to the higher overtriangulation factor of $o_{DT} = 1.10$ for k-way NCA partitioning compared to $o_{DT} = 1.05$ for cyclic median partitioning. As discussed in the previous section, the higher overtriangulation factor is caused by the jagged border between the partitions, resulting in a larger border triangulation and consequently also in higher merging times, as seen in Figure 2.24.

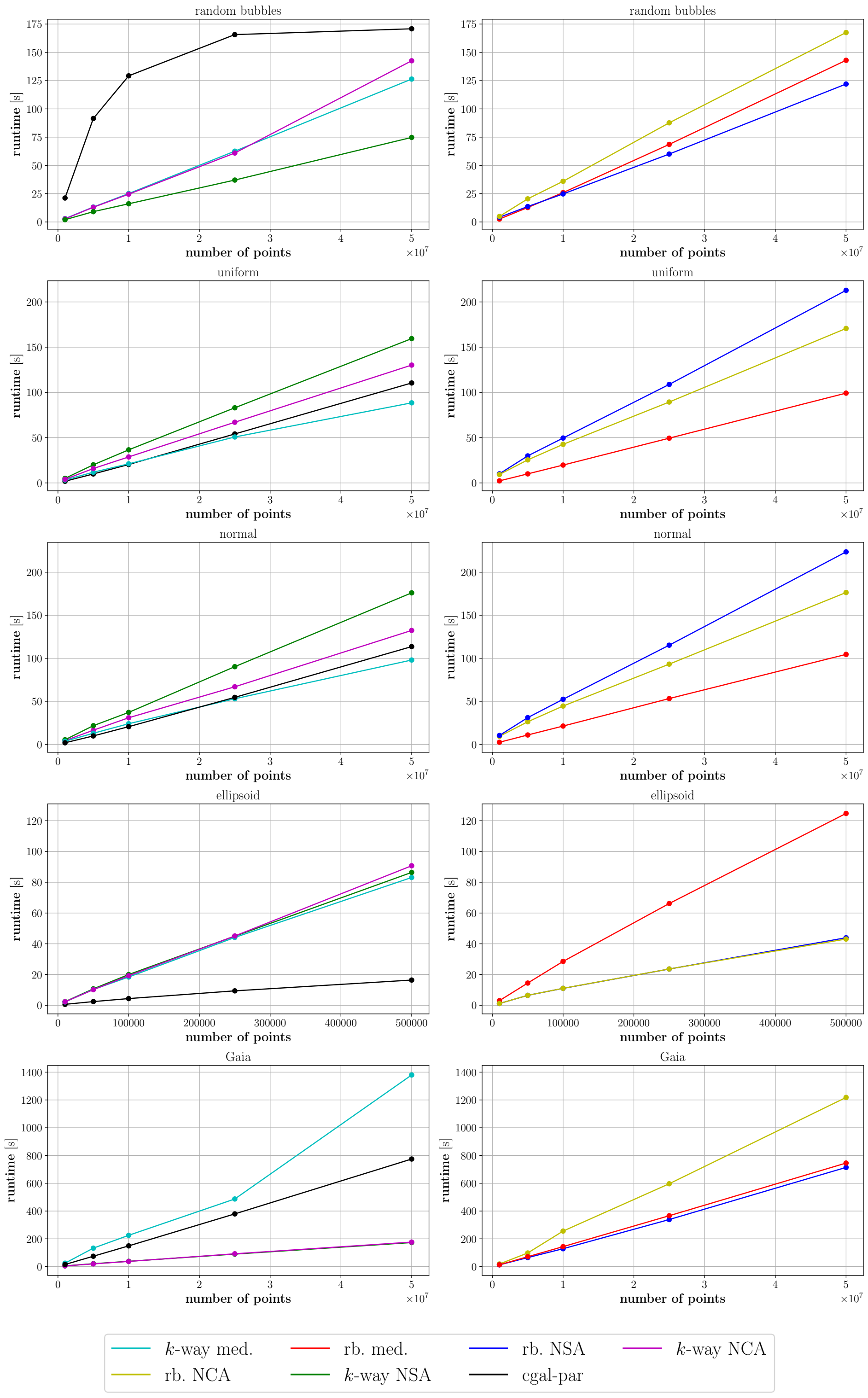


Figure 2.22 Runtime evaluation for $k = t = 16$, parallel KaHIP, $\eta(n) = \sqrt{n}$, grid-based intersection test with $c_{\mathcal{G}} = 1$ and logarithmic edge weights.

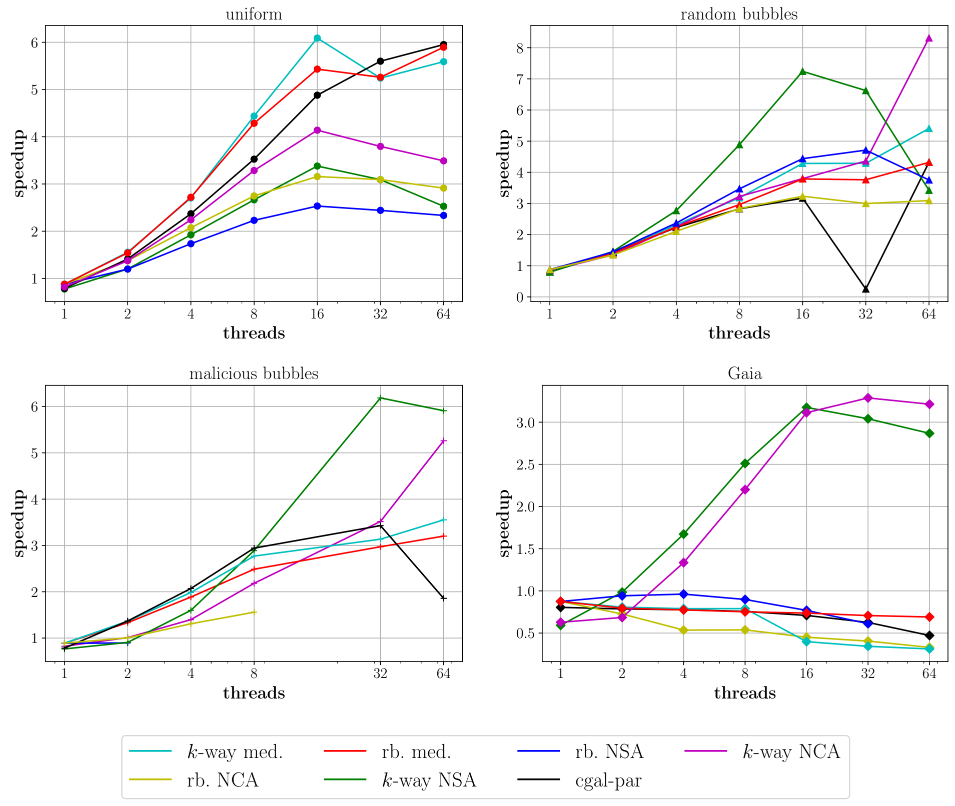


Figure 2.23 Absolute speedup over sequential CGAL for $k = t$, parallel KaHIP, $\eta(n) = \sqrt{n}$, grid-based intersection test with $c_{\mathcal{G}} = 1$ and logarithmic edge weights. All distributions are tested with the maximum number of points given in Table 2.5 points.

Of particular interest is the scaling behavior of our algorithm with an increasing number of threads. Figure 2.23 shows a strong scaling experiment for a fixed choice of KaHIP configuration, edge weights and sample size. The absolute speedup of an algorithm A over the sequential CGAL algorithm is given by $\text{Speedup}_A(t) := \frac{T_{\text{CGAL}}}{T_A(t)}$ for t threads.

In the presence of exploitable input structure—such as for the random bubble distribution—direct k-way partitioning, with both NSA and NCA, scales well on one physical processor (up to 16 cores). It clearly outperforms the original cyclic partitioning scheme and the parallel DT algorithm of CGAL. Nevertheless, NSA does not scale well to two sockets ($t > 16$ threads) and hyper-threading ($t > 32$ threads). The overtriangulation factor of 1.19 for 64 threads compared to 1.015 for 16 suggests that the jagged border produced by NSA results in prohibitively large border triangulations. NCA produces smoother borders and does not suffer from this effect.

Considering our real-world dataset, both direct k-way partitioning schemes also exhibit the best scaling behavior. As illustrated in Figure 2.14, the dataset comprises a large dense ring accompanied by several smaller isolated regions. This can be exploited to reduce border triangulation sizes and achieve a speedup, compared to the slowdown for the cyclic partitioning scheme and CGAL's parallel algorithm. The former is due to large border triangulations in the central ring, whereas the latter suffers from contention in the central region. Interestingly, whereas NCA compares poorly to NSA on the random bubbles distribution, it performs equally well on the Gaia data set.

The performance for normally distributed points can be attributed to the high overtriangulation factor, refer to Figure 2.21 and its discussion in the previous section.

Clearly, direct k-way partitioning outperforms recursive bisection in every configuration. Following the theoretical considerations in Section 2.5.1.1 regarding the number of merge-steps required, this is to be expected. A measure to level the playing field would be to only allow for $\eta(n)$ *total* number of sample points on all levels, i.e., adjust the sample size on each level of the recursion according the expected halving of the input size. Additionally, the recursive ascent could be omitted by performing a k-way merge step at the bottom of the recursion tree.

Figure 2.24 shows a breakdown of the algorithm runtime for a fixed choice of KaHIP configuration, edge weights and sample size. The sample-based partitioning requires 30 % to 50 % more runtime than the cyclic scheme. For favorable inputs with an exploitable structure, this additional runtime is more than mitigated by faster merging.

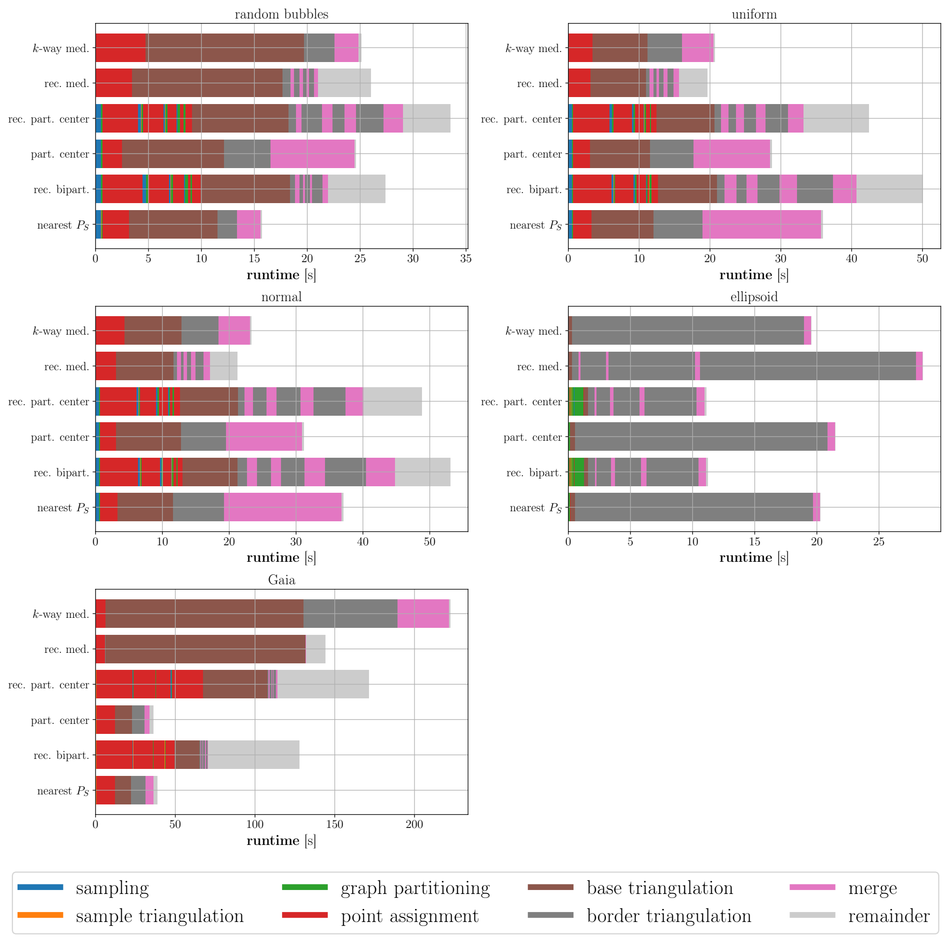


Figure 2.24 Runtime breakdown for $k = t = 16$, parallel KaHIP, $\eta(n) = \sqrt{n}$, grid-based intersection test with $c_{\mathcal{G}} = 1$ and logarithmic edge weights.

2.5.3 Conclusions

In this section, we presented a novel data-sensitive divide-step for our parallel D&C DT algorithm. The input is partitioned according to the graph partitioning of a Delaunay triangulation of a small input point sample. The partitioning scheme robustly delivers well-balanced partitions for all tested input point distributions. For input distributions exhibiting an exploitable underlying structure, it further leads to small border triangulations and fast merging. On favorable inputs, we achieve a speedup of almost a factor of two over our previous partitioning schemes and over the parallel DT algorithm of CGAL. These inputs include synthetically generated data sets as well as the Gaia DR2 star catalog. For uniformly distributed input points, the more complex divide-step incurs an overall runtime penalty compared to the original approach, opening up two lanes of future work: i) smoothing the border between the partitions to reduce the overtriangulation factor, and/or ii) an adaptive strategy that chooses between the classical partitioning scheme and our new approach based on easily computed properties of the chosen sample point set, before computing its DT. Furthermore, building on the idea of Lee et al. [LPP01], the partition borders could be traced with Delaunay edges to avoid merging all together. The sample-based divide step can also be integrated into our distributed-memory algorithm, where the improved load balancing and border size reduces the required communication volume for favorable inputs.

2.6 Random Delaunay Graph Generation

The design and analysis of graph algorithms for massive data sets requires large graph instances for benchmarking and testing. Real-world datasets are scarce and often not publicly available. Furthermore, they only represent current needs and do not reflect future requirements of graph processing. Network generators solve this problem to some extent. They provide synthetic instances based on random network models. These models are able to accurately describe a wide variety of different real-world scenarios: from ad-hoc wireless networks to protein-protein interactions [CSN09; MP10]. A substantial amount of work has been contributed to understanding the properties and behavior of these models. In theory, network generators allow us to build instances of arbitrary size with controllable parameters. This makes them an indispensable tool for the systematic evaluation of algorithms on a massive scale. For example, the well known Graph 500 benchmark [Mur+10], uses the R-MAT graph generator [CF06] to build instances of up to 2^{42} vertices and 2^{46} edges.

Even though generators like R-MAT scale well, the generated instances are limited to a specific family of graphs [CF06]. Many other important network models still fall short when it comes to offering a scalable network generator and in turn to make them a viable replacement for R-MAT. These shortcomings can often be attributed to the apparently sequential nature of the underlying model or prohibitive hardware requirements.

Lamm [Lam17] presents scalable network generators using a communication-free paradigm, i.e., the generators require no communication between PEs [SS16]. Each PE is assigned a disjoint set of local vertices. It then is responsible for generating all incident edges for this set of vertices. This is a common setting in distributed computation [Lum+07]. Expanding on the work of Lamm [Lam17], in [Fun+19] we present network generators for a wide range of graph models, including the classic Erdős-Rényi models $G(n, m)$ and $G(n, p)$ [ER59; Gil59] and different spatial network models including random geometric graphs (RGGs) [Jia04], random hyperbolic graphs (RHGs) [Kri+10] and random Delaunay graphs (RDGs). The latter was contributed by the author of this dissertation and is the focus of this section.

2.6.1 Preliminaries

In this section we introduce the concept of periodic boundary conditions for Delaunay triangulations and briefly review the communication-free random sampling algorithm of Sanders et al. [San+18] used in our generator.

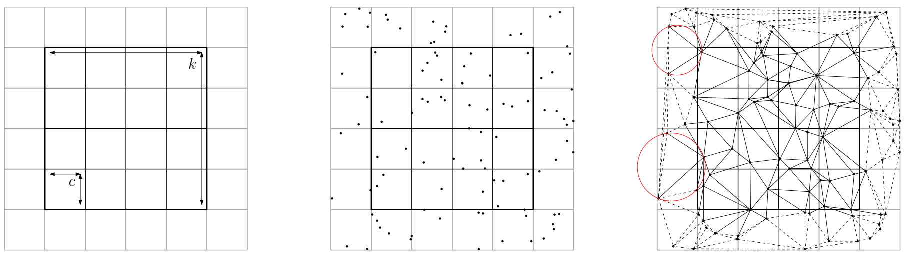
Periodic Boundary Conditions. For the RDG generator, we are concerned with Delaunay graphs defined by points sampled uniformly at random from the d-dimensional unit cube $[0, 1)^d$ for $d \in \{2, 3\}$. We view this as a good model for meshes as they are frequently used in scientific computing. Indeed, these simulations frequently use *periodic boundary conditions*, in order to make small simulations representative for a large simulated system (e.g., [Stu12]). This can also be viewed as replacing the infinite Euclidean space by a d-dimensional torus. We adopt these periodic boundary conditions, i.e., we implicitly compute the Delaunay triangulation of a point set where for every point $\mathbf{p}$ in the unit cube, also the points $\mathbf{p} + \mathbf{o}$ with $\mathbf{o} \in \{-1, 0, 1\}^d$ are in the point set. Two points in the unit cube are connected in the output, if any of their copies are connected. For a scalable distributed graph generator, periodic boundary conditions have the advantage that we avoid the need to compute some very long Delaunay edges that appear at the convex hull of random point sets.

Communication-free Distributed Random Sampling. To generate random vertices on distributed PEs, random sampling is a fundamental building block. Sanders et al. [San+18] propose a set of simple divide-and-conquer algorithms to sample n elements from a (finite) universe N on P PEs. Their algorithms follow the observation that by splitting the current universe into equal sized subsets, the number of samples in each subset follows a hypergeometric distribution. Based on this observation, they develop a D&C algorithm to determine the number of samples for each PE. In particular, each PE first determines its local interval of the input universe and then recursively generates a set of hypergeometric random variates. At each level of the recursion, it follows the remaining subset of the universe that contains its local interval. Hypergeometric random variates are synchronized without the need for communication by making use of pseudorandomization via (high quality) hash functions. To be more specific, for each subtree of the recursion, a unique seed value is computed (independent of the rank of the PE). Afterwards, a hash value for this seed is computed and used to initialize the pseudorandom number generator (PRNG) for the random variates. Therefore, PEs that follow the same recursion subtrees generate the same random variates, while variates in different subtrees are independent of each other. Once the remaining subset is smaller than a given threshold, a linear time sequential algorithm [Vit87] is used to determine the local samples. They continue to show that their algorithm runs in time $\mathcal{O}(n/P + \log P)$ with high probability (w.h.p.)[11] if the maximum universe size per PE is $\mathcal{O}(N/P)$ [San+18].

2.6.2 RDG Generator

Generating random Delaunay graphs can be separated into two main steps: 1) generating a random point set and 2) computing its Delaunay triangulation (DT). The point generation algorithm follows largely the one proposed by Lamm [Lam17] for random geometric graphs. It is restated in the following for completeness. The DT construction algorithm is proposed by the author of this dissertation and is specifically tailored to benefit from the uniform distribution of the generated points.

[11] i.e., with probability of at least $1 - P^{-c}$ for any constant c



(a) Chunks and cells. **(b)** Generated points. **(c)** Generated RDG.

Figure 2.25 Illustration of the RDG generator. The PE's chunk is marked by bold, black lines, the cells by thinner ones. Cells of neighboring chunks are marked by gray lines. For the generated DT, only edges with at least one endpoint in the PE's chunk (solid lines) are stored locally. As there are circumcircles not fully contained in the bounding box of the generated cells (red), another layer of neighboring cells must be generated and the DT re-triangulated.

Point Generation. Each PE generates a random local point set using the communication-free sampling algorithm described in Section 2.6.1. The generator uses the notion of chunks. A chunk represents a rectangular section of the unit cube. We partition the unit cube into P disjoint chunks and assign one of them to each PE. There is one caveat with this approach, in that the possible values for P are limited to powers of d. To alleviate this issue, we generate more than P chunks and distribute them evenly between PEs. Specifically, we generate $k = 2^{db} \geq P$ chunks, with $b \geq 1$, and distribute them to the PEs in a locality-aware way by using a Z-order curve [Mor66]. Each PE is then responsible for generating the vertices in its assigned chunks, using a divide-and-conquer approach. Each chunk is further subdivided into cells of side length c. We set c to the mean distance of the $(d+1)$th-nearest neighbor for n vertices distributed uniformly in the unit d-hypercube [BC08],

$$c \approx \sqrt[d]{\frac{d+1}{n}}, \text{ resulting in } n_c \approx \frac{n}{d+1} \text{ cells.} \tag{2.2}$$

Therefore, for a vertex v in a particular cell, all its Delaunay neighbors are, in expectation, in the same cell or in one of its $3^d - 1$ neighboring cells, refer to Figure 2.25a. For small values of n, we generate at least $k = 2^{db}$ cells, ensuring that a cell is always contained in one chunk and therefore assigned to a single PE. For large n we limit the number of cells n_c to a configurable maximum, in order to limit the memory consumption of the cell data structure.

The probability for a vertex to be assigned to an individual cell p is the ratio of the area of the cell to the area of the whole unit cube. Thus, we can generate $k = 2^{db}$ binomial random variates to compute the number of vertices within each of the chunks. The binomial distribution is parameterized using the number of remaining vertices n and the aforementioned chunk assignment probability p. The variates are generated by exploiting pseudorandomization via hash functions seeded on the current recursion subtree. Therefore, we generate the same variates on different PEs that follow the same recursion to a particular chunk. In turn, we require no communication for generating local vertices. Note that the resulting recursion tree has at most $\lceil \log k \rceil$ levels and size $(2^d k - 1)/2^d - 1$. Once a PE is left with a single chunk, we compute additional binomial random variates to get the number of vertices in each cell of side length c.

As we want each PE to generate *all* incident Delaunay edges for its local vertices, we have to generate cells adjacent to the PE's chunk(s). Because of our choice for the cell side length c, this means, in expectation, we have to generate all cells directly adjacent to the chunk(s) of a PE. Due to the communication-free design of our algorithm, the generation of these cells is done through re-computations using the same divide-and-conquer algorithms as for the local cells. We therefore repeat the vertex generation process for the neighboring cells, as shown in Figure 2.25b. Note that for sufficiently large graphs, each chunk consists of many cells so that redundantly generating border layers of cells becomes a negligible overhead. After all points are generated, we can compute the DT according to the following algorithm.

DT Computation. To produce the DT of the generated point set, our algorithm proceeds as follows. Each PE considers its assigned chunks plus a *halo* of neighboring cells. Initially, the cells directly adjacent to the chunk are added to the halo. The PE computes the DT of the vertices in its chunks plus halo and checks two conditions:

- all points in the convex hull of the DT lie in the halo, not within the PE's chunks; and
- each simplex s with at least one point from inside a chunk has a circumsphere that is completely contained within the bounding box of the PE's chunks plus halo, i.e., s is *finalized* and cannot violate the Delaunay criterion for any point beyond the current halo.

The local computation finishes when both conditions are fulfilled. Otherwise, the halo is expanded by one layer of cells and the DT is updated with the newly generated points. Afterwards, the two conditions are checked again. This is a similar notion of finalized simplices as used in our D&C DT algorithm (Section 2.4) and by e.g., [Lo12]. When adding points of additional cells, our use of pseudorandomness ensures that all PEs generate the same vertices for the same cell. We assume periodic boundary conditions to produce well-formed DT graphs even for the outermost chunks and avoid long Delaunay edges at the border of the DT. In Figure 2.25c the second termination condition of our algorithm is not fulfilled and the halo must be expanded by one layer of cells.

We do not have a complete analysis of the algorithm but conjecture the following running time bound.

▶ **Conjecture 11.** *The RDG generator has to compute the DT for an expected n_{DT} number of points, with*

$$n_{\mathrm{DT}} := \frac{n}{P} + \mathcal{O}\left(\left(\frac{n}{P}\right)^{\frac{d-1}{d}}\right) \textit{ points,}$$

resulting in an expected running time of $\mathcal{O}(n_{\mathrm{DT}} + \log P)$ for uniformly distributed vertices.

Proof (sketch). Each PE can generate the vertices of its chunk and a *constant* number of neighboring chunks in $\mathcal{O}(n/P + \log P)$ time [Fun+19, Lemma 5].

With our choice of cell side length according to Equation (2.2), $c = (\frac{d+1}{n})^{\frac{1}{d}}$, the number of cells in a chunk's halo with r layers of cells is given by

$$2d\sum_{i=1}^{r}\left(\left(\frac{n}{(d+1)P}\right)^{\frac{1}{d}} + i\right)^{d-1} \in \mathcal{O}\left(\left(\frac{n}{P}\right)^{\frac{d-1}{d}}\right) \text{ if } r \text{ is constant and } d \ll \frac{n}{P}.$$

Each cell contains $d+1$ vertices in expectation. For uniformly distributed points, we conjecture that, with our heuristic choice of cell side length c, only few layers of cells need to be added to the halo beyond the directly adjacent ones. Our experiments confirm this notion, refer to Section 2.6.3.3. Therefore, only a constant number neighboring cells need to be generated by a PE to determine the DT of the vertices of its chunk.

The Delaunay triangulation of a uniformly distributed point set can be computed in expected linear time [Mau84], yielding Conjecture 11. ◀

2.6.3 Experimental Evaluation

In the following we present the experimental evaluation of our RDG generator, regarding its running time and scaling behavior. For an evaluation of the other implemented generators we refer to [Fun+19].

2.6.3.1 Implementation

A C++ implementation of our graph generators, called KaGen, is publicly available on GitHub.[12] We use Spooky Hash[13] as a hash function for pseudorandomization. Hash values are used to initialize a Mersenne Twister [MN98] in order to generate uniform random variates.

[12] `https://github.com/KarlsruheGraphGeneration/KaGen`
[13] `http://www.burtleburtle.net/bob/hash/spooky.html` (accessed 08-12-2023)

All algorithms and libraries are compiled using GCC version 5.4.1 using optimization level `fast` and `-march=native`. In the distributed setting, we use Intel MPI version 1.4 compiled with GCC version 4.9.3. Our implementation uses the CGAL library [HS15] to compute the DT of the vertices of a chunk and its halo. CGAL provides a state-of-the art DT construction algorithm, which is also used as base case algorithm in our D&C DT construction algorithm, presented in Section 2.4.

2.6.3.2 Experimental Setup

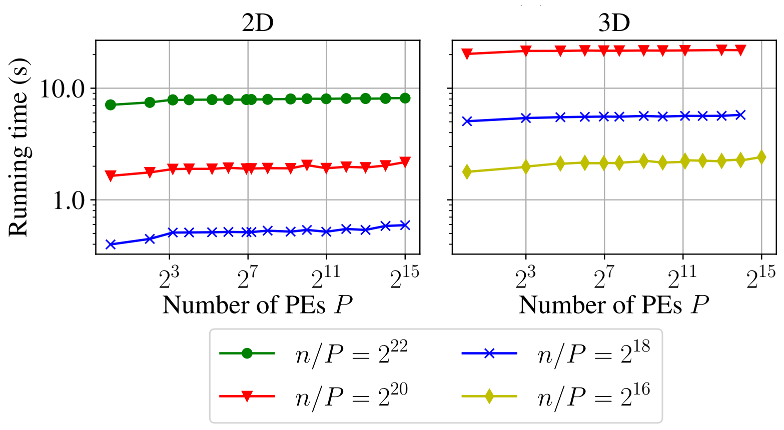
We use the Phase 1 thin nodes of the SuperMUC supercomputer for scaling experiments and parallel comparisons. The SuperMUC thin nodes consist of 18 islands and a total of 9216 nodes. Each compute node has two Sandy Bride-EP Xeon E5-2680 8-core processors with 32 GB of main memory. Each node runs the SUSE Linux Enterprise Server (SLES) operating system. We use the maximum number of 16 cores per node for our scaling experiments. The maximum size of our generated instances is limited by the memory per core (2 GB). If not mentioned otherwise, all results are averages of ten iterations with different seeds.

We analyze the scaling behavior of our algorithms in terms of weak and strong scaling. Weak scaling measures how the running time varies with the number of PEs for a fixed problem size *per PE*. Analogously, strong scaling measures the running time for a fixed problem size over *all PEs*. Due to memory limitations of the SuperMUC, strong scaling experiments are performed with a minimum of 1024 PEs. Again, results are averaged over ten iterations with different seeds.

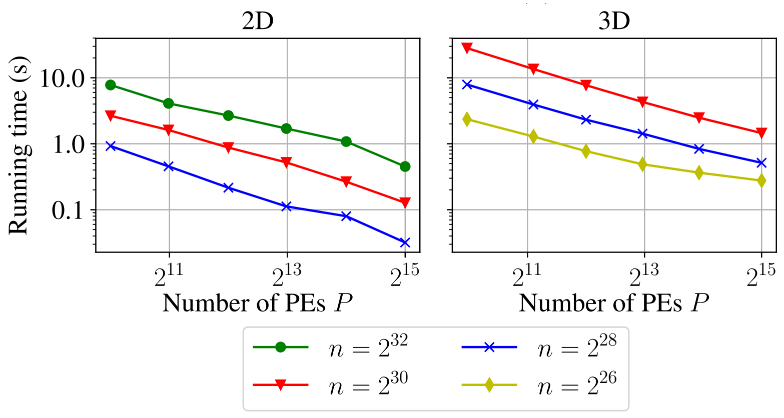
2.6.3.3 Results

For the weak scaling experiments, we vary the input size per PE n/P from 2^{18} to 2^{22} for the two-dimensional RDG and—due to memory constraints—from 2^{16} to 2^{20} for the three-dimensional one. Moreover, for 3D RDG and 2^{15} PEs, only the smallest input size could be computed within the memory limit per core of SuperMUC. For the strong scaling experiments, the input size varies from 2^{26} to 2^{32}. For weak scaling, our experiments show an almost constant running time—depicted in Figure 2.26—well in agreement with our conjectured asymptotic running time of $\mathcal{O}(n/P + \log P)$. The initial increase in runtime can be attributed to the redundant vertex generation of neighboring cells. As the halo rarely grows beyond the directly adjacent cells, no significant further increase in runtime can be observed for more than 2^8 PEs. For strong scaling, the running time decreases by a factor of 17 going from 2^{10} to 2^{15} PEs, resulting in an efficiency of approximately 0.55.

Figure 2.27 shows the maximum halo size required for all simplices of the computed DT to be finalized. Additionally, the cell side length according to Equation (2.2) is plotted. As conjectured, in most cases a single layer of cells is sufficient for the DT to converge.



(a) Weak scaling.



(b) Strong scaling.

Figure 2.26 Running time experiments for the RDG generator for two and three dimensions.

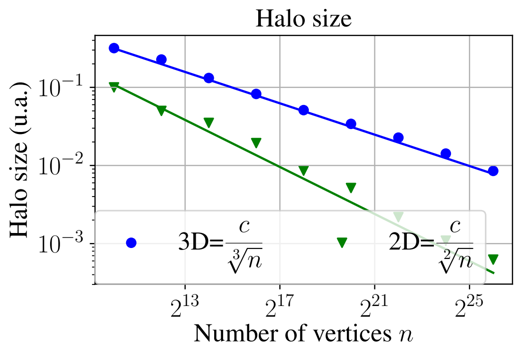


Figure 2.27 Maximum halo size required until the DT converged for varying number of vertices n. The lines show the cell size according Equation (2.2).

2.6.4 Conclusions

In this section we presented a scalable graph generator for random Delaunay graphs. The generator uses a divide-and-conquer scheme and pseudorandomization via hash functions to generate massive point sets on distributed PEs in a communication-free manner. Our experimental evaluation demonstrates the near-optimal scaling behavior of our generator. Therefore, our generators enable RDGs to be used for research on a massive scale. In order to help researchers to use our generators, we provide a widely usable open-source library. In future work, we would like to extend our generator to the streaming model in order to reduce the memory requirements per PE. Streaming algorithms for DT construction have been proposed in the literature [Ise+06; WGG11] and their use would allow us to generate even larger instances.

2.7 Summary and Outlook

In this chapter we presented two novel algorithms to construct the Delaunay triangulation of a given point set. The first algorithm is a divide-and-conquer algorithm for general point sets in arbitrary dimension, capable of processing large data sets in parallel on shared- and distributed-memory machines. The algorithm divides the input points into smaller subsets either through cyclic median splitting or using a sample-based approach that can better exploit the structure of the data set to allow for fast merging of the resulting partial triangulations. The merging of the partial triangulations is performed by re-triangulating a small subset of their vertices—the border vertices—and using efficient hash table operations to combine the three triangulations into one. All steps of the algorithm are parallelized. Our experiments on synthetic and real-world data sets show that our algorithm outperforms CGAL's DT construction algorithm. Furthermore, our experiments demonstrate that the choice of partitioning strategy has a significant impact on the performance of the algorithm, with either the median splitting or one of the sample-based approaches being the best choice depending on the structure of the input. Choosing the best partitioning strategy automatically remains for future work.

The second DT construction algorithm that we proposed is specifically designed for uniformly distributed points. It is integrated into the network generator KaGen, which is capable of producing massive graphs on distributed-memory machines. The algorithm exploits periodic boundary conditions and the uniformity of the input to limit the number of neighboring points that are required to determine the local triangulation of a processing element. We demonstrate its nearly perfect scaling behavior to over 32 000 PEs.

Impact and Subsequent Work. After publication of our D&C DT algorithm several other new algorithms for DT construction have been proposed [Car+19; MPR18; NR20; Su+20]. We will briefly discuss these algorithms and their relation to ours.

Su et al. [Su+20] present a serial incremental insertion algorithm with a novel adaptive Hilbert sorting scheme to reduce the time spent on locating the containing simplex of a newly added point. Marot et al. [MPR18] present an improved incremental insertion algorithm for shared-memory parallelism based upon the algorithm of Batista et al. [Bat+10]. They factorize the IN-SPHERE predicate into two parts—a query point independent simplex part that can be precomputed and a query point dependent part that can use the precomputed values. This optimization renders the runtime of the IN-SPHERE predicate negligible in the overall algorithm execution. The authors only evaluate their algorithm against sequential and parallel CGAL and do not compare against our algorithm. Their serial implementation outperforms CGAL by a factor of 3. Their parallel implementation achieves a speedup of 3.1 over parallel CGAL on four cores and 4.5 on 64 cores. In Section 2.4.4.1, we report a speedup of 4 on 32 cores over parallel CGAL. This suggests that our algorithm performs on par with Marot et al.'s one. Furthermore, our algorithm could benefit from integrating their serial implementation as base case algorithm.

Caraffa et al. [Car+19] present a distributed DT construction algorithm for Spark clusters. Each PE processes the points of one *tile*. After triangulating their local points, a star splaying algorithm [She05] is used to merge the local triangulations into a global one. The authors report an efficiency of their parallelization of 0.4 for 28 cores, corresponding to a speedup of ~ 11, which is similar to our distributed-memory algorithm. The authors do not compare their algorithm against other implementations.

Nguyen and Rhodes [NR20] present another distributed approach to DT construction for large-scale datasets. They partition the input into regions and compute a coarse-grained triangulation of the entire dataset to determine *independent* regions that can be triangulated in parallel. Their algorithm shares the same notion of finalized simplices and border simplices as ours. Finalized simplices are written to disk and removed from the triangulation, whereas border simplices are kept in the global triangulation to *refine* them with points from other partitions as they are scheduled for triangulation. The algorithm relies on a master-worker architecture to determine independent regions, schedule triangulations and collect border simplices. Their results show an impressive *relative* speedup of 135 for 256 PEs, but they do not compare their implementation against other algorithms. Furthermore, the authors only conduct experiments on two-dimensional datasets and do not mention whether their algorithm generalizes to higher dimensions.

In conclusion, our D&C DT algorithm is still competitive with the latest shared- and distributed-memory DT construction algorithms. Integrating the optimizations presented by Marot et al. [MPR18] could further improve its competitiveness.

Our network generator KaGen has been cited by 67 papers as of writing of this dissertation. KaGen has been used to devise and test new graph algorithms, served as basis to develop more specialized graph generators for targeted user groups as well as model and study the generated graph networks themselves. The Delaunay graph generator specifically has been used by e.g., [HST22; TPM20; vLTM18].

3 Yao Graph Generation

Summary: Yao graphs are geometric spanners that connect each point of a given point set to its nearest neighbor in each of k cones drawn around it. They were introduced to construct minimum spanning trees in d dimensional spaces and have since been applied to, e.g., wireless networks. An optimal $\mathcal{O}(n \log n)$-time algorithm to construct Yao graphs for a given point set has been proposed in the literature but—to the best of our knowledge—never been implemented. Instead, algorithms with quadratic complexity are used in popular packages to construct these graphs. In this dissertation we present the first implementation of the optimal Yao graph algorithm. We engineer the data structures required to achieve the $\mathcal{O}(n \log n)$-time bound and detail algorithmic adaptations necessary to take the original algorithm from theory to practice. We propose a priority queue data structure that separates static and dynamic events and might be of independent interest for other sweepline algorithms. Additionally, we propose a new Yao graph algorithm based on a uniform grid data structure that performs well for medium-sized inputs. We evaluate our implementations on a wide variety of synthetic and real-world datasets and show that they outperform current publicly available implementations by at least an order of magnitude.

Attribution: This chapter is taken mostly verbatim from [FS23b] and its accompanying technical report [FS23a]. The author of this dissertation was the main author and contributor of the paper, with editing provided by Peter Sanders.

3.1 Introduction

Yao graphs are directed graphs that connect each point of a given point set to its nearest neighbor in each of k cones, refer to Figure 3.1 for an example. A Yao graph is a cone-based spanner [Dam18], i.e., a subgraph of the complete Euclidean graph that preserves the shortest path between any pair of vertices up to a constant factor. More formally, a geometric t-spanner is a weighted graph, where for any pair of vertices there exists a t-path between them, which is a path with weight at most t times their spatial distance. The parameter t is known as the *stretch factor* of the spanner. Upper bounds on the stretch factor of Yao graphs have been the subject of extensive research. While the stretch factor of Yao graphs with $k \leq 3$ cones is proven to be unbounded, bounds have been established for all graphs with $k \geq 4$ cones [Bar+15]. Whereas for $k \geq 7$ cones the stretch factor is bounded by the general formula $\left(1+\sqrt{2-2\cos(2\pi/k)}\right)/(2\cos(2\pi/k)-1)$, bounds on Yao graphs with 4 to 6 cones require complex individual arguments [Bar+15; DN17].

Yao introduced this kind of graphs to construct minimum spanning trees in d-dimensional space [Yao82]. They have been applied to wireless networks for topology control [SVZ07; Zha+17] and routing [Si+14]. In 1990, Chang et al. [CHT90] presented an optimal algorithm to construct these graphs in $\mathcal{O}(n \log n)$ time. Due to the intricate nature of their algorithm and the reliance on expensive geometric constructions, to the best of our knowledge, there is no implementation of their algorithm available. Instead, an algorithm with an inferior $\mathcal{O}(n^2)$-time bound is used in the cone-based spanners package of the popular CGAL library [STP22].

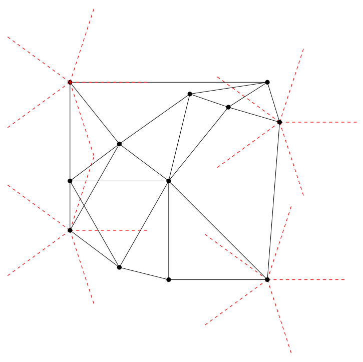


Figure 3.1 Yao graph for ten points and $k = 5$ cones. The five cones are illustrated as red dashed lines around four example points.

Contribution. In this dissertation we present the first publicly available implementation of Chang et al.'s optimal algorithm for Yao graph construction. We take their algorithm from theory to practice by engineering the data structures required to achieve the $\mathcal{O}(n \log n)$-time bound and provide detailed descriptions of all operations of the algorithm that are missing in the original paper, such as input point ordering, handling of composite boundaries and enclosing region search. In our event queue, we separate static (input point) events and dynamic (intersection point) events. This greatly improves the efficiency of priority queue operations and might be a useful technique for other sweepline algorithms. We test our algorithm on a wide range of synthetic and real-world datasets. We show that, despite the intricate nature of the algorithm and the use of expensive geometric constructions, our implementation achieves a speedup of an order of magnitude over other currently available implementations. Additionally, we develop a new Yao graph algorithm based on a uniform grid data structure that only uses simple geometric predicates, is easy to parallelize, and performs well for medium-sized inputs.

Outline. In Section 3.2 we review related work on the construction of Yao graphs. Section 3.3 presents three algorithms for Yao graph construction: a naive algorithm, our novel grid-based algorithm and the optimal algorithm of Chang et al., with the algorithmic adaptions necessary for its implementation. Further implementation details of Chang et al.'s algorithm, such as data structures and geometric operations, are described in Section 3.4. We evaluate our implementations and compare them against their competitors in Section 3.5. Section 3.6 summarizes this chapter and presents an outlook on future work.

Definitions. Given a set $\mathbf{P}$ of points in two-dimensional Euclidean space and an integer parameter $k > 1$, the Yao graph $Y_k = (\mathbf{P}, \mathbf{E})$ is a directed graph, connecting every point $p \in \mathbf{P}$ with its nearest neighbor in each of k cones [Yao82]. Every cone $\mathcal{C}_i = (\theta_L, \theta_R)$, $0 \le i < k$, is defined by its two limiting rays with angles $\theta_L = \frac{2(i+1)\pi}{k}$ and $\theta_R = \frac{2i\pi}{k}$. We denote the cone $\mathcal{C}_i$ with apex at point p as $\mathcal{C}_i^p$. We furthermore define, that the *left*—or counterclockwise—boundary ray with angle θ_L belongs to a cone $\mathcal{C}$, whereas the *right* one does not, i.e., for a given point $p \in \mathbf{P}$ and cone $\mathcal{C}_i^p$ we define the set of points $\mathbf{P} \cap \mathcal{C}_i^p := \{q \in \mathbf{P} : \sphericalangle(p, q) \in [\theta_L, \theta_R)\}$, with $\sphericalangle(p, q)$ denoting the angle between p and q. Then the edge set $\mathbf{E}$ of the Yao graph $Y_k = (\mathbf{P}, \mathbf{E})$ can be formally defined as $\mathbf{E} := \{(p, q) : \forall i \in [0, k), \forall p \in \mathbf{P}, q = \arg\min_{v \in \mathbf{P} \cap \mathcal{C}_i^p} (d(p, v))\}$, with $d(\cdot, \cdot)$ denoting the Euclidean distance function. Yao graphs can be generalized to higher dimensions [DGM09], however we focus on two-dimensional graphs in this dissertation.

3.2 Related Work

Yao [Yao82] presents a $\mathcal{O}\big(n^{5/3}\log n\big)$-time algorithm to compute a solution to the *Eight-Neighbors Problem*—a Yao graph with $k = 8$. It is based on a tessellation of the Euclidean space into cells. For a given point and cone, each cell of the tessellation is characterized whether it can contain nearest neighbor candidates in order to reduce the number of necessary distance computations. The problem is solved optimally by Chang et al., who present a $\mathcal{O}(n\log n)$-time algorithm for constructing the Yao graph of a given point set and a fixed parameter k [CHT90]. Their algorithm follows the same structure as Fortune's algorithm for constructing the Voronoi diagram of a point set [For87], using the sweepline technique originally introduced by Bentley and Ottmann for computing line-segement intersections [BO79]. However, even though there are many implementations of Fortune's algorithm available, there is no implementation of Chang et al.'s Yao graph algorithm that we are aware of. Instead, for instance, the CGAL library's cone-based spanners package implements a less efficient $\mathcal{O}\big(n^2\big)$-time algorithm [STP22]. Their algorithm is an adaption of a sweepline algorithm for constructing Θ-graphs [NS07]. Θ-graphs are defined similarly to Yao graphs, except that the nearest neighbor in each cone is not defined by the Euclidean distance but by the projection distance onto the cone's internal angle bisector. This allows for a $\mathcal{O}(n\log n)$-time sweepline algorithm, that uses a balanced search tree as sweepline data structure to answer one-dimensional range queries [NS07]. For Yao graphs, such a reduction in dimensionality is not possible, thus, CGAL's algorithm employs linear search within the sweepline data structure to find the nearest neighbor, leading to the $\mathcal{O}\big(n^2\big)$-time bound. However, CGAL's algorithm is much simpler to implement than the optimal algorithm proposed by [CHT90] and does not require geometric constructions, just predicates. Table 3.1 in Section 3.4 provides an overview of the required geometric operations by both algorithms.

3.3 Yao Graph Construction Algorithms

In this section, we discuss three algorithms to construct the Yao graph of a given point set. After briefly presenting a naive $\Theta\big(n^2\big)$-time algorithm as baseline, we introduce a novel grid-based algorithm that, while still having a worst-case time complexity of $\mathcal{O}\big(n^2\big)$, performs much better on realistic inputs. We then present our adaption of Chang et al.'s optimal sweepline algorithm.

3.3.1 Naive Algorithm

A *naive* $\Theta\big(n^2\big)$-time algorithm to construct the Yao graph can be obtained by slightly modifying a naive all nearest neighbor algorithm. In addition to comparing the distance of all input point pairs (p, q), the algorithm needs to furthermore determine the cone of q with respect to p. The cone $\mathcal{C}_i^p$ that q lies in with respect to p can first be approximated by $i = {}^{\sphericalangle(p,q)}/_{k}$. Then two oriented side of line tests with the bounding rays of $\mathcal{C}_i^p$ suffice to exactly determine the cone q lies in—either $\mathcal{C}_i^p$ or $\mathcal{C}_{i-1}^p$, if q lies directly on the right bounding ray of $\mathcal{C}_i^p$. Note that this test is independent of the number of cones k. The algorithm requires only two geometric predicates: distance comparison and oriented side of line test, and its only dependency on k is the number of nearest neighbors it needs to store per point.

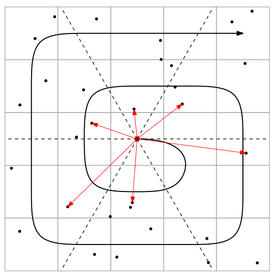


Figure 3.2 Grid-based Yao graph construction algorithm. The cone boundaries are represented by dashed lines. The algorithm visits grid cells in the order of the thick curve. Found edges of the Yao graph are labeled in red.

3.3.2 Grid-based Algorithm

Our novel *grid*-based algorithm refines the naive one by limiting the number of distance comparisons through the use of a uniform grid [Akm+89]. This is in contrast to Yao's original algorithm [Yao82], which uses a non-uniform tessellation. Our algorithm firstly places all input points in the uniform grid data structure that splits the bounding box of all input points in $\mathcal{O}(n)$ equal-sized cells. For each input point p, the algorithm first visits p's own grid cell and computes for each point q in the cell its distance to p and its cone $\mathcal{C}_i^p$. The algorithm then visits the grid cells surrounding p's cell in a spiraling manner, as shown in Figure 3.2. For each visited cell, the algorithm computes the distances and cones for the points contained in it with respect to p until all cones of point p are *settled*. A cone is settled, if a neighbor v has been found within that cone and no point in adjacent grid cells can be closer to p than v. Note that some cones may remain unsettled until all grid cells have been visited if no other input points lie within that cone for point p. This worst-case determines the $\mathcal{O}(n^2)$ runtime bound for our algorithm. We conjecture that the average-case runtime is much better, as we will show in Section 3.5. In addition to the geometric predicates required by the naive algorithm, the algorithm needs to determine the grid cell of a point and the minimum distance of a point to a grid cell. Similar to the naive algorithm, and in contrast to Yao's original algorithm, the grid-based algorithm does not depend on k.

3.3.3 Chang et al.'s Algorithm

In their 1990 paper, Chang et al. present a $\mathcal{O}(n \log n)$-time sweepline algorithm to compute the *oriented Voronoi diagram* (OVD) of a point set. Through a small modification, their algorithm can compute the *geographic neighborhood graph* (GNG)—or Yao graph—of a point set within the same, optimal, bound [CHT90, Theorem 3.2, Theorem 4.1].[1] To construct the Yao graph $Y_k = (\mathbf{P}, \mathbf{E})$ with k cones for point set $\mathbf{P}$, k sweepline passes are required, each considering a specific cone $\mathcal{C} = (\theta_L, \theta_R)$. Algorithm 3.1 provides an overview of the algorithm, with many details omitted for clarity.

The sweepline for cone $\mathcal{C} = (\theta_L, \theta_R)$ proceeds in direction $\tau = \frac{\theta_L + \theta_R}{2} + \pi$, i.e., opposite to the cone's internal angle bisector. Input points are swept in the order of their projection onto τ—represented as blue dashed line in Figure 3.3—given by sorting

$$\rho_\tau(p) := \begin{pmatrix} x \\ y \end{pmatrix} \cdot \begin{pmatrix} \cos\tau \\ \sin\tau \end{pmatrix} \quad \forall p = \begin{pmatrix} x & y \end{pmatrix}^T \in \mathbf{P}. \tag{3.1}$$

[1] Our implementation computes the Yao graph but can easily be modified to compute the OVD.

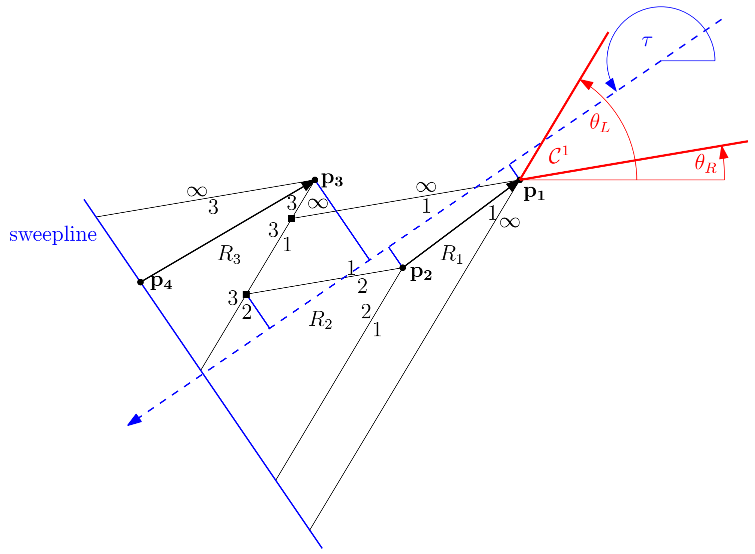


Figure 3.3 Example state of the sweepline algorithm for cone $\mathcal{C} = (\theta_L, \theta_R)$ (marked in red). Input points (circles, bold labels) are numbered in the order they are swept by the sweepline, with their projections on the cone's internal angle bisector shown in blue. Rays are labeled with the regions they are separating. Intersection events are marked with a square. Already determined edges of the Yao graph are indicated by arrows.

Algorithm 3.1 High-level sweepline algorithm.

```
Input:  Points $\mathbf{P} = \{p_1, \dots, p_n\}$ with $p_i \in \mathbb{R}^2$, $k \in \mathbb{N}$
Output: Yao graph $Y_k = (V, E)$

 1: $Y_k = (V, E) \leftarrow (\mathbf{P}, \emptyset)$
 2: for $i \in [0..k-1]$ do
 3:     $\theta_L \leftarrow \frac{2(i+1)\pi}{k}$   $\theta_R = \frac{2i\pi}{k}$          ▷ compute boundary angles of cone $\mathcal{C}_i$
 4:     priority queue $Q \leftarrow \mathbf{P}$     ▷ points are sorted along sweepline direction $\frac{\theta_L+\theta_R}{2} + \pi$
 5:     sweepline data structure $SL \leftarrow \emptyset$
 6:     while $p \leftarrow \text{popMin}(Q)$ do
 7:         if $p$ is input point then
 8:             find region $R_q$ containing $p$ in $SL$        ▷ region defined by rays $B_L$ and $B_R$
 9:             add edge $(p, q)$ to $Y_k$                  ▷ $q$ is nearest neighbor of $p$ in cone $\mathcal{C}_i$
10:             insert bounding rays of region $R_p$—$B_L^*$ and $B_R^*$—into $SL$ between $B_L$ and $B_R$
11:             add intersections of $B_L$ and $B_R$ with $B_L^*$ and $B_R^*$ to $Q$
12:         if $p$ is intersection then                    ▷ rays $B_L$ and $B_R$ intersect at $p$
13:             determine new boundary $B^*$ between left region of $B_L$ and right region of $B_R$
14:             replace $B_L$ and $B_R$ in $SL$ with $B^*$
15:             add intersections of $B^*$ with its neighbors in $SL$ to $Q$
16: return $Y_k$
```

All input points are inserted into an event priority queue Q with priority $\rho_\tau(p)$ and event type *input point*. Each input point p is the origin of a cone $\mathcal{C}^p$ with boundary rays B_L and B_R. The *region* R_p extends into the opposite direction of cone $\mathcal{C}^p$, as shown in Figure 3.3. R_p determines the region of the plane where point p is the nearest neighbor with respect to cone $\mathcal{C}^p$ for any point being swept after p. The invariant of the algorithm is that once a point has been swept, its nearest neighbor in cone $\mathcal{C}$ has been determined [CHT90, Lemma 3.1]. For instance, in Figure 3.3, p_2 is in the region of p_1, therefore p_1 is the nearest neighbor for p_2 with respect to cone $\mathcal{C}$. A boundary ray $B_\square$, with $\square \in \{L, R\}$, always separates the regions of two input points, thus it is defined by its point of origin $B_\square^p$ and angle $B_\square^\theta$ as well as its left and right region, $B_\square^l$ and $B_\square^r$ respectively. The region outside any point's cone is labeled with infinity. In Figure 3.3, the left boundary ray B_L of region R_2, originating at $B_L^p = p_2$ with angle $B_L^\theta = \theta_L$, separates region $B_L^l = 1$ and $B_L^r = 2$. Due to intersecting boundary rays, boundaries between regions can also be the union of a line segment and a ray, as described in detail in Section 3.3.3.2. However, for simplicity of presentation we still refer to these composite boundaries as boundary rays, unless this distinction is of relevance. The algorithm maintains an ordered data structure SL of rays currently intersecting the sweepline. The rays are ordered left-to-right and the data structure needs to support insert, remove and find operations in $\mathcal{O}(\log n)$ time, as well as access to the left and right neighbors of a given ray. In Section 3.4.2 we describe a balanced binary search tree that supports these operations and is tuned for our application. Algorithm 3.2 presents the details of our algorithm, which are further described in the following. An example execution of the algorithm is depicted in Figure 3.4.

3.3.3.1 Event Types

There are three different *event types*, each associated with a point p:

1. input points,
2. intersection points, and
3. deletion points.[2]

In the following, we describe how each event is handled by the algorithm. Through the execution of the algorithm, the priority queue Q contains all unprocessed input points, the intersection points of the boundaries of adjacent regions as well as deletion points for composite boundaries, ordered according to $\rho_\tau(p)$. If several events coincide, their processing order can be arbitrary.

1. Input Points. At the beginning of the algorithm execution, the priority queue Q is equal to point set $\mathbf{P}$ sorted according to $\rho_\tau(p)$. For an input point event with associated point p, the sweepline data structure SL is searched for the region R_q containing p. Section 3.4.2 describes this operation in detail. Region R_q is defined by its two bounding rays B_L and B_R and their associated regions $B_L^r = B_R^l = R_q$.[3] We can then add edge (p, q) to the edge set $\mathbf{E}$ of Y_k, as proven in [CHT90, Lemma 3.1]. The point p is the apex of region R_p, bounded to the left by $B_L^* = (p, \theta_L + \pi)$, separating regions R_q and R_p, and bounded to the right by $B_R^* = (p, \theta_R + \pi)$, separating regions R_p and R_q. These new rays are inserted into SL

[2] Not present in the original algorithm description by Chang et al. [CHT90] and omitted in Algorithm 3.1.

[3] Note that Chang et al. [CHT90] explicitly store rays and regions in their sweepline data structure. However, since a region is identifiable by its two bounding rays, we choose this simpler representation of the sweepline state.

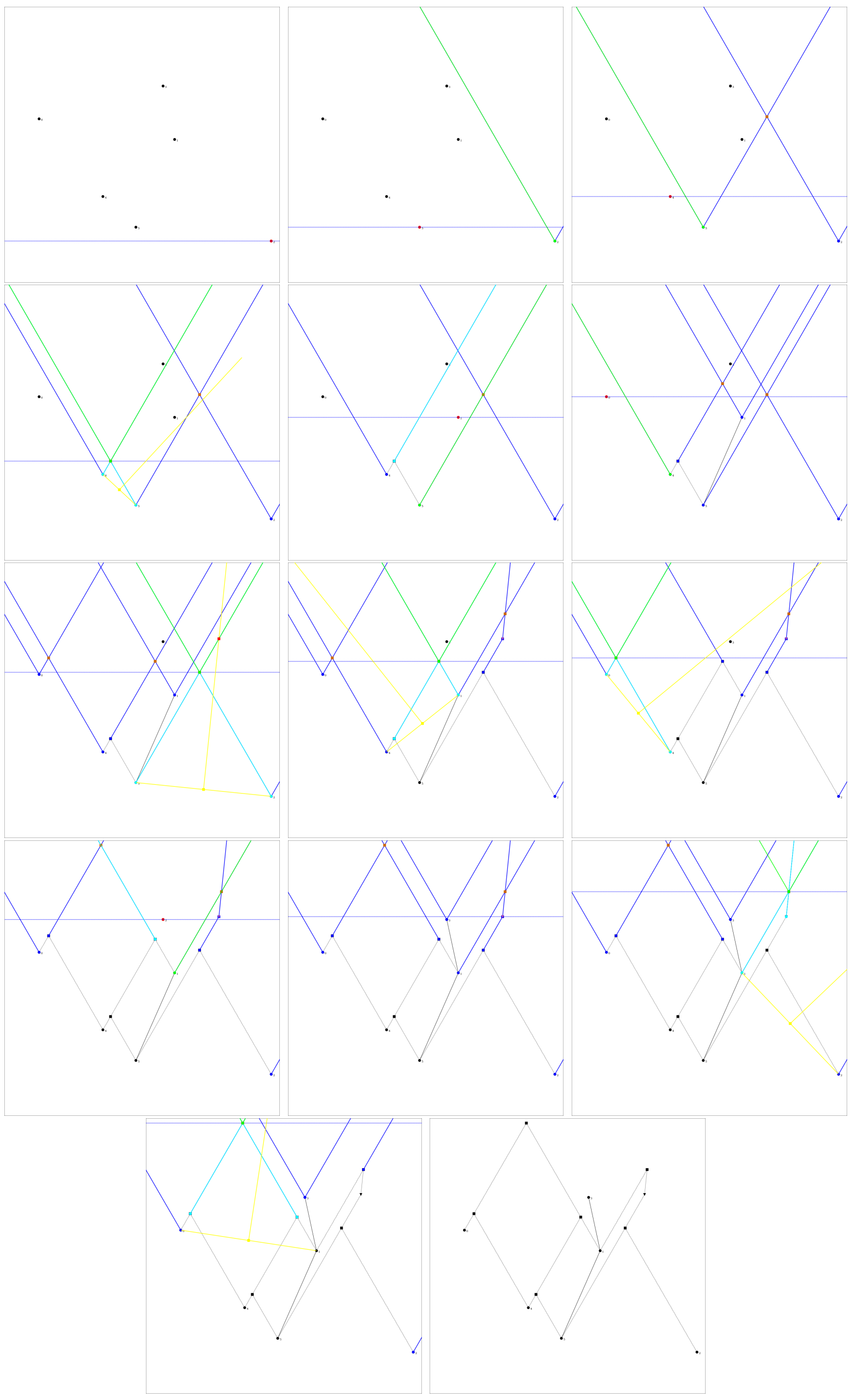


Figure 3.4 Example execution for $n = 6$ points and cone $\mathcal{C} = ({}^{4\pi}/_{3}, {}^{5\pi}/_{3})$, with $\tau = {}^{\pi}/_{2}$ (upward). The currently processed point p is marked in red, the sweepline is a dashed, blue line. All rays currently intersecting the sweepline are blue, except for the left boundary ray B_L (cyan) and right boundary ray B_R (green) of p. Intersection points are marked as squares, deletion points as triangles. For intersection events, the intersecting rays are cyan and the bisector line is yellow. Edges of the Yao graph are solid black lines and settled cone boundaries are dashed.

between B_L and B_R, forming the subsequence $[B_L, B_L^*, B_R^*, B_R]$. Lastly, intersection points of the considered rays need to be addressed. If B_L and B_R previously intersected in point v, its associated intersection point event needs to be removed from the priority queue Q, as B_L and B_R are no longer neighboring rays. Instead, possible intersection points between B_L and B_L^* as well as B_R^* and B_R are added to Q for future processing.

2. Intersection Points. An intersection point v is associated with its two intersecting rays B_L and B_R. They separate regions R_p, R_m and R_q, as shown in Figure 3.5, with

$$R_p := B_L^l \qquad B_L^r = R_m = B_R^l \qquad B_R^r =: R_q.$$

Region R_m terminates at intersection point v and a new boundary B^* between R_p and R_q originates at v. The shape of B^* depends on the configuration of points p and q and can either be a simple ray or a union of a line segment and a ray. Section 3.3.3.2 describes in more detail how B^* is determined. B^* then replaces the subsequence $[B_L, B_R]$ in the sweepline data structure. Again, intersection points of the considered rays need to be addressed. If B_L has an intersection point with its left neighbor or if B_R has an intersection point with its right neighbor, the associated intersection point events need to be removed from the event queue Q. Correspondingly, if B^* intersects its left or right neighbor, the appropriate intersection point events are added to Q.

3. Deletion Points. Deletion events are not part of the original algorithm described by Chang et al., as the authors do not specify how to handle composite boundaries in [CHT90]. We use them to implement boundaries consisting of a line segment and a ray. The deletion event marks the end of the line segment and the beginning of the ray. The ray replaces the composite boundary in the sweepline data structure.

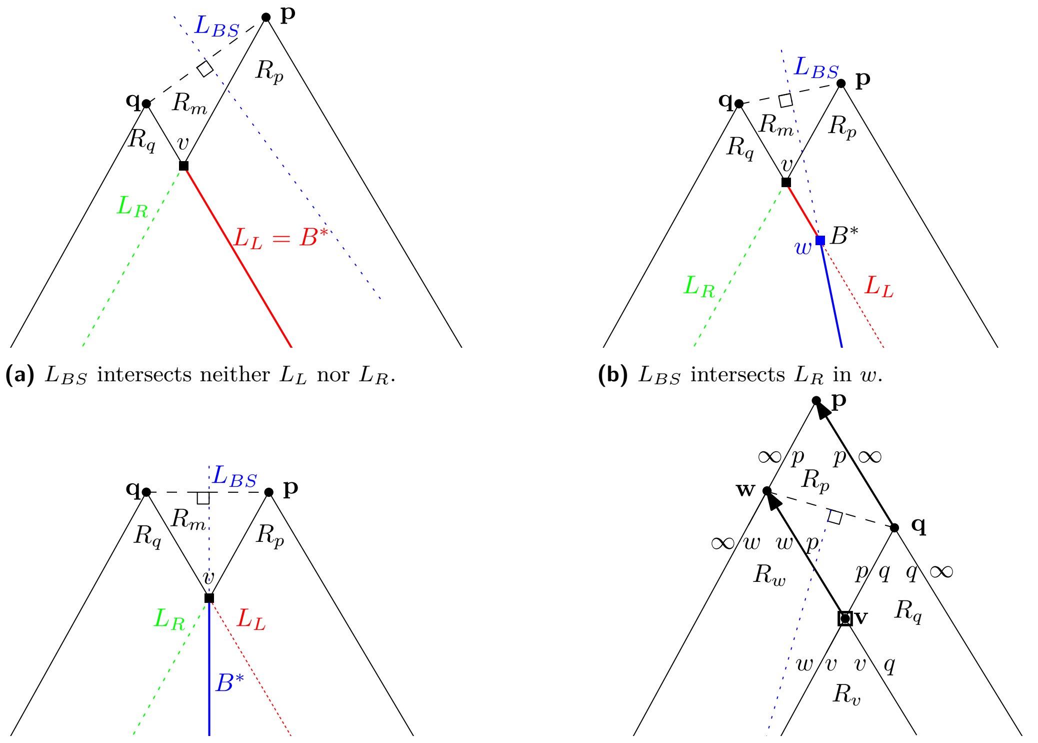
3.3.3.2 Boundary Determination

As described in the previous section, at an intersection point event v, the two intersecting rays B_L and B_R are merged into a new boundary B^*, separating regions $R_p := B_L^r$ and $R_q := B_R^l$. The shape of the boundary is determined by the configuration of points p and q. The determination is based on the number of intersection points between lines

- $L_L = (v, \theta_L + \pi)$ (green),[4]
- $L_R = (v, \theta_R + \pi)$ (red), and
- the bisector L_{BS} of line $\vec{pq}$, $L_{BS} = (\frac{\vec{p}+\vec{q}}{2}, \sphericalangle(p,q) + \pi/2)$ (blue, dashed).

If L_{BS} intersects neither L_L nor L_R then $B^* = L_L = (v, \pi + \theta_R)$ if p was swept before q, otherwise $B^* = L_R = (v, \pi + \theta_R)$ (Figure 3.5a). Intuitively, the region of the lower point with respect to the sweepline direction continues, whereas the upper region stops at intersection point v. If L_{BS} intersects both lines L_L and L_R, then the two intersection points must coincide with v (Figure 3.5c). In this case, $B^* = (p, \sphericalangle(p,q) + \pi/2)$, i.e., the boundary continues with the angle of the bisector from point v. Intuitively, the overlapping area between R_p and R_q is split at the bisector. Otherwise, i.e., L_{BS} intersects either L_L or L_R in a point w, the resulting boundary B^* will be the union of the line segment $\vec{vw}$ and ray $(w, \sphericalangle(p,q) + \pi/2)$ (Figure 3.5b). In this case, a deletion point event is added to the priority queue at point w.

[4] The given colors refer to the lines in Figure 3.5.



(a) L_{BS} intersects neither L_L nor L_R.

(b) L_{BS} intersects L_R in w.

(c) L_{BS} intersects both L_L and L_R in v.

(d) Input points lying on cone boundaries.

Figure 3.5 Illustration of the three possible configurations for boundary B^* following an intersection point event. In all examples input point p is swept before q. Lines L_L, L_R and L_{BS} are dotted, the resulting boundary B^* is denoted in bold. In Figure d), Yao graph edges are shown as bold arrows.

Input Points Collinear on Cone Boundaries. Chang et al. make the assumption that no two input points lie on a line with angle θ_L or θ_R. In the following we shall lift this requirement. Input points sharing a common line with angle $\theta_\Box$ become *visible* from each other. This impacts the regions the passing boundary rays are separating. Refer to Figure 3.5d for a graphical representation of the following discussion. Recall, that the *left*—or counterclockwise—boundary ray with angle θ_L belongs to a cone $\mathcal{C}$, whereas the *right* one does not. If a boundary ray $B_\Box = (p, \theta_\Box, B^l_\Box, B^r_\Box)$ intersects an input point q then $B_\Box$ terminates at q and a new boundary ray $B'_\Box$ is formed. If $B_\Box$ is a left boundary, i.e., $\theta_\Box = \theta_L$, then edge (p, q) is added to Y_k and $B'_\Box$ separates regions $B^l_\Box$ and R_q. Otherwise, if it is a right boundary then no edge is added to Y_k and $B'_\Box$ separates R_q and $B^r_\Box$. In Figure 3.5d, point q is on the left boundary of R_p, therefore edge (q, p) is added to Y_k, whereas w is on the right boundary of R_p and no edge is added. Point v lies on the right boundary of R_q and left boundary of R_w, hence edge (v, w) is added to Y_k.

3.3.3.3 Analysis

The total number of events processed by the sweepline algorithm is the sum of input point events N_{input}, intersection point events N_{IE} and deletion events N_{DE}. In order to bound the number of events processed by the sweepline algorithm, we consider the number of rays that can be present in the sweepline data structure during the execution of the algorithm.

Algorithm 3.2 Sweepline algorithm for cone $\mathcal{C}$ defined by (θ_L, θ_R). $L(p, \theta, R_a, R_b)$ and $L(\overleftrightarrow{pv}, R_a, R_b)$ denote the ray originating at p with angle θ and the line segment from p to v, respectively, both separating regions R_a and R_b.

```
Input:  Points $\mathbf{P} = \{p_1, \ldots, p_n\}$ with $p_i \in \mathbb{R}^2$, cone $\mathcal{C} = (\theta_L, \theta_R)$
Output: Edges $E_\mathcal{C}$ in cone $\mathcal{C}$ of Yao graph $Y_k = (V, E)$

 1: $\tau \leftarrow \frac{\theta_L + \theta_R}{2} + \pi$                      ▷ opposite of cone's internal angle bisector
 2: $Q \leftarrow \{(\rho_\tau(p), p, I) : p \in \mathbf{P}\}$                  ▷ initialize PQ with input points
 3: $SL \leftarrow \emptyset$
 4: $E_\mathcal{C} \leftarrow \emptyset$
 5: while $p \leftarrow$ popMin($Q$) do
 6:     if $p$ is input point then
 7:         $B_L, B_R \leftarrow$ findRegion($p, SL$)                          ▷ $B_L$ and $B_R$ enclose $p$
 8:         $E_\mathcal{C} \cup= (p, B_L^r)$                                   ▷ assert $\left(B_L^r == B_R^l\right)$
 9:         if $B_L \cap B_R = v$ then delete $v$ from $Q$
10:         $B_L^* \leftarrow L(p, \theta_L + \pi, B_L^r, R_p)$
11:         $B_R^* \leftarrow L\left(p, \theta_R + \pi, R_p, B_R^l\right)$
12:         insert $[B_L^*, B_R^*]$ into $SL$ between $B_L$ and $B_R$
13:         if $B_L \cap B_L^* = v$ then $Q \cup= (\rho_\tau(v), v)$
14:         if $B_R \cap B_R^* = v$ then $Q \cup= (\rho_\tau(v), v)$
15:     if $p$ is intersection then
16:         $B_L, B_R \leftarrow$ intersecting rays at $p$
17:         $a \leftarrow B_L^l$     $b \leftarrow B_R^r$
18:         if $B_L \cap \text{prev}(B_L) = v$ then delete $v$ from $Q$    ▷ left neighbor boundary on $SL$
19:         if $B_R \cap \text{succ}(B_R) = v$ then delete $v$ from $Q$    ▷ right neighbor boundary on $SL$
20:         $L_{BS} \leftarrow L(\frac{\vec{a}+\vec{b}}{2}, \sphericalangle(a,b) + \frac{\pi}{2}, R_a, R_b)$        ▷ bisector of line segment $\overleftrightarrow{ab}$
21:         $L_L \leftarrow L(p, \theta_L + \pi, R_a, R_b)$
22:         $L_R \leftarrow L(p, \theta_R + \pi, R_a, R_b)$
23:         if $L_L \cap L_{BS} = \emptyset = L_{BS} \cap L_R$ then         ▷ bisector intersects no line from $p$
24:             $B^* = L\left(p, \pi + \begin{cases} \theta_L & \text{if } \rho_\tau(a) < \rho_\tau(b) \\ \theta_R & \text{else} \end{cases}, R_a, R_b\right)$
25:         if $L_L \cap L_{BS} = p = L_{BS} \cap L_R$ then                 ▷ bisector intersects both lines in $p$
26:             $B^* = L(p, \sphericalangle(a,b) + \frac{\pi}{2}, R_a, R_b)$
27:         if $L_L \cap L_{BS} = v$ or $L_{BS} \cap L_R = v$ then          ▷ bisector intersects one line in $v$
28:             $B^* = L(\overleftrightarrow{pv}, R_a, R_b) + L(v, \sphericalangle(a,b) + \frac{\pi}{2}, R_a, R_b)$
29:             $Q \cup= (\rho_\tau(v), v)$                                     ▷ deletion event
30:         replace $[B_L, B_R]$ in $SL$ with $B^*$
31:         if $B^* \cap \text{prev}(B^*) = v$ then $Q \cup= (\rho_\tau(v), v)$
32:         if $B^* \cap \text{succ}(B^*) = v$ then $Q \cup= (\rho_\tau(v), v)$
33:     if $p$ is deletion point then
34:         $B \leftarrow$ ray belonging to $p$         ▷ $B = L(\overleftrightarrow{pv}, R_a, R_b) + L(v, \sphericalangle(a,b) + \frac{\pi}{2}, R_a, R_b)$
35:         replace $B$ in $SL$ with $L(v, \sphericalangle(a,b) + \frac{\pi}{2}, R_a, R_b)$
36: return $E_\mathcal{C}$
```

Every input point event adds two rays to the sweepline data structure SL, resulting in a total of $2n$ rays. It possibly removes one intersection event from the event queue Q and may add up to two new such events. Every intersection point event removes two rays from SL and adds one new ray, thus reducing the sweepline size by one. Therefore, at most $2n$ intersection events can be processed before all rays are removed from the sweepline. An intersection event possibly removes two additional intersection events aside from itself from Q and may add one new intersection event.[5] Additionally, one deletion event may be added to Q. Therefore, at most $2n$ deletion events may be processed, each of which leaves the number of rays and intersections unchanged. In total,

$$\begin{aligned} N_{\text{events}} &\leq N_{\text{input}} + N_{\text{IE}} + N_{\text{DE}} \\ &\leq n + 2n + 2n = 5n \end{aligned} \tag{3.2}$$

With a balanced binary search tree as sweepline data structures each event can be processed in $\mathcal{O}(\log n)$ time. Thus, each of the k sweepline passes requires $\mathcal{O}(n \log n)$ time.

3.4 Implementation Details

In this section, we highlight the data structures and geometric operations used in our implementation of Chang et al.'s Yao graph construction algorithm.

3.4.1 Geometric Kernels

The algorithm by Chang et al. [CHT90] requires various geometric predicates and constructions. We implement our own version of the required predicates and constructions in an *inexact* manner. Additionally, the user can employ kernels provided by the CGAL library. The EPIC kernel provides exact predicates and inexact constructions, whereas the EPEC kernel features exact predicates and exact constructions [Brö+22].

Table 3.1 lists the geometric predicates used by the different algorithms for Yao graph construction presented Section 3.3 and the one of CGAL's cone-based spanners package [STP22].

5 Technically, B^* can intersect both its neighbors, leading to two intersection events. However, when the first—as defined by $\rho_{\mathcal{T}}(\cdot)$—intersection event is processed, it will delete the second event from Q, as B^* is removed from SL and a new boundary ray is inserted by the first event.

Table 3.1 Comparison of geometric predicates used in algorithms for Yao graph construction.

	Chang et al.	CGAL	Naive	Grid
Complexity	$\mathcal{O}(kn \log n)$	$\mathcal{O}(kn^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Predicates				
Eucl. distance comp.	X	X	X	X
dist. to line comp.	X	X		
oriented side of line	X		X	X
Constructions				
cone boundaries	X	X	X	X
box construction				X
line projection	X			
ray intersection	X			

Only the sweepline algorithm requires the computation of particularly costly intersections. The naive as well as grid-based Yao graph algorithm require an oriented side of line predicate only if cone boundaries are constructed exactly, in order to determine the cone $\mathcal{C}_i^p$ a point q lies in with respect to point p. Additionally, the grid algorithm could construct the grid data structure using exact computations. However, in our implementation we only use inexact computations to place the input points into grid cells, as we did not encounter the need for exact computation in any of our experiments. Note that the determination whether a grid cell could hold a closer neighbor than the currently found one is done using the (exact) Euclidean distance comparison predicate.

In order to reduce the number of expensive ray-ray intersection calculations, we store all found intersection points in a linear probing hash table, with the two intersecting rays as key, see, e.g., Line 13 in Algorithm 3.2. If we need to check whether two rays are intersecting—e.g., Line 9 of the same algorithm—we merely require a hash table lookup.

3.4.2 Sweepline Data Structure

Chang et al. prove a $\mathcal{O}(n \log n)$-time complexity for the algorithm [CHT90, Theorem 3.2]. In order to achieve this bound, the data structure maintaining the rays currently intersected by the sweepline must provide the following operations all in $\mathcal{O}(\log n)$ time:

- insert,
- remove,
- predecessor search, and
- access to left and right neighbor ($\mathcal{O}(1)$ in our data structure).

In order to support these operations, we use a doubly linked list of rays, with an AVL search tree on top [AL62]. Figure 3.6 shows a graphical representation of our data structure. As the order of the rays along the sweepline is known at the time of insertion—see Algorithm 3.2 Line 12—the $\mathcal{O}(\log n)$-time search phase of a traditional AVL data structure can be omitted and new rays can be inserted in a bottom-up manner. However, this optimization requires the need for parent pointers in the tree. Since always two neighboring rays are inserted into the sweepline data structure at the same time, we implement a special insert operation for this case that only requires one rebalancing operation for both rays. For removal operations, the position of the ray within the sweepline is known as well—refer to Line 30. Thus, similar to insert operations, no search phase is required for removals and the operation can be

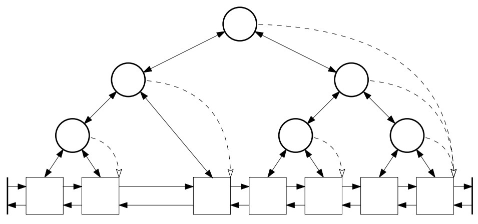


Figure 3.6 The sweepline data structure is a doubly linked list of rays with an AVL search tree on top. Additionally, each node has a pointer to the rightmost ray in its subtree (dashed)

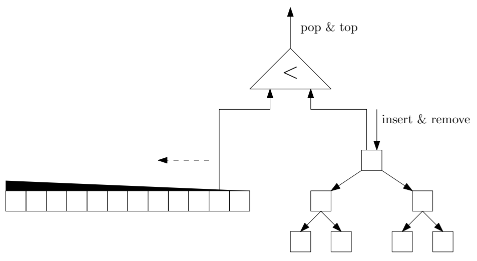


Figure 3.7 The priority queue consists of a *static*, sorted array of input points and a *dynamic*, addressable PQ for intersection and deletion events.

performed in a bottom-up manner. The algorithm always removes the two neighboring rays B_L and B_R and replaces them with B^*. B^* has the same left neighbor as B_L and the same right neighbor as B_R. Therefore, we can simply replace B_L with B^* in the data structure and just need to remove B_R, leading to merely one rebalancing operation.

The search for the enclosing region of a point p—Algorithm 3.2 Line 7—is performed by finding the first ray B_R, currently intersecting the sweepline, that has p to its right. This requires the evaluation of an oriented side of line predicate at each level of the tree. The left neighbor B_L of B_R, must have p to its left or on it. Therefore, B_L and B_R enclose p and $B_L^r = B_R^l = R_q$ gives the region p is contained in. To facilitate the search, each internal node of the tree needs to refer to the rightmost ray in its subtree. As rays are complex objects, we use pointers to the corresponding leaf to save memory. Given the expensive search operations, AVL trees—as strictly height-balancing trees—are preferable to data structures with weaker balancing guarantees, such as red-black trees [ŠT12].

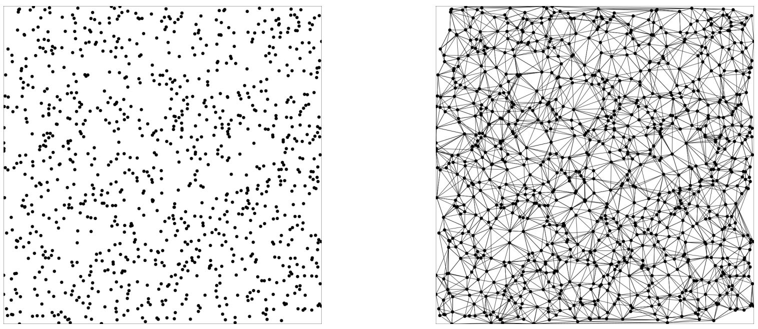
3.4.3 Priority Queue

The priority queue (PQ) Q is initialized with all input points at the beginning of the algorithm. During event processing, intersection and deletion events may be added and removed from Q, therefore requiring an addressable priority queue. The objects are ordered according to Equation (3.1), thus keys are (exact) numerical values. Our experiments show that, typically, for n input points, only about $\mathcal{O}(\sqrt{n})$ intersection and deletion events are in Q at any given step. Using the same PQ for all events would result in expensive dynamic PQ operations. As input point events are static in Q, we can use a two-part data structure as shown in Figure 3.7. Input point events are stored in an array—sorted by priority in Q—with a pointer to the smallest unprocessed element. Intersection and deletion events are stored in an actual addressable priority queue. We use an addressable binary heap for this part of the data structure. The TOP operation needs to compare the minimum element of the PQ with the element pointed to in the array and return the minimum of both. POP either performs a regular pop on the PQ or moves the pointer of the array to the next larger element. INSERT and REMOVE operations can access the PQ directly, as only this part of the data structure is dynamic. Thus, the actual dynamic PQ is much smaller resulting in more efficient SIFTDOWN and SIFTUP operations in the binary heap. The smaller heap size not only reduces the tree height but also makes the heap more cache friendly.

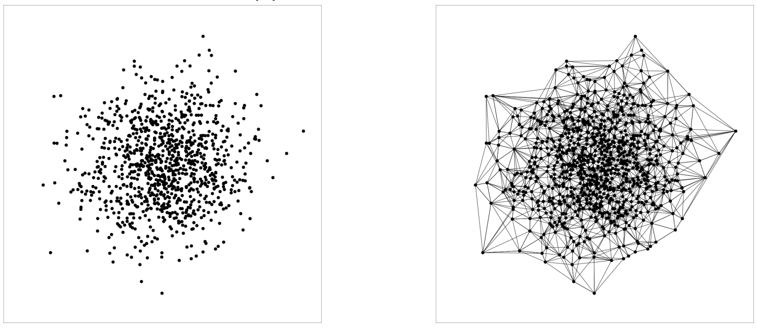
This optimization might be of interest for other algorithms that initialize their priority queue with all input points and only have a small number of dynamically added events in their priority queue at any given time. Note that the *total* number of processed intersection and deletion events can surpass the number of input points by far, however only a few of these events are in the PQ at the same time.

3.5 Evaluation

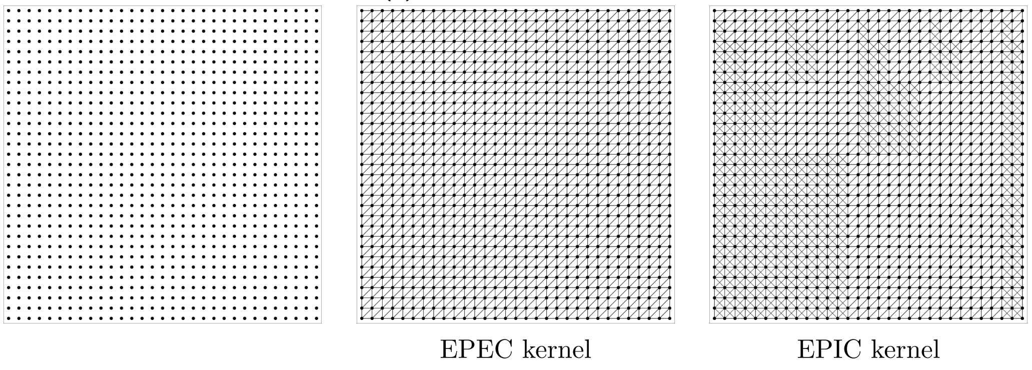
In this section, we evaluate our implementations of the Yao graph construction algorithms presented in Section 3.3 on a variety of datasets. As mentioned before, we are not aware of any previous implementations of Chang et al.'s Yao graph algorithm. Therefore, our main competitor is the Yao graph algorithm from the CGAL library's cone-based spanners package [STP22]. We are not aware of any other tuned implementations to construct Yao graphs as further competitors.



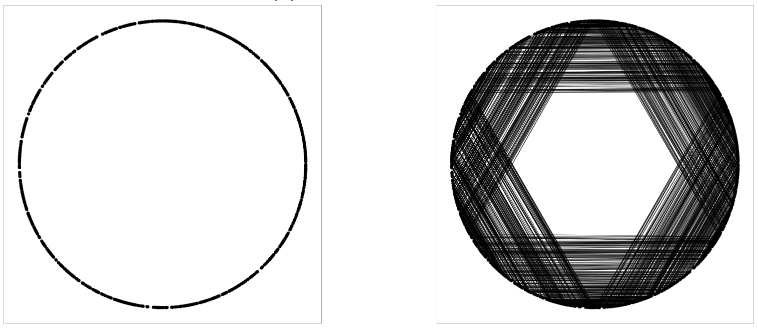
(a) Uniform distribution.



(b) Gaussian distribution.

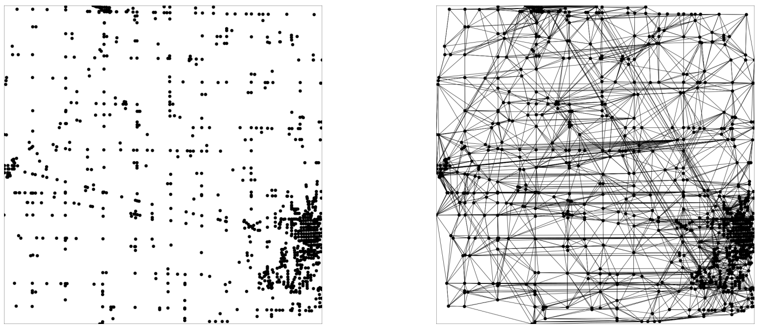


(c) Grid distribution.



(d) Circle distribution.

Figure 3.8 Input distributions for $n = 1000$ points and resulting Yao graph for $k = 6$. For the grid distribution, the resulting graphs from exact constructions and inexact constructions are shown.


(e) Road dataset.

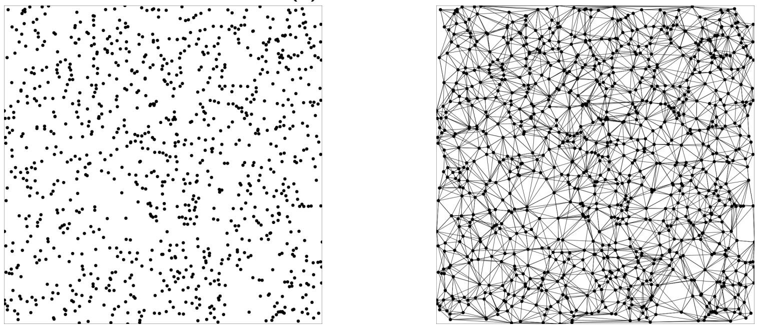
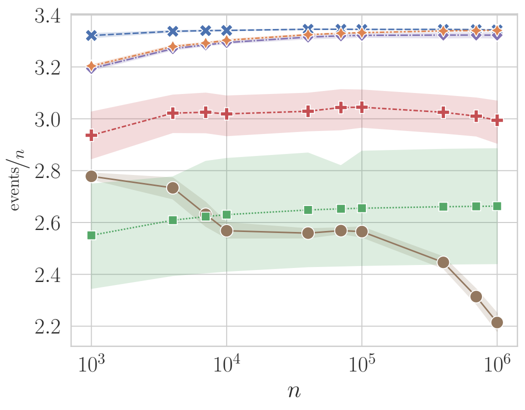

(f) Star dataset.

Figure 3.8 (cont.) Input distributions for $n = 1000$ points and resulting Yao graph for $k = 6$. For the grid distribution, the resulting graphs from exact constructions and inexact constructions are shown.

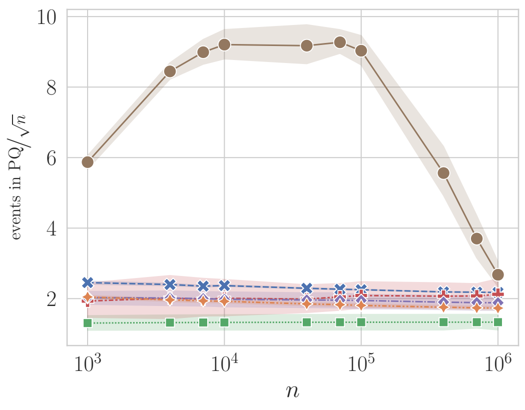
Experimental Setup. We test all algorithms on a variety of synthetic and real-world datasets. We use input point sets distributed uniformly and normally in the unit square, as well as points lying on the circumference of a circle and at the intersections of a grid—the former being a worst-case input for the grid algorithm, the latter being a bad case for numerical stability. Furthermore, two real-world datasets are used in our evaluation: intersections in road networks and star catalogs. As road networks we use graphs from the 9th DIMACS implementation challenge [DGJ09]. To generate a road network of a desired size n from the Full USA graph, we use a random location and grow a rectangular area around it until at least n nodes are within the area. US cities feature many points on a grid and therefore present a challenge for numerical stability. We furthermore use the Gaia DR2 star catalog [Gai18], which contains celestial positions for approximately 1.3 billion stars. We use a similar technique as for road networks to generate subgraphs of a desired size. Here, we grow a cube around a random starting location until it contains the desired number of stars. We then project all stars onto the xy-plane as two-dimensional input for our experiments. Figure 3.8 shows examples of our input datasets and the resulting Yao graphs.

We implemented all algorithms in our C++ framework YaoGraph, available on GitHub.[6] Our code was compiled using GCC 12.1.0 with CGAL version 5.0.2. The experiments were performed on a server with an Intel Xeon Gold 6314U CPU with 32 cores and 512 GiB of RAM, running Ubuntu 20.04 with kernel version 5.4.0. For all experiments we used three different random seeds and $k = 6$ unless otherwise specified.

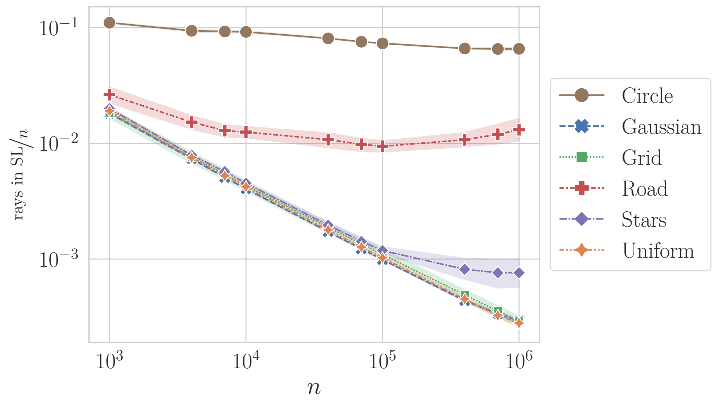
[6] `https://github.com/dfunke/YaoGraph`



(a) Total number of events processed.



(b) Maximum number of intersection and deletion events concurrently in PQ.

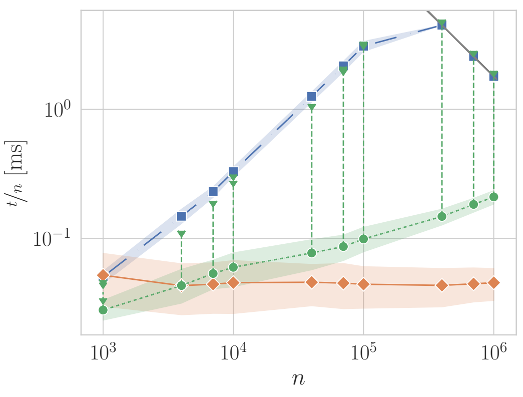


(c) Maximum number of rays in sweepline data structure.

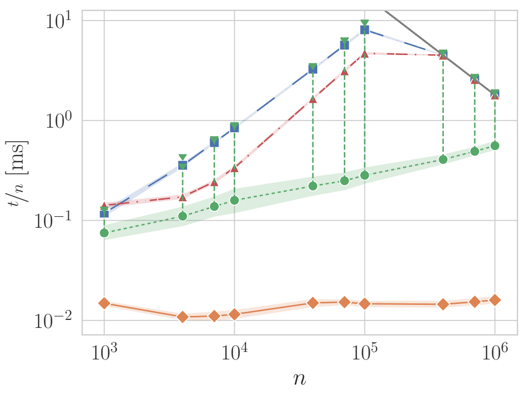
Figure 3.9 Statistics for varying input point distribution and point set sizes for $k = 6$ cones. Error bands give the variation over the different cones being calculated.

3.5.1 Algorithmic Metrics

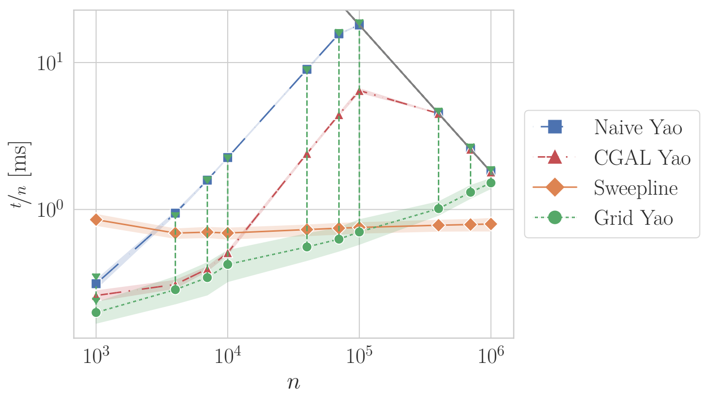
In this section we consider three statistics of the sweepline algorithm's execution: number of processed events as well as occupancy of the priority queue and sweepline data structure. Figure 3.9a shows the number of events processed per input point by the algorithm. Each input point event generates 2.3 intersection and/or deletion events on average, with very little variance with regard to input size and distribution—except for the grid distribution and road graphs. Both exhibit larger variance, depending on whether the cone's boundaries coincide with grid lines or not. Figure 3.9b shows the maximum number of intersection and deletion events that are in the priority queue at any given time during the algorithm execution. This number scales with $\mathcal{O}(\sqrt{n})$ for most studied inputs, which motivates our choice of the two-part priority queue as discussed in Section 3.4.3. The behavior of the circle distribution requires further investigation. The maximum number of rays in the sweepline data structure at any point during algorithm execution shows no clear scaling behavior, refer to Figure 3.9c. It scales with $\mathcal{O}(\sqrt{n})$ for most synthetic input sets, but approaches a constant fraction of the input size for the circle (≈10 %), road (≈1 %) and star (≈0.1 %) datasets.



(a) Inexact kernel.



(b) CGAL EPIC kernel.



(c) CGAL EPEC kernel.

Figure 3.10 Algorithm runtime experiments. Experiments over varying input sizes are performed with $k = 6$ cones. Error bands give the runtime variation over the different input point distributions. For the grid algorithm, the circle distribution is plotted separately with triangular markers. The gray line represents the time limit of 30 min per algorithm. Experiments over varying number of cones are with $n = 1 \times 10^5$ uniformly distributed input points.

3.5.2 Runtime Evaluation

In this section we discuss the results of our runtime experiments. Figures 3.10a–3.10c show the (scaled) running time of all algorithms, displaying variations due to input distributions as error bands. Detailed runtime plots for the different distributions are presented in Figure 3.12. Note that only the grid and the sweepline algorithm are sensitive to the input point distribution. As previously seen in Figure 3.9a, the number of processed events by the sweepline algorithm is relatively stable for all distributions. Therefore, only little variation is seen in the runtime of the algorithm. This also shows, that the size of the sweepline data structure has only negligible influence on the algorithm runtime, as no higher runtime is observed for the road or circle datasets. Our inexact kernel shows more runtime variation than CGAL's highly optimized kernels, mainly due to the grid distribution with its many points directly on cone boundaries. The sweepline algorithm clearly outperforms CGAL's Yao graph implementation. Furthermore, even though the sweepline algorithm requires much more involved computations, it is superior to the simple grid algorithm for non-exact constructions. Only for exact constructions, large inputs are required to negate the more

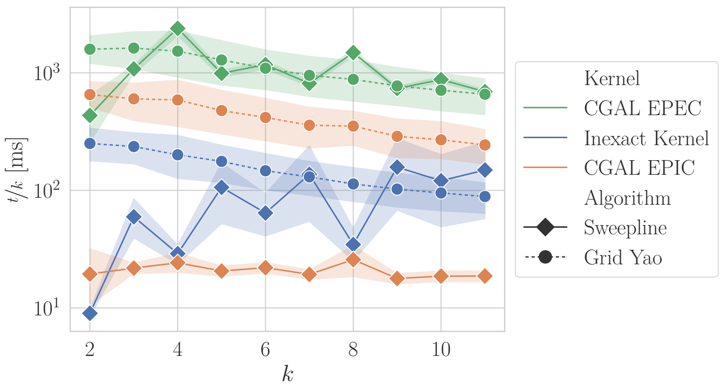


Figure 3.11 Algorithm runtime experiments. Experiments over varying number of cones are with $n = 1 \times 10^5$ uniformly distributed input points.

expensive operations of the sweepline algorithm. The exact construction kernel leads to a runtime overhead of 100 compared to the EPIC kernel. However, if points lie directly on cone boundaries, exact constructions are necessary to obtain correct results, as seen in Figure 3.8c. The data dependency is more pronounced for the grid algorithm, which performs well for most datasets but degenerates to the naive algorithm for the circle distribution, due to the many empty grid cells in the circle's interior.

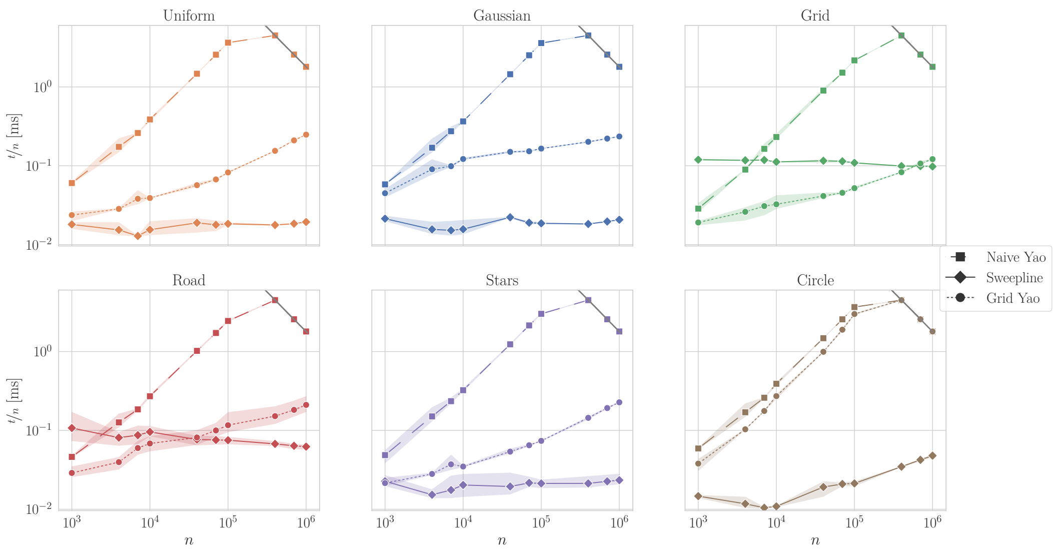
To compute a Yao graph with k cones, the sweepline algorithm requires k passes. This linear relationship can be seen in Figure 3.11. The grid algorithm has no dependency on k—except for the size of the neighborhood of a point. However, our experiments show that the runtime of the algorithm increases with increasing k. We attribute this to the fact that more grid cells need to be visited in order to settle all cones of a point p. As cones become narrower with increasing k, chances are higher that a specific cone of p has no points within a visited grid cell and thus further cells need to be visited. We did not perform these experiments with the naive algorithm or the CGAL algorithm, due to their long runtimes. CGAL's algorithm also requires one pass per cone, whereas the naive algorithm's runtime dependency on k is negligible.

Table 3.2 shows the impact of our engineered data structures on the runtime of the sweepline algorithm. The optimization of the AVL tree operations to avoid unnecessary rebalancing operations has the largest impact on the runtime. For larger inputs, the priority queue optimization would become more important.

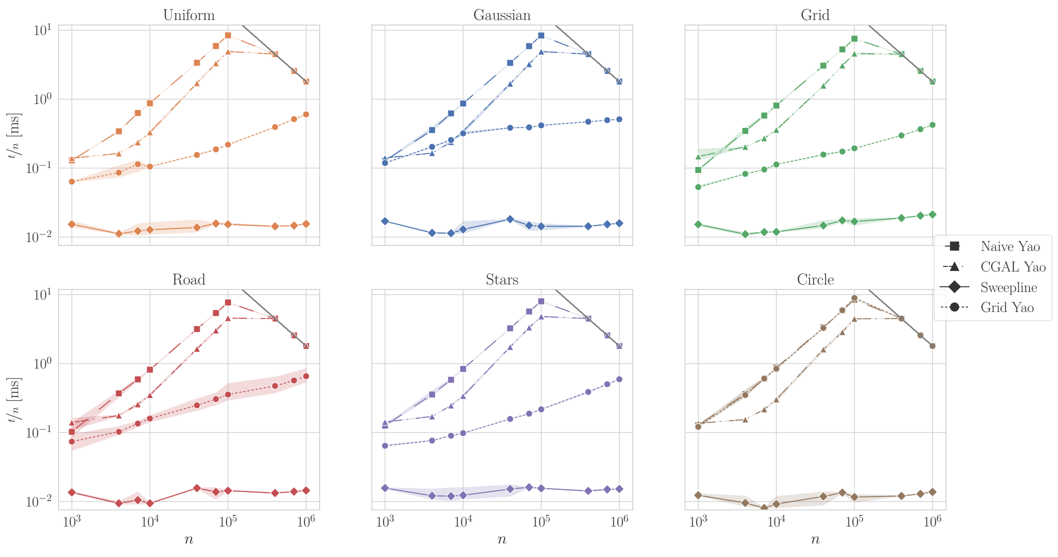
Table 3.2 Impact of our optimizations on the runtime of the sweepline algorithm for $n = 1 \times 10^4$ uniformly distributed input points and $k = 6$ cones.

Optimization	Runtime [s]	Improvement [s]	Improvement [%]
Baseline	1.940	–	–
Priority queue	1.779	0.161	8.30
Robin Hood hash table†)	1.593	0.186	10.46
AVL tree operations	1.296	0.297	18.64
Total	–	0.644	33.20

†) `https://github.com/martinus/robin-hood-hashing` (accessed 08-12-2023)

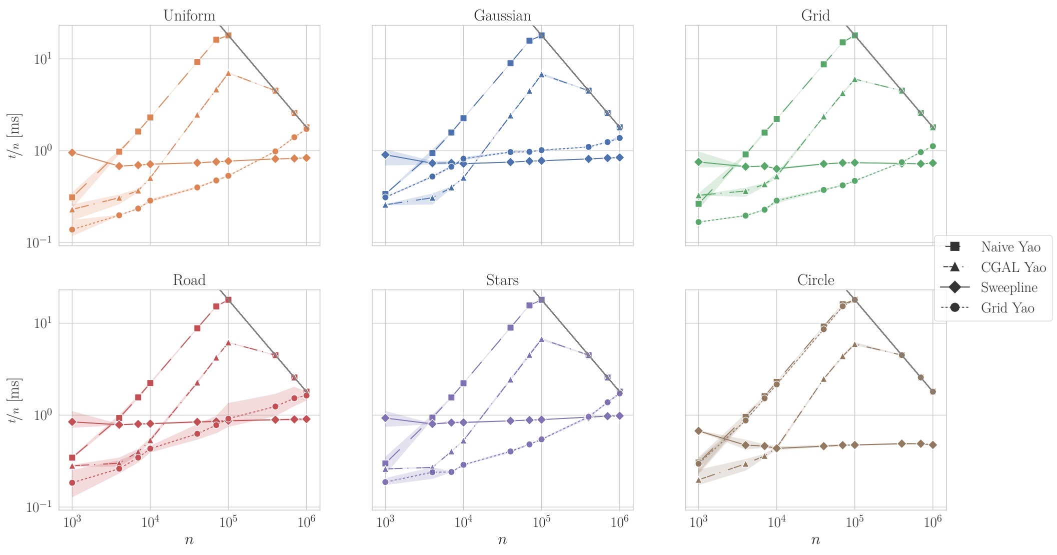


(a) Inexact kernel.



(b) CGAL EPIC kernel.

Figure 3.12 Algorithm runtime experiments. Experiments over varying input sizes are performed with $k = 6$ cones. The gray line represents the time limit of 30 min per algorithm.



(c) CGAL EPEC kernel.

Figure 3.12 (cont.) Algorithm runtime experiments. Experiments over varying input sizes are performed with $k = 6$ cones. The gray line represents the time limit of 30 min per algorithm.

3.6 Conclusions

We present the—to the best of our knowledge—first implementation of Chang et al.'s optimal $\mathcal{O}(n \log n)$-time Yao graph algorithm. Our implementation uses carefully engineered data structures and algorithmic operations and outperforms current publicly available Yao graph implementations—particularly CGAL's cone-based spanners package—by at least an order of magnitude. We furthermore present a very simple grid-based Yao graph algorithm that also outperforms CGAL's implementation, but is inferior to Chang et al.'s algorithm for larger inputs. However, the algorithm could be further improved by using a precomputed mapping of the grid neighborhood to cones, in order to only visit grid cells that can contain points in hitherto unsettled cones. Moreover, the algorithm is trivially parallelizable over the input points, whereas Chang et al.'s algorithm can only be easily parallelized over the k cones. The parallelization within one sweepline pass remains for future work.

4 Mountain Isolation

Summary: One established metric to classify the significance of a mountain peak is its isolation. It specifies the distance between a peak and the closest point of higher elevation. Peaks with high isolation dominate their surroundings and provide a nice all-round view from the top. With the availability of worldwide digital elevation models (DEMs), the isolation of all mountain peaks can be computed algorithmically. Previous algorithms run in worst-case time that is quadratic in the DEM size. We present a novel sweep-plane algorithm that runs in time $\mathcal{O}(n \log n + pT_{\mathrm{NN}})$ *where* n *is the number of sample points in the DEM,* p *the number of considered peaks and* T_{NN} *the time for a two-dimensional nearest neighbor query in an appropriate geometric search tree. We refine this to a two-level approach that has high locality and good parallel scalability. Our implementation reduces the time for calculating the isolation of every peak on Earth from hours to minutes while improving precision.*

Attribution: This chapter is taken mostly verbatim from [FHS23a] and its accompanying technical report [FHS23b]. The author of this dissertation was the main author and contributor of the paper. Nicolai Hüning provided the implementation and performed experiments for his Bachelor thesis [Hün22]. Additionally, he authored parts of the evaluation. Peter Sanders contributed the algorithmic analysis and provided editing.

4.1 Introduction

High-resolution digital elevation models (DEMs) are an interesting example of large datasets with a big potential for applications—such as flood risk analysis or flight and urban planning—but equally big challenges due to their enormous size. For example, WorldDEM, provided by the TanDEM-X mission, covers the entire globe with a resolution of 0.4 arcseconds and is currently the highest-resolution worldwide DEM available [VS20]. It consists of $6 \cdot 10^{12}$ individual sample points and amounts to approximately 25 TB of data. Modern LIDAR technology allows for less than $1\,\mathrm{m}^2$ samples, resulting in more than 300 TB of data for Earth's land surface. The algorithm engineering community has greatly contributed to unlocking the potential of DEMs by developing scalable algorithms for features such as contour lines, watersheds, and flooding risks [Aga+08; dBT11; RLA17]. This dissertation continues this line of research by studying the isolation of mountain peaks which is a highly nonlocal feature and requires us to deal with Earth's complicated (not-quite spherical) shape.

A mountain's significance is typically characterized using three properties—elevation, isolation, and prominence [Gri04]. Whereas elevation is a fundamental property, isolation and prominence are derived measures. Isolation—also referred to as dominance—measures the distance along the sea-level surface of the Earth between the peak and the closest point of higher elevation, known as the *isolation limit point* (ILP). Prominence measures the minimum difference in elevation from a peak and the lowest point on a path to reach higher ground—called the *key col*. Refer to Figure 4.1 for an illustration.

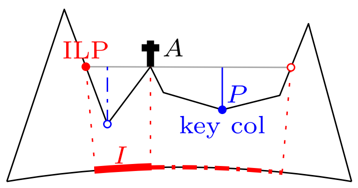


Figure 4.1 Illustration of a mountain A's isolation I and prominence P. The isolation is determined by the ILP (red) and is measured along the sea-level surface of the Earth. The prominence is defined by the *key col* (blue) and is the difference in elevation between it and A.

Whereas previously both measures had to be determined manually by laboriously studying topographic maps, they can now be computed algorithmically using DEMs. Kirmse and de Ferranti [KdF17] present the current state-of-the-art to compute both measures. Their algorithm determines a peak's isolation by searching in concentric rectangles of increasing size around the peak for higher ground. To determine the isolation of every peak on Earth, their algorithm has a worst-case running time of $\mathcal{O}(n^2)$, with n being the number of sample points in the DEM. For high-resolution DEMs of Earth and other celestial bodies, e.g., the Moon [Bar+16] or Mars [Gwi+10], algorithms with better scalability are needed. Also, continuously updated digital elevation models (CUDEMs), such as NOAA's DEM of North America's coastal regions [Ama+23], require efficient algorithms for frequent reprocessing. In particular, parallel and external algorithms are required.

Contribution and Outline After presenting basic concepts in Section 4.2 and related work in Section 4.3 we describe the main algorithmic result in Section 4.4. We begin with a sequential algorithm that sweeps Earth's surface top-down with a sweep-plane storing the points having surfaces at that elevation. Processing a peak then amounts to a nearest neighbor query in the sweep-plane data structure. This results in an algorithm with running time $\mathcal{O}(n \log n + pT_{\text{NN}})$ where n is the input size, p the number of considered peaks and T_{NN} the time for a two-dimensional nearest neighbor query in an appropriate geometric search tree. To make this more scalable, we then develop an algorithm working on the natural hierarchy of the data which is specified in *tiles*. This algorithm performs most of its work in two scans of the data which can work independently and in parallel tile-by-tile. Only the highest points in each tile need to be processed in an intermediate global phase. Each of these three phases has a structure similar to the simple sequential algorithm.

In Section 4.5 we explain how two-dimensional geometric search trees can be adapted to work on the surface of the Earth by deriving the required geometric predicates. After outlining implementation details in Section 4.6, we evaluate our approach using the largest publicly available DEM data in Section 4.7.

4.2 Preliminaries

In this section we briefly discuss basic concepts of spherical geometry, including two distance measures to approximate distances along the surface of the Earth. Furthermore, we introduce key aspects of digital elevation models and review state-of-the-art models.

Spherical Geometry. Most planets can generally be approximated by spheres. In the geographic coordinate system, a point $p = (\phi, \lambda)$ on the surface of a sphere is identified by its latitude ϕ and longitude λ. Latitude describes p's north-south location, measured from the equator, $\phi \in [-90°, 90°]$ with negative values south of the equator. Longitude describes p's east-west location, measured from the prime meridian through Greenwich, $\lambda \in (-180°, 180°]$ with negative values west of the prime meridian.

For close distances, Earth's surface is sufficiently flat to use planar Euclidean distances between points. For longer distances, spherical distance calculations are necessary, which are computationally more expensive. On the surface of the sphere, two points are always connected by at least two great circle segments. The shortest such segment, the *geodesic*, can be computed for points $p_1 = (\lambda_1, \phi_1)$ and $p_2 = (\lambda_2, \phi_2)$ with sphere radius R according to

$$\begin{aligned} \alpha &= \sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \sin^2\left(\frac{\lambda_1 - \lambda_2}{2}\right)\cos(\lambda_1)\cos(\lambda_2) \\ d(p_1, p_2) &= R\tan^{-1}\left(\frac{\sqrt{\alpha}}{\sqrt{(1-\alpha)}}\right) \end{aligned} \tag{4.1}$$

As Earth and most other planets are not perfect spheres, ellipsoids are a more accurate approximation of their shape. The World Geodetic System (WGS) [Nat14] defines such an ellipsoidal approximation of the Earth, requiring the following—even more computationally expensive—distance formula:

$$\begin{aligned} \Delta\lambda &= (\lambda_2 - \lambda_1)/2 \qquad \Delta\phi = (\phi_2 - \phi_1)/2 \qquad \Lambda = (\lambda_2 + \lambda_1)/2 \\ s &= \sin^2\Delta\phi \cdot \cos^2\Delta\lambda + \cos^2\Lambda \cdot \sin^2\Delta\lambda \\ c &= \cos^2\Delta\phi \cdot \cos^2\Delta\lambda + \sin^2\Lambda \cdot \sin^2\Delta\lambda \\ w &= \tan^{-1}(\sqrt{s}/\sqrt{c}) \qquad r = \frac{\sqrt{sc}}{w} \\ d(p_1, p_2) &= 2aw\left(1 + f\frac{3r-1}{2c}\sin^2\Lambda \cdot \cos^2\Delta\phi - f\frac{3r+1}{2s} \cdot \sin^2\Delta\phi \cdot \cos^2\Lambda\right) \end{aligned} \tag{4.2}$$

with a and f denoting the equatorial radius and flattening of the WGS84 ellipsoid [Nat14].

Digital Elevation Models. Digital elevation models (DEMs) have become one of the most important tools to analyze Earth's surface in geographic information systems. They represent Earth's surface by providing elevation measurements on a grid of sample points. The data is mostly stored in files of 1 square degree of coverage each, called tiles. A tile is addressed by its smallest latitude and longitude. In order to enable seamless processing of several tiles, each tile stores one sample row/column overlap with its neighbors. The resolution of DEMs is given by the length of one sample at the equator in arcseconds (″).

State-of-the-art worldwide DEMs, such as WorldDEM, provide a resolution of 0.4″ spacing between sample points—about 12 m at the equator [VS20]. Local DEMs, such as the swissALTI^{3D}, even have a resolution of only 0.5 m sample point spacing [Bun22]. Freely available DEMs, such as the Shuttle Radar Topography Mission (SRTM) DEM [Jet19], provide a resolution of 3″—about 90 m at the equator—and an absolute vertical height error of no more than 16 m for 90 % of the data [Rod+05]. Unfortunately, it covers only areas between 60° North and 56° South and contains large void areas, especially in mountainous regions, which are of particular interest for us. In a laborious process, de Ferranti [dFer11] fused raw SRTM data with other publicly available datasets [Ala08; Tac+11] and digitized topographic maps to create a worldwide, void-free DEM. Figure 4.2 illustrates the difference between raw SRTM data and the viewfinderpanoramas DEM.

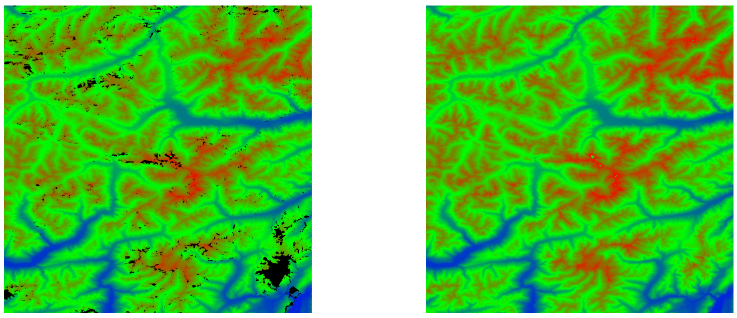


Figure 4.2 Comparison of tile $46°N$ $10°E$ (Ötztal and Ortler Alps). Left raw data from NASA's SRTM with voids (black), right enhanced data by viewfinderpanoramas.org.

4.3 Related Work

Graff et al. [GU93] are the first to use DEM data to classify terrain into mounts, plains, basins or flats. As there is no definitive definition of these terrain features, several methods for terrain classification are studied in the literature [TMF19], including fuzzy logic [FW98; FWC04] or, more recently, deep learning [Tor+18].

Prominence has received most of the attention when it comes to algorithmic computation of mountain metrics [Hel05; KdF17]. Kirmse and de Ferranti [KdF17] present the current state-of-the-art regarding isolation and prominence computation. As their main focus lies on the prominence calculation, they present a rather simple $\mathcal{O}(n^2)$-time isolation calculation algorithm.

In their algorithm they first calculate potential peaks which are samples that are at least as high as their eight neighboring samples. Afterwards, a search for the closest higher ground (ILP) is conducted, where, centered on each peak, concentric rectangles of increasing size are checked to find a sample with higher elevation. Since the closest higher ground of a peak could also be in a neighboring tile, these must be checked as well. If a higher ground was found, the distance to this sample is used to constrain the search in neighboring tiles. If not, tiles in increasing rectangles around the peak-tile are checked until an ILP is found or the complete world has been checked. Before searching a neighboring tile, the maximum elevation of the tile is checked. When it is smaller than the peak elevation, the tile can be ignored. Tiles and their maximum elevation are cached, because they need to be loaded rather frequently. Inside the tile that contains the peak, planar Euclidean distance approximations are used to find an ILP, for neighboring tiles, distances are computed according to the spherical distance function. Because a majority of peaks have a small isolation, the ILP is often within the same tile as the peak, so mostly planar Euclidean distance approximations are used to determine a peak's ILP. This reduces computational costs but also the accuracy for peaks with small isolation. Only in a final step before output is the distance between a peak and its determined ILP calculated using the precise, but expensive, ellipsoid distance function. Kirmse and de Ferranti's algorithm is unnamed, we refer to it as ConcIso for brevity.

Sweepline algorithms are introduced by Bentley and Ottmann [BO79] to compute line segment intersections. The technique is generalized by Anagnostou et al. to three-dimensional space [AGP90]. Sweepline algorithms have been applied to various geometric problems in two- and three-dimensional space, such as computing Voronoi diagrams [For87] or route mining [YLX20]. R-Trees are often used in spatial databases and have been adapted to geodetic distance computations by Schubert et al. [SZK13].

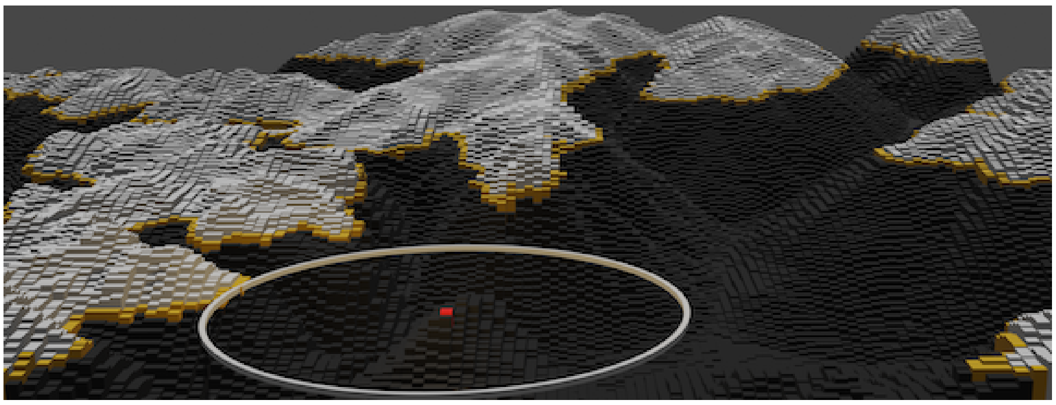


Figure 4.3 Illustration of DEM grid and sweep-plane algorithm by using a voxel representation. The red point represents the current peak. Orange DEM points are active and contained in the sweep-plane data structure. White points are inactive and already removed from the data structure. Black points have not yet been processed.

4.4 Algorithms

In this section we present our novel sweep-plane algorithm to calculate the isolation of mountain peaks. The algorithm takes as input the search area—a quadrilateral A defined by its north-west and south-east corners—as well as the DEM data. We will first present a single-sweep algorithm that processes the entire search area in one sweep-plane pass. Subsequently, we present a scalable three-pass algorithm that reads the DEM-tiles of A twice and can process tiles in parallel.

4.4.1 Single-Sweep Algorithm

To determine the isolation of a peak, its closest point with higher elevation—the isolation limit point (ILP)—needs to be found. Given the search area A, each sample point p of the DEM within A has two associated events: an *insert* event at p's elevation and a *remove* event at the elevation of its lowest immediate (NESW) neighbor. Additionally, if a point is the high-point of its 3x3 neighborhood it is associated with a *peak* event at its elevation. We describe the peak detection routine in more detail in Section 4.6.2. All events are created at the beginning of the algorithm and can therefore be added to a static sequence sorted by descending elevation. Peak events are processed before other events at the same elevation.

The sweep-plane then moves downward from the highest sample point to the lowest and traces the contour lines of the terrain, refer to Figure 4.3. The sweep-plane data structure SL is a two-dimensional geometric search tree that maintains a set of currently *active* points. A sample point p becomes active at its insert event, when it is swept by the sweep-plane and inserted into SL. Point p becomes *inactive* and is removed from SL at its remove event, at which point its lowest neighboring point is activated and thus all of p's neighboring points are either active or have already been deactivated. When a peak event for sample point p is processed, a nearest neighbor query for the closest active sample point to p in SL is performed. The returned point w is the ILP for the peak: Since w is active and peaks are processed before other events, w must be higher than p. There cannot be any closer ILP as points activated later are not higher than p and since higher points v that are already deactivated are surrounded by points that are all higher than p. At least one of them must be closer to p than v.

Analysis. Given a DEM with n points, we can detect its p peaks in time $\mathcal{O}(n)$. The resulting $2n+p$ events can be sorted in $\mathcal{O}(n \log n)$ time by elevation. Insertion and removal operations take $\mathcal{O}(\log n)$ worst-case time in several geometric search trees [Ben75; FB74; Smi00]. Nearest neighbor search complexity is more complicated and still an open problem in many respects. Therefore, we describe it abstractly as T_{NN}. Simple trees used in practice such as k-D trees [Ben75] or quad-trees [FB74] achieve logarithmic time "on average". Chan [Cha20] presents a theoretical data structure based on shallow cuttings that achieves a $\mathcal{O}(\log^4 n)$ worst-case time bound. Approximate queries within a factor of $1+\epsilon$ are possible in time $\mathcal{O}(\log(n)/\epsilon^2)$ [Ary+98]. Overall, we get the claimed time $\mathcal{O}(n \log n + pT_{\mathrm{NN}})$.

4.4.2 Scalable Multi-Pass Algorithm

High-resolution DEMs are massive data sets of currently up to 25 TB that require scalable algorithms. The algorithm described in the previous section can be parallelized to some extent but its sweeping character limits parallelism. Moreover, a geometric search tree covering the entire Earth could get quite large. We therefore develop a two-level algorithm that allows for more coarse-grained parallelism and better locality. We adopt the natural hierarchy of the input data using tiles of a fixed area but note that grouping several tiles into a larger one or splitting a tile into smaller ones would be possible in principle. Furthermore, a more general multi-level algorithm could be developed using a similar approach.

Multi-level algorithms start at the finest level to extract information for global[1] processing at coarser level. The global results are then passed down to compute the final solution. In our two-level algorithm this implies that we have two passes reading the DEM tiles from external memory while a single global pass works with simple per-tile information. The first (*bounding*) and last (*finalization*) pass can work in parallel on each tile. The global (*high-point*) pass works in internal memory and is also parallelized. The first two passes establish a global Tile-Peak map that stores for each tile which peaks *can* have an ILP candidate in it. The third pass processes these assigned peaks for a tile and determines their ILPs. All passes follow a very similar structure to our single-sweep algorithm from Section 4.4.1. They are described in detail in the following. Additionally, Figure 4.4 provides an overview of our algorithm.

Bounding Pass. The purpose of the first pass is to establish an upper bound on the isolation of a peak and therefore limit the number of tiles that need to be searched for its ILP in the finalization pass. To establish this upper bound, we find a tile-local ILP candidate for each peak using our single-sweep algorithm (gray arrows in Figure 4.4). Only the highest point in each tile will not have a tile-local ILP candidate and is treated separately in the high-point pass (colored triangles in Figure 4.4). Given the upper bound on the isolation of a peak p, we can assign p to all tiles within radius of the upper bound that could contain a closer ILP for p. In the third pass, these neighboring tiles then need to process p in order to find ILP candidates within them. To link peaks to tiles for processing in the third pass, we build a Tile-Peak map that stores for each tile a list of peaks that could have an ILP within it.

High-Point Pass. As there is no local ILP for the high-point h of each tile, we address high-points separately in the second pass. This pass uses only one type of event—a combination of insert and peak event from the other passes—and processes only one point per tile,

[1] Pun intended.

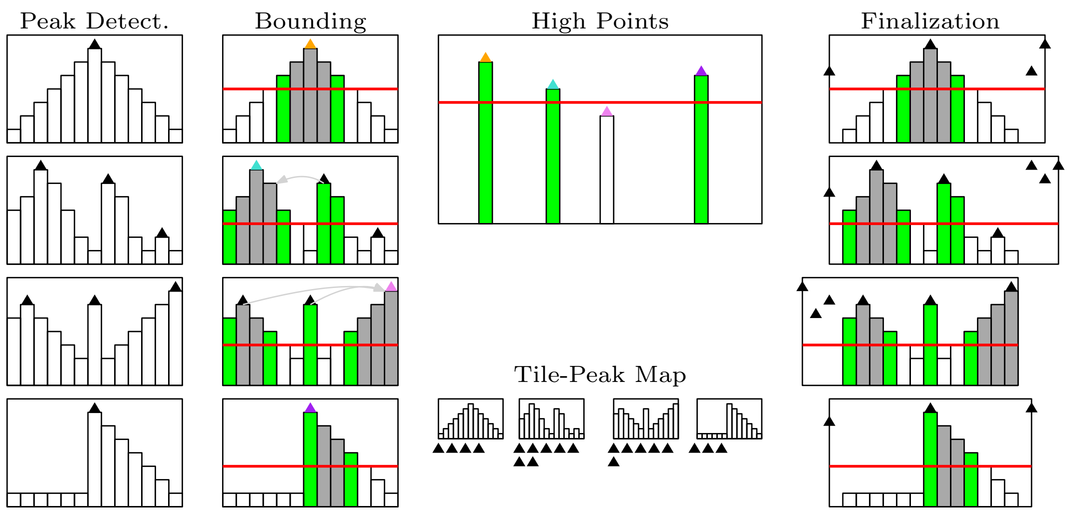


Figure 4.4 Overview of our scalable multi-pass algorithm for four tiles in two dimensions. Peaks are detected for each tile with the peak finding algorithm of ConcIso. The bounding pass determines an upper bound on the isolation for each peak and assigns it to tiles that could contain closer ILP candidates in the Tile-Peak map. The high-points of each tile without a local upper bound are processed by a separate pass considering only the high-points. In the finalization pass, all peaks assigned to a tile are checked for an ILP candidate within the tile. The final determination of the closest ILP out of the found candidates for each peak is not depicted. For the sweep-plane passes, the sweep-plane is pictured in red, green points are active in the sweep-plane data structure, gray points are inactive and hollow points have not yet been processed. Peaks are marked by triangles and their tile-local ILP candidate is denoted by a gray arrow. The high-point of each tile is marked by a different color triangle for each tile.

namely their high-points. For each high-point h, we perform a nearest neighbor query in the sweep-plane data structure for h's closest higher point p. The distance between h and p is an upper bound on h's isolation. The sweep-plane data structure is then traversed again to find all tiles whose closest point to h is at least as close to h as p and h is linked to these tiles in the Tile-Peak map. Finally, h is inserted into the sweep-plane data structure and the next lower high-point is processed.

Finalization Pass. In the third pass, each tile is processed again by a sweep-plane algorithm similar to the single-sweep variant. The peak events of this pass are all peaks that have been assigned to the tile in the Tile-Peak map. For each assigned peak, the ILP candidate within the processed tile is determined. After all tiles have been processed by the third pass, each peak has as many ILP candidates as tiles it has been linked to. Thus, in a final step, for each peak the closest found ILP candidate is set as the true ILP of the peak.

Algorithmic Details and Analysis Sketch. Let us first look at the total **work** performed to process A tiles containing n sample points overall. The bounding pass performs a similar amount of work as the global algorithm except that it defers nonlocal nearest neighbor searches to the subsequent passes. More precisely, the high-points of each tile are deferred to the high-point pass while other peaks are deferred to neighboring tiles within the upper bound of their tile-local ILP candidate. On average, there is a constant number of such neighboring tiles, as the upper bound on the isolation cannot exceed the length of the diagonal of a tile.[2]

The high-point pass potentially defers nearest neighbor search work for the highest point of each tile to a potentially large number of tiles. However, this is not so different from what a search-tree based nearest neighbor search in a global tree data structure would do. Our two-level algorithm can be viewed as a vertically split (quad-)tree algorithm where updates and nearest neighbor searches are reordered to improve locality. The overall amount of work done is quite similar though.

The finalization pass can be viewed as completing the deferred nearest neighbor searches. Thus, the main overhead of the two-pass algorithm compared to the global algorithm is that the data is swept through search trees twice. We mitigate this effect by building only a coarse search tree in the bounding pass. This has no negative effect on precision as the bounding pass is only needed to identify the tiles where an ILP can be. Only the finalization pass computes the actual ILPs.

Let us now look at **I/O-costs**. Assume one tile and the high-points fit in internal memory. This is similar to a "semi-external"-assumption used in previous algorithm engineering papers on DEM processing, e.g., [Aga+08]. The I/O volume of our two pass algorithm is dominated by reading all tiles twice.[3] This is actually better than an external-memory implementation of the global algorithm as that algorithm would have to sort the data by altitude before being able to scan the input in the right order for the sweep. Even using pipelining [DKS08], sorting would require at least two reading and one writing pass over the DEM data.

Finally, let us consider **parallelization**. Bounding and finalization passes can work on A tiles in parallel. In order to also parallelize the high-point pass, we implement a variant that uses a static search tree with subtrees augmented by the maximum elevation occurring in a subtree. Then the A nearest dominating point searches can be done in parallel. See Section 4.6.2 for details.

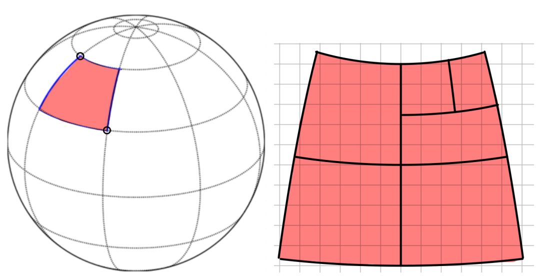
4.5 Predicates for Search Trees on Spherical Surfaces

The sweep-plane data structure needs to be a dynamic data structures that supports efficient nearest neighbor queries. Space-partitioning trees such as k-D trees or quad-trees are well-suited data structures for this application [Ben75; FB74]. These trees recursively divide the input space into smaller and smaller blocks. For a spherical surface, the space is divided into quadrilaterals which are aligned with latitude and longitude, refer to Figure 4.5a. A quadrilateral is defined by its north-west and south-east corners. Each quadrilateral can then be further subdivided into smaller quadrilaterals. The root of a space-partitioning tree covers the entire input area.

Our sweep-plane algorithm requires two types of queries in these geometric search trees: a) whether a given point p lies inside a quadrilateral Q and b) the shortest distance between point p and quadrilateral Q. The former can be easily answered by comparing p's latitude and longitude with the coordinates of Q's north-east and south-west corner. For the latter,

[2] Near the poles, there are many candidate tiles that may be closer than a local ILP candidate. However, averaged over all tiles, this effect is not large. It is interesting to note though that this is an artifact of a pseudo-high longitudinal resolution that is not reflected in the actual precision of the sensors but stems from artificially mapping the data using a Mercator projection. Meshes with more uniform cells are possible, for example the icosahedral grid used in some modern climate/weather models [Jun+22].

[3] In comparison, the Tile-Peak map has negligible data volume and can be handled in an I/O-efficient way by observing that the first two passes only insert to it and the third pass only reads it tile by tile. Thus, we can first buffer its data in an external log which is sorted by tile before the finalization pass.


(a)

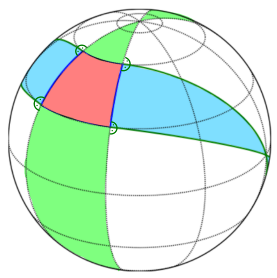
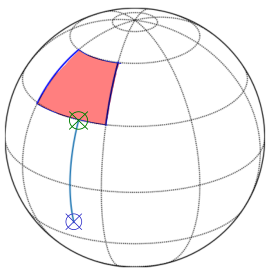
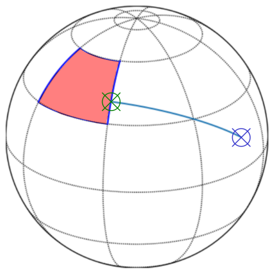

(b)

Figure 4.5 Representation of a latitude-longitude aligned quadrilateral and areas for the different min-distance cases between a quadrilateral and a point.


(a) Case 2.


(b) Case 3.

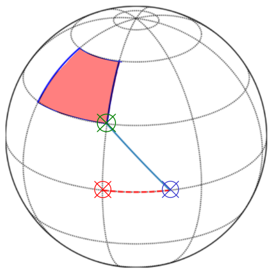

(c) Case 4.

Figure 4.6 Example configurations for the minimum distance between a point and a quadrilateral.

there are four configurations of p and Q that need to be considered, refer to Figure 4.5b:

1. $p \in Q$ (red area),
2. p is between the longitude lines of Q (green area),
3. p is between the four great circles through the corners of Q, which are perpendicular to the longitude edges of Q (blue area),
4. all other positions (white area).

For case 1 the distance is zero. In case 2, the closest point $s \in Q$ to p is on the intersection between the longitude line of p and one of the latitude-edges of the quadrilateral, as the shortest distance between two latitudes is along the longitude lines—also refer to Figure 4.6a. The shortest distance of p to Q is thus the shorter distance between p and the north and south latitude of Q.

In the remaining cases, s must lie on one of the longitude lines of the quadrilateral. To determine the longitude edge closest to p, we calculate the center longitude of Q and rotate it to align with the meridian. We rotate p by the same amount. If the longitude of p is now positive, the west longitude-edge of Q is closer to p, otherwise the east one.[4] Having determined the closest longitude edge, we can now calculate point s using linear algebra in Euclidean space. For this, point p and the edge-points defining the closest longitude edge, l_N and l_S, are transformed according to

$$x := \cos(\mathrm{lat}(x))\cos(\mathrm{long}(x)) \qquad y := \cos(\mathrm{lat}(x))\sin(\mathrm{long}(x)) \qquad z := \sin(\mathrm{lat}(x)) \tag{4.3}$$

[4] This is possible because we split the Earth at the antimeridian. Therefore, the western longitude of Q is always smaller than the eastern one.

Note that this transformation assumes Earth to be a sphere. For a more accurate transformation assuming Earth to be an ellipsoid refer to [Nat14]. Now, the closest point s to p can be determined according to

$$A := l_S \times l_N \qquad B := p \times A \qquad S := A \times B \tag{4.4}$$

Observe that S is normalized, because Equation (4.3) produces normalized vectors and Equation (4.4) uses only cross products. The idea behind these formulae is the following: A is the plane of the great circle defined by the edge-points of the longitude edge, l_N and l_S. B is the plane of the great circle that is perpendicular to A and goes through the point p. Lastly, the intersection between these two great circles is calculated (S), yielding the geodesic between s and p, that is perpendicular to the longitude circle and thus the shortest possible one. S is now transferred back to longitude and latitude using

$$\mathrm{lat}(s) := \arcsin(z) \qquad \mathrm{long}(s) := \mathrm{arctan}_2(x, y)$$

This yields point s on the longitude-edge of Q that is closest to p. If the latitude of s is between the top and bottom latitude of Q, s is the point with the shortest distance to p in Q (case 3, Figure 4.6b). Otherwise, one of the corners is the point with the shortest distance to p (case 4, Figure 4.6c).

Given these primitives, insert and query operations on space-partitioning trees for spherical surfaces are identical to the ones for Euclidean space.

4.6 Implementation Details

We implement our new sweep-plane algorithm in the MOUNTAINS C++ framework by Kirmse and de Ferranti [KdF17].[5] The framework provides essential functionalities for the work with tiled DEM data, such as data loading and conversion, peak discovery and distance computations. Our code is available on GitHub.[6] In the following we provide details of our implementation.

4.6.1 Data Structures

We implement a fully dynamic k-D tree, that supports insert and remove operations in $\mathcal{O}(\log n)$ worst-case time and nearest neighbor searches in $\mathcal{O}(\log n)$ expected time [FBF77] using the predicates described in the previous section.[7] Points are stored in the leaves of the tree, which have a fixed capacity of C points. On exceeding capacity C, a leaf is split according to a center-split policy along the longer side of the quadrilateral. The first points inserted into the tree often belong to the highest peak in a tile and are thus close to each other. In order to prevent the tree from degenerating, we pre-build the first k levels in a quad-tree-like manner. Our experiments show $k = 4$ to be a good choice. To improve cache-efficiency, tree nodes are allocated in blocks and are re-used after deletion.

The Tile-Peak map is implemented as an internal memory hash map with the latitude and longitude of a tile as key and a list of peaks as value.

[5] `https://github.com/akirmse/mountains` (accessed 08-12-2023)
[6] `https://github.com/dfunke/mountains`
[7] We also adapted a quad-tree using our predicates, which was, however, outperformed by the k-D tree in our experiments.

4.6.2 Algorithm

Given the search area A, all contained DEM tiles can be processed in parallel. We use a work queue and thread pool to distribute the computations among the available processing elements. The Tile-Peak map is initialized in advance with all tiles. To sort the events of a pass, we use the efficient sorting algorithm ips4o of Axtmann et al. [Axt+17].

Bounding Pass. We use the peak detection algorithm of Kirmse and de Ferranti [KdF17, Sec. 2.2] to find all peaks within a tile. It considers all points to be peaks that have eight neighboring points of lower or equal elevation. If a peak consists of several equally high sample points, only a single one is added to the set of peaks.[8] Since only an upper bound is calculated during the bounding pass, we down-sample the resolution of the DEM after peak detection. Since all peaks are within the tile which is processed, distances are rather small and can be approximated using planar Euclidean geometry during the nearest neighbor search. The upper bound is computed using the spherical distance function according to Equation (4.1). If the determined upper bound on the isolation of a peak is below a threshold $I_{\min}$ we discard the peak as insignificant. Peaks with an upper bound above $I_{\min}$ are added to the Tile-Peak map for processing in the finalization pass as described in Section 4.4.2.

High-Point Pass. While processing the tiles in the bounding pass, we build a geometric search tree T that partitions the entire search area down to the tile level. Internal nodes of T save the highest elevation in their subtree. After all tiles have been processed, we can use this information to efficiently determine upper bounds on the isolation of the high-points of each tile. For each high-point h, we find the closest tile containing a point of higher elevation than h in search tree T. The maximum distance between h and any point within the found tile serves as an upper bound on h's isolation. Given this upper bound we can add h to all tiles containing potential ILPs in the Tile-Peak map. This approach is trivially parallelizable over the number of high-points in the search area.

Finalization Pass. In this pass we use the full resolution of the input DEM. For each tile, the peaks processed in this pass are the ones that are assigned to it in the Tile-Peak map. We use ellipsoid distance computations according to Equation (4.2).

4.7 Evaluation

In this section we evaluate our novel sweep-plane algorithm to calculate the isolation of mountain peaks, which we named SweepIso. We evaluate it with regard to runtime behavior and solution quality and compare it against ConcIso from Kirmse and de Ferranti [KdF17].

Experimental Setup. All benchmarks are conducted on a machine with an AMD EPYC Rome 7702P with 64 cores and 1024 GB of main memory running Ubuntu 20.04 with kernel version 5.4.0 and GCC 9.4. The DEM data is stored on an Intel P4510 2 TB NVMe SSD. We use the 3″ data set from viewfinderpanoramas [dFer11] with worldwide coverage. To build smaller test instances from the worldwide data set, we choose a random starting tile and add neighboring tiles in a spiraling manner around it until the desired number of tiles is reached. For each instance size, we generate several instances to cover a wide range of

[8] The algorithm by Kirmse and de Ferranti always chooses the north-west corner of a peak.

Table 4.1 Test instances used for worldwide and NA-EU runtime time testing.

(a) Worldwide data set.

Tile count	Random samples
4	50
8	40
16	32
32	24
64	18
128	12
256	8
512	4
1024	4
2048	4
4096	3
8192	3
16 384	2
26 095	2

(b) NA-EU data set.

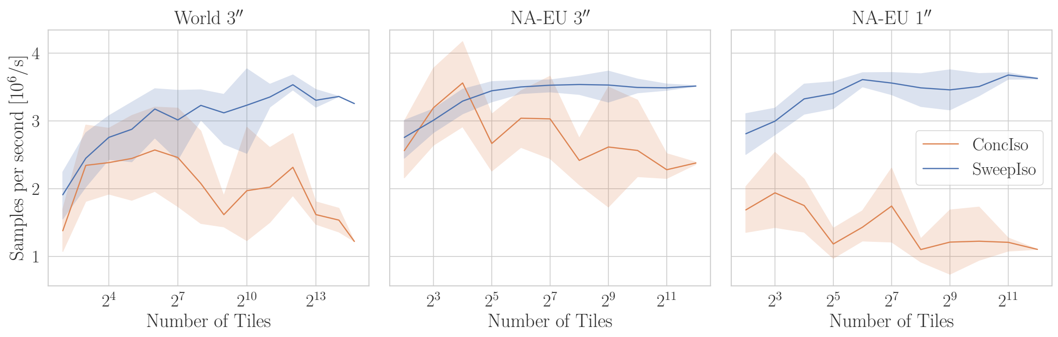
Tile count	Random samples
4	50
8	40
16	32
32	24
64	18
128	12
256	8
512	4
1024	4
2048	4
4096	3

Table 4.2 DEM models of Earth, Mars and the Moon used in our experiments. The reported runtime is the single-threaded runtime for the entire data set.

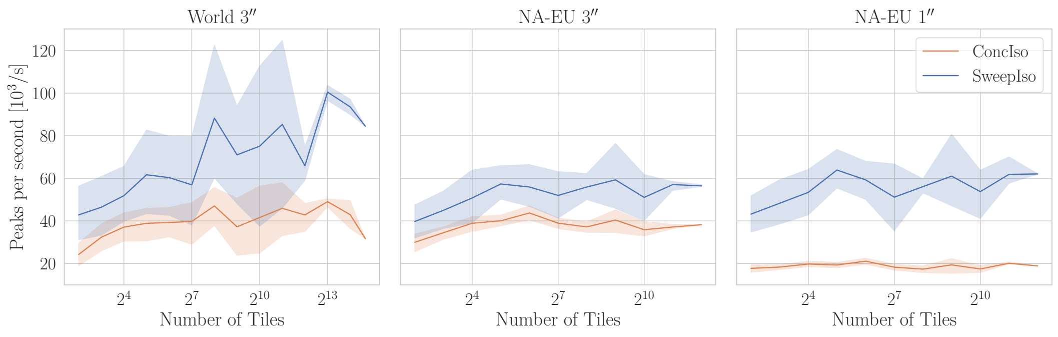
Name	Resolution	Points $[1 \times 10^9]$	Size [GB]	Coverage	Runtime [h]
Earth SRTM	$3''$ (90 m)	20	71	Global	2.07
Earth SRTM NA-EU	$1''$ (30 m)	49	105	Cont. US+CAN+EU	3.8
Moon SLDEM2015	$7''$ (59 m)	11	22	$[60° N, 60° S]$	2.5
Mars MGS MOLA - MEX HRSC Blended	$12''$ (197.6 m)	5.7	11	Global	1.3

terrain, refer to Table 4.1 for details. Additionally, we use the $3''$ and $1''$ DEM for the USA, Canada and most of Europe from viewfinderpanoramas [dFer11] to study the scaling behavior for higher-resolution DEMs. This data set corresponds to 4294 tiles or roughly 16 % of the total test data set. We will call this data set NA-EU. We also use data sets from other celestial bodies: Mars [Gwi+10] and the Moon [Bar+16]. Table 4.2 lists some properties of the studied data sets along with the runtime of our algorithm to determine the isolation of every peak contained in them.

For all benchmarks we use an isolation threshold $I_{\min}$ of 1 km and report the mean runtime of 5 runs. I/O costs are not part of the reported figures as we use the MOUNTAINS framework [KdF17] for them without an attempt at optimization. They are about the same time as computation for $3''$ DEMs and about 10 % of computation time for $1''$ ones and thus could be overlapped with the computation in an optimized framework.



(a) Throughput in sample points per second over number of tiles.



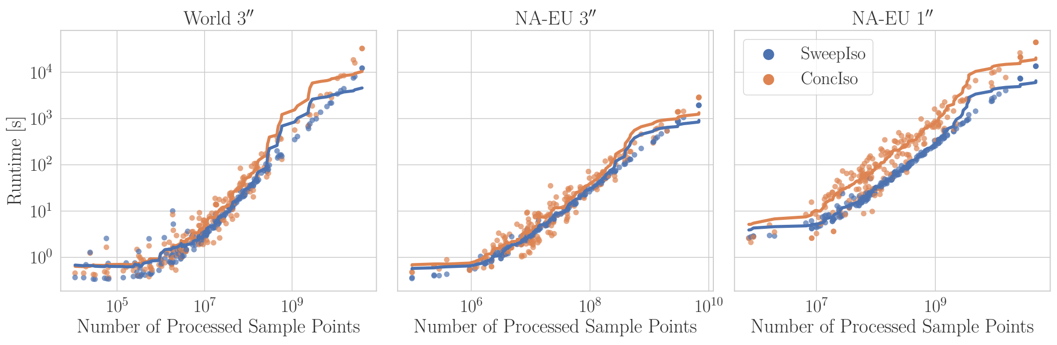
(b) Throughput in processed peaks per second over number of tiles.

Figure 4.7 Single-threaded runtime comparison of SweepIso and ConcIso, regarding throughput of sample points and processed peaks over number of tiles.

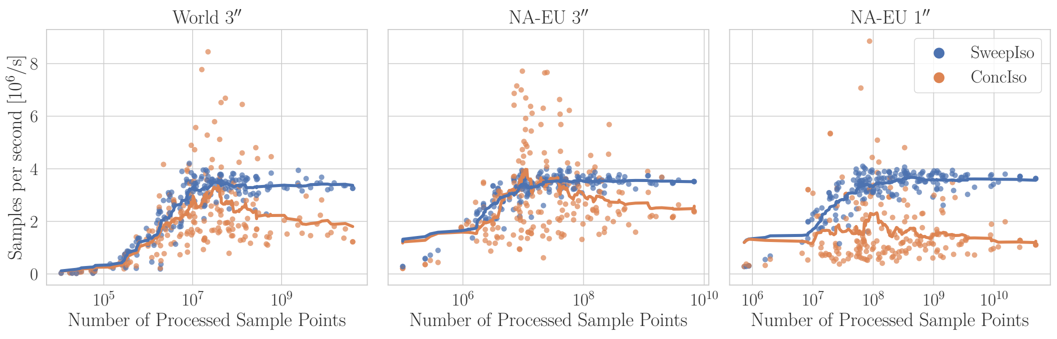
4.7.1 Runtime and Scaling Behavior

The runtime of the algorithms depends on the number of sample points in the DEM. These can either increase due to a larger search area or a higher-resolution DEM. Another factor is the number of processed peaks, since every peak starts a local search in ConcIso and a nearest neighbor query in SweepIso. A larger search area increases the number of tiles and the number of peaks, whereas higher-resolution data mostly increases the number of points per tile. High-resolution DEMs can contain more peaks than lower-resolution ones due to the more truthful representation of the terrain, however these are predominantly low-isolation peaks and are filtered out in the first pass. We study both effects in our experiments by using different resolution DEMs as well as increasing search areas.

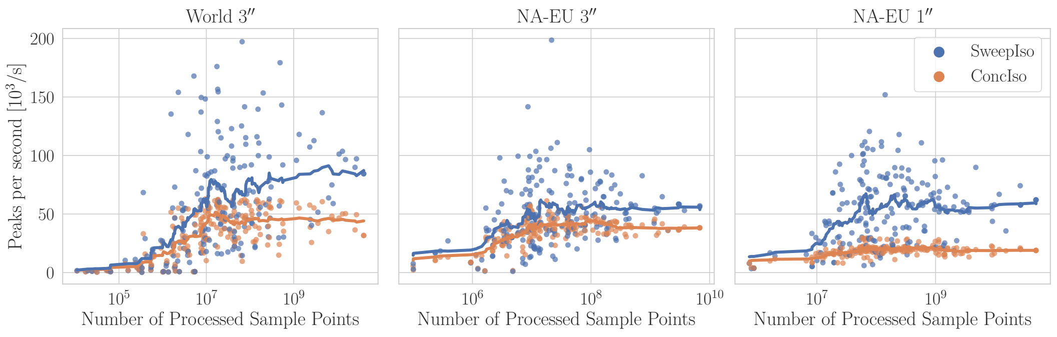
SweepIso exhibits a nearly constant throughput of sample points per second with increasing search area, while ConcIso's throughput degrades—refer to Figure 4.7a. Figure 4.10a shows that SweepIso outperforms ConcIso by a factor of 2 to 3 in terms of sample point throughput. For instance, we reduce the time required to compute the isolation of every peak on Earth from 9 h down to 3.5 h. However, ConcIso's runtime scales significantly better than its $\mathcal{O}(n^2)$ worst-case runtime bound would suggest. This is because most peaks have a relatively low isolation. In fact 99.996 % of discovered peaks on the world data set have an isolation below 50 km and more than 99 % below 10 km. Since one tile covers on average an area of 70 km × 111 km, the nearest higher point is most of the time within the same tile as the peak, where fast approximations for the distance calculation are used. Nevertheless, for high-resolution DEMs the computation cost per peak is significantly higher for ConcIso than for SweepIso, refer to Figure 4.7b.



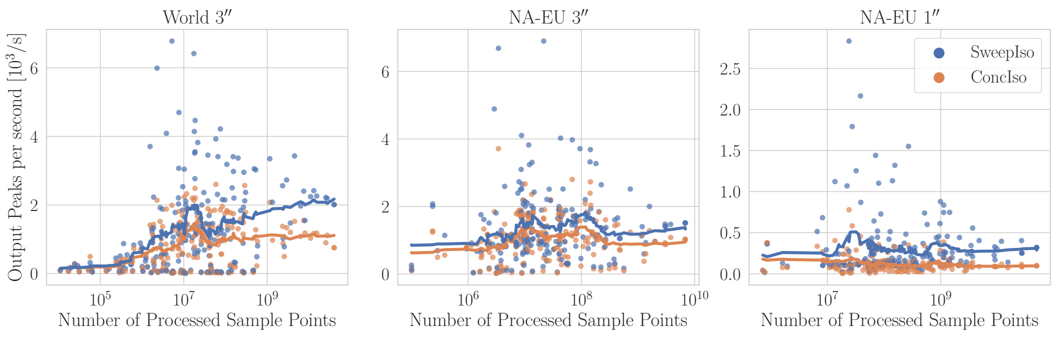
(a) Execution time over processed sample points.



(b) Throughput in points per second over processed sample points.

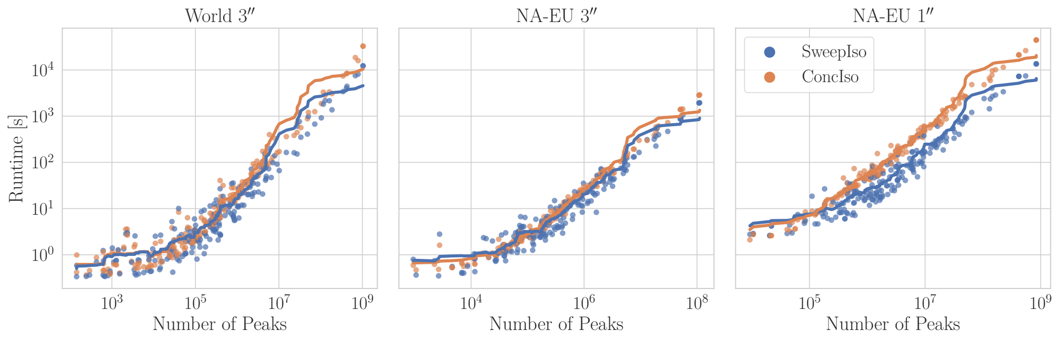


(c) Throughput in peaks per second over processed sample points.

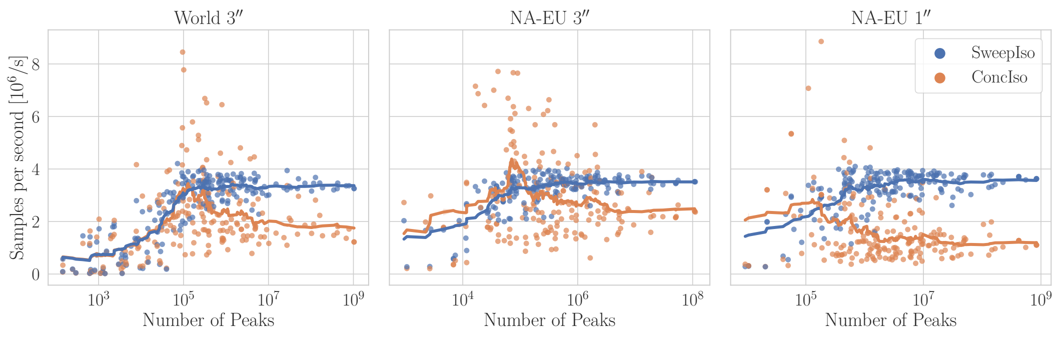


(d) Throughput in output peaks per second over processed sample points.

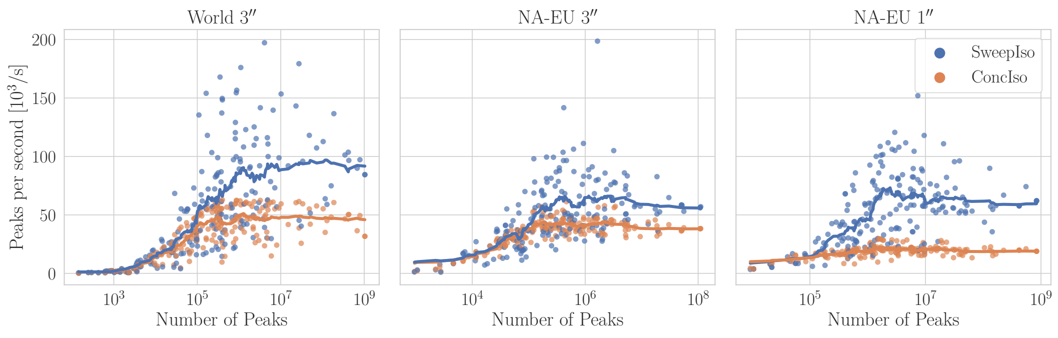
Figure 4.8 Single-threaded runtime comparison of SweepIso and ConcIso.



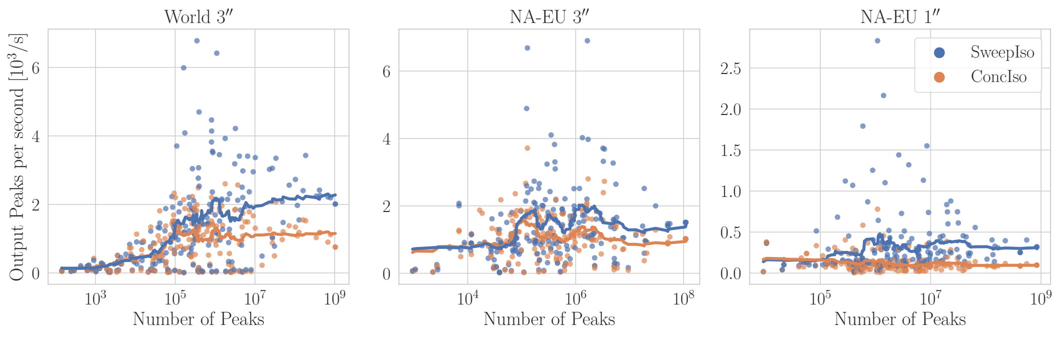
(e) Execution time over processed peaks.



(f) Throughput in points per second over processed peaks.

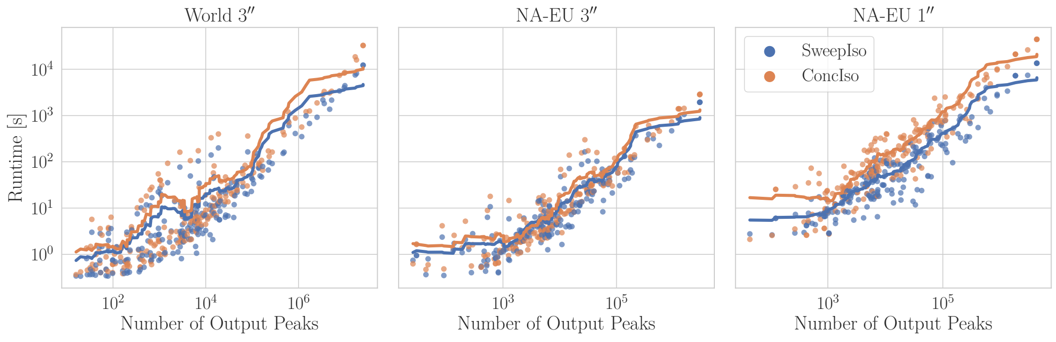


(g) Throughput in peaks per second over processed peaks.

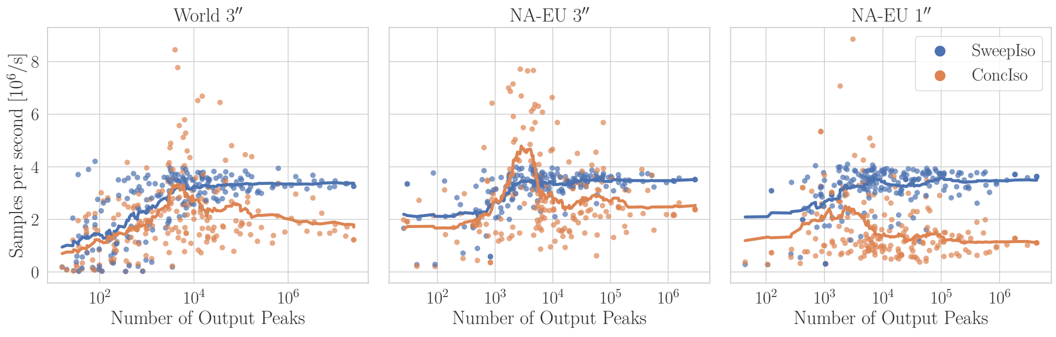


(h) Throughput in output peaks per second over processed peaks.

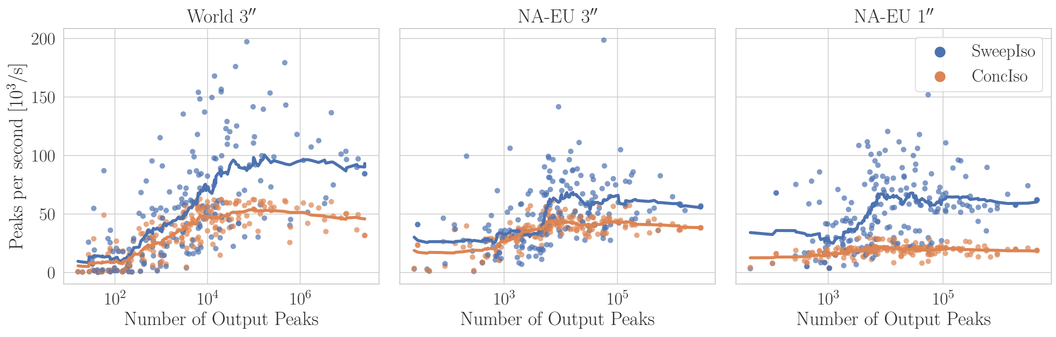
Figure 4.8 (cont.) Single-threaded runtime comparison of SweepIso and ConcIso.



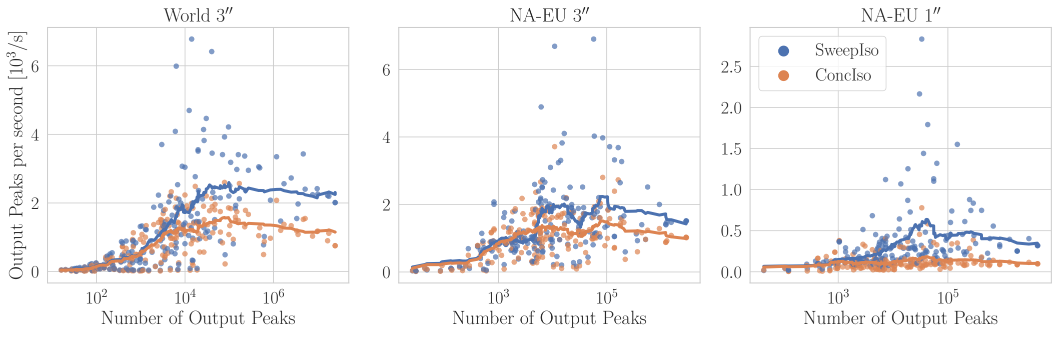
(i) Execution time over output peaks.



(j) Throughput in points per second over output peaks.

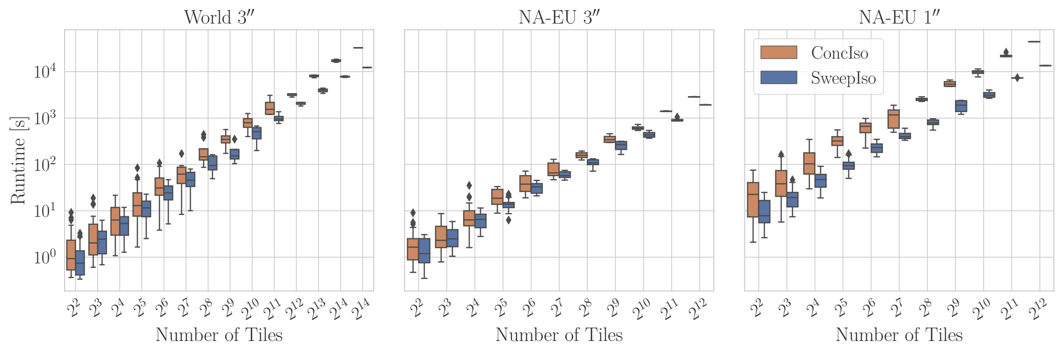


(k) Throughput in peaks per second over output peaks.

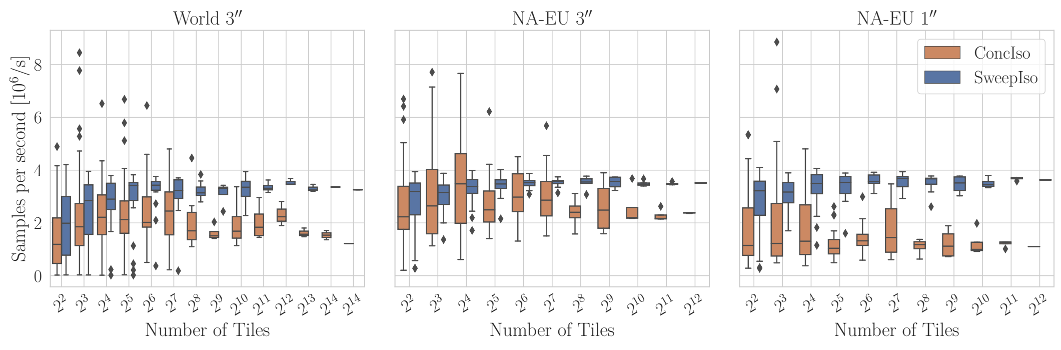


(l) Throughput in output peaks per second over output peaks.

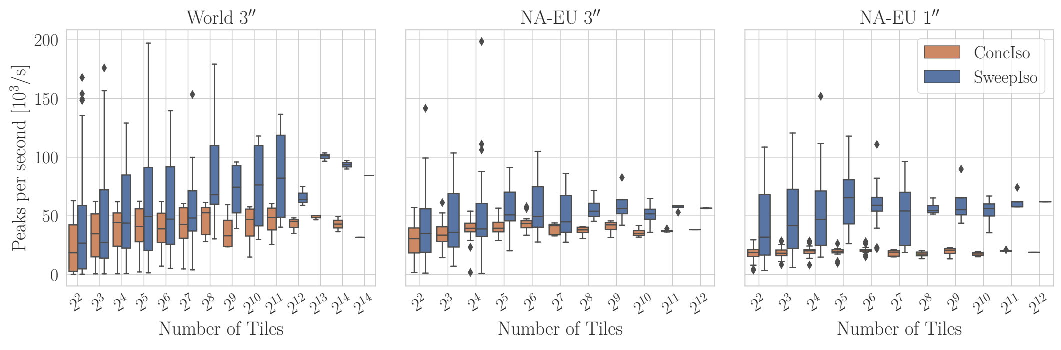
Figure 4.8 (cont.) Single-threaded runtime comparison of SweepIso and ConcIso.



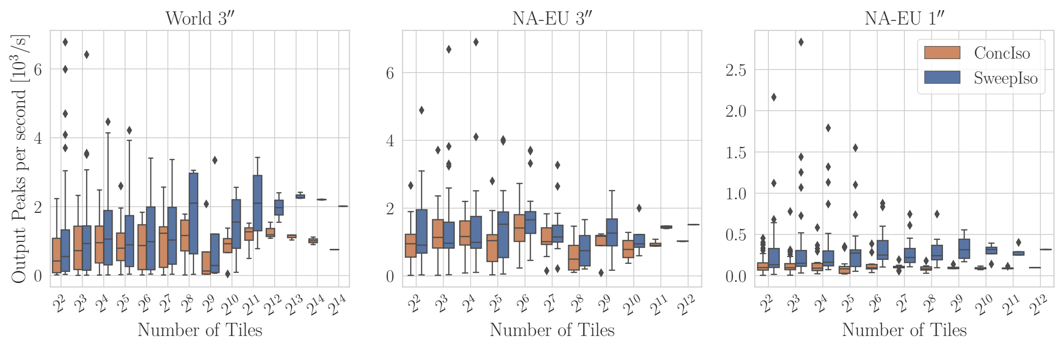
(m) Execution time over number of tiles.



(n) Throughput in points per second over number of tiles.



(o) Throughput in peaks per second over number of tiles.



(p) Throughput in output peaks per second over number of tiles.

Figure 4.8 (cont.) Single-threaded runtime comparison of SweepIso and ConcIso.

To analyze the runtime behavior of SweepIso and ConcIso more in-depth, we can characterize the input according to the number of

- tiles,
- processed sample points,
- detected peaks, or
- peaks above the isolation threshold $I_{\min}$.

A fixed number of tiles can cover a wide range of terrain forms, influencing both the number of peaks in the search area and the number of sample points. The latter is due to large bodies of water being voided in the DEM, resulting in fewer sample points in a tile. The performance of the algorithms can be measured by

- elapsed time,
- throughput of detected peaks, or
- throughput of peaks above $I_{\min}$.

Figure 4.8 presents a detailed comparison of both algorithms regarding these input characteristics and performance metrics on a per instance level. In the following we discuss some key insights into the runtime behavior of SweepIso and ConcIso. For SweepIso, the throughput of peaks is nearly independent of the resolution of the DEM (Figure 4.8c), however the throughput of output peaks—those with isolation above $I_{\min}$—significantly decreases for higher-resolution DEMs (Figure 4.8d). This is due to the higher number of low-isolation peaks in high-resolution DEMs that need to be processed but are ultimately filtered out. Comparing the throughput of sample points and peaks reveals the different algorithmic approaches of SweepIso and ConcIso, refer to Figures 4.8b and 4.8c. SweepIso processes sample points at a near constant rate, but shows a high variance in the throughput of peaks. This is because it performs an approximately constant work per sample point to reduce the work required per peak. The throughput of peaks therefore depends on the number of non-peak sample points that need to be processed per peak, which is highly dependent on the terrain of the search area. ConcIso on the other hand processes peaks at a nearly constant rate with little variance, because no work per non-peak sample point is performed. As most peaks have a low isolation, the number of sample points that need to be processed per peak is low and relatively constant. Nevertheless, our approach is faster than ConcIso for all tested resolutions and search areas. Figures 4.8m–4.8p further show the dependence of SweepIso and ConcIso on the processed terrain forms. Both algorithms are highly influenced by terrain covered by the processed tiles, as seen in the total execution time (Figure 4.8m). Considering the variance of the throughput of sample points and peaks in Figures 4.8n and 4.8o, confirms the observations made above.

Figure 4.9 shows the runtime composition of the bounding and the finalization pass. As the DEM resolution is reduced in the bounding pass, the finalization pass requires the majority of the runtime, especially for higher-resolution inputs. Geometric computations only make a small fraction of the runtime. The high-point pass requires just 0.001 % of the total execution time and is therefore omitted in the figure.

Figure 4.10a shows the single-threaded speedup of SweepIso over ConcIso. For 1″ DEMs, SweepIso is over 3 times faster than ConcIso, with a speedup of factor 2 for the 3″ DEMs world data set. A more detailed speedup comparison is shown in Figure 4.11. Figure 4.10b shows the results of our multithreaded runtime experiments with the NA-EU data set. SweepIso scales well with physical cores, but does not benefit from hyper-threading (HT). For 64 cores it reaches a speedup of 25. We verified the scaling behavior of

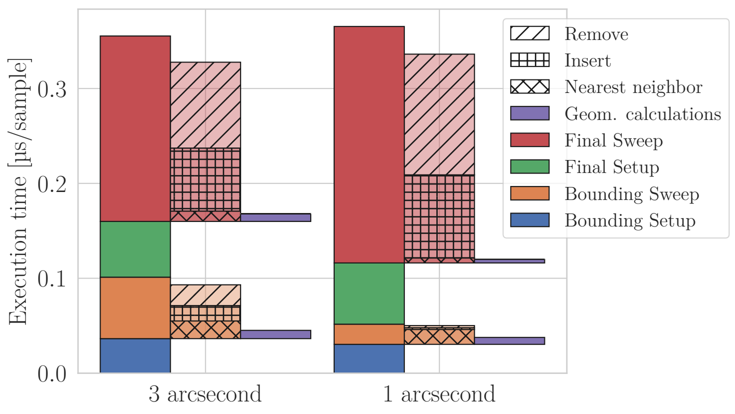
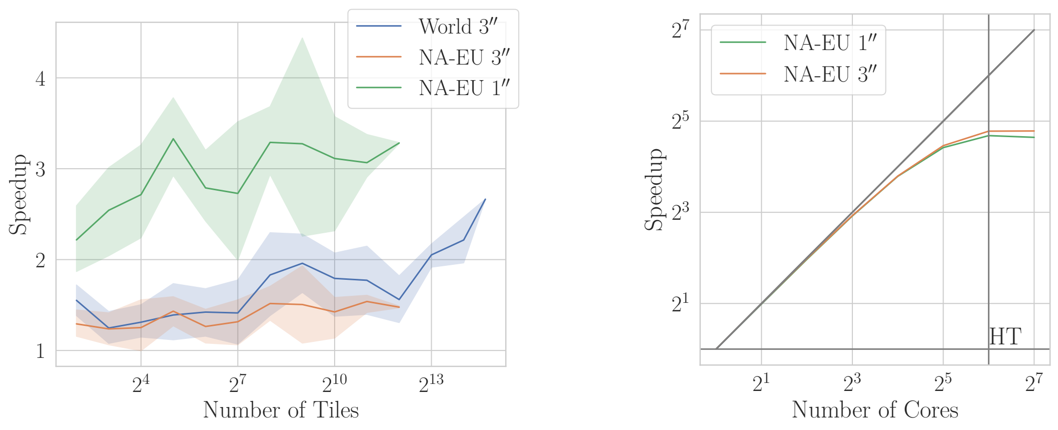


Figure 4.9 Runtime composition of the bounding and finalization pass for 3″ and 1″ data. The sweep-plane operations of these passes are further subdivided into remove, insert, nearest neighbor search operations as well as the time for distance calculations.



(a) Single-threaded speedup.

(b) Multi-threaded relative speedup.

Figure 4.10 Single-threaded speedup of SweepIso over ConcIso and relative speedup of SweepIso for multi-threaded execution.

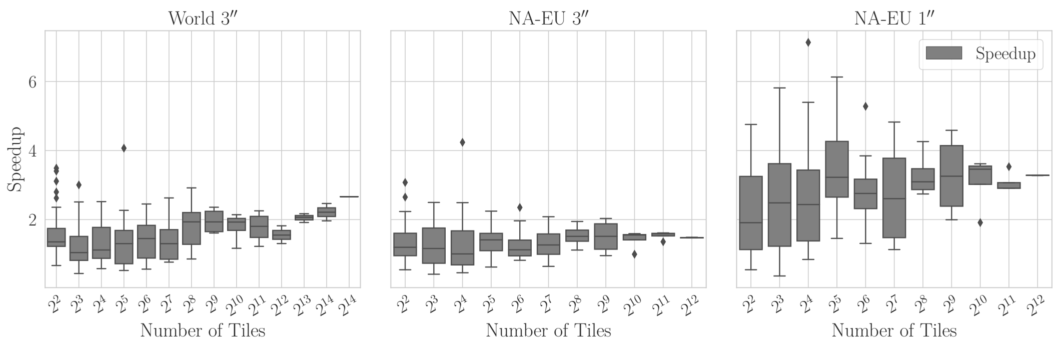


Figure 4.11 Single-threaded speedup of SweepIso over ConcIso.

the algorithm for the entire Earth and were able to confirm a speedup of 25 on 64 cores. This corresponds to a runtime of 8 min to calculate the isolation of every peak on Earth. We were not able to execute ConcIso with multiple threads due to issues in the implementation.

4.7.2 Solution Quality

The isolation and witnessing ILP of a peak may change over a geologic timescale due to tectonic activity, erosion, and deposition, but, in principle, are well-defined at any particular point in time. However, the algorithmically computed values depend on imprecisions in both the data and the used algorithms.

Due to the design of SweepIso, more expensive distance approximations can be used to find the ILP than in ConcIso. This results in closer ILPs being found. Figure 4.12 displays the distribution of the difference in isolation and the distance between the found ILPs between SweepIso and ConcIso, using the same data. Even if the isolation values between the two algorithms do not vary greatly, the distances between the ILPs do. For example for the Cerro Gordo summit in Mexico both algorithms find ILPs that are more than 600 km apart while SweepIso's ILP is merely 0.7 km closer to the peak.

To further evaluate the results, we used the collection of peaks with more than 300 km isolation from Peakbagger [Sla23]. The comparison showed that for about 75 % of peaks in this list the isolation deviation is below 2 km. In a DEM, a sample point corresponds to the average elevation of the area it represents. This often leads to an underestimation of a peak's elevation, sometimes significantly [KdF17]. For instance, according to our DEM data, Galdhøpiggen (the highest point of Scandinavia) is 11 m lower than Glittertind. This causes a significant change in isolation of both mountains. Table 4.3 lists the five summits with the biggest differences in isolation between peakbagger and our calculation. The five most isolated peaks on Mars and the Moon as determined by our algorithm are presented in Table 4.4. To the best of our knowledge, this is the first such list.

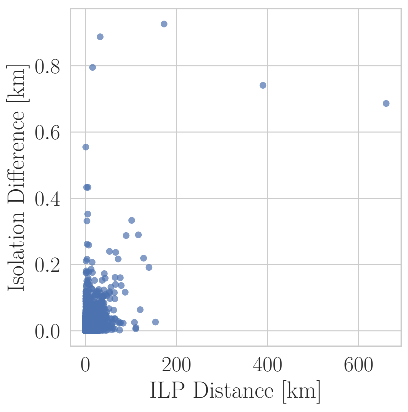


Figure 4.12 Comparison of difference in isolation between SweepIso and ConcIso and distance between found ILPs.

Table 4.3 The biggest differences between our calculated isolation and Peakbagger (PB) data. Ground truth elevation data is due to [COM17; Pea23].

Mountain	PB Rank	PB Isolation [km]	SweepIso Isolation [km]	Notes
Galdhøpiggen (Norway)	47	1568.3	12.9	*Galdhøpiggen* DEM elev.: 2455 m – real elev.: 2469 m *Glittertind* DEM elev.: 2466 m – real elev.: 2457 m Isolation 1563.5 km
Mt. Kirkpatrick (Antarctica)	45	1585.2	53.5	found ILP at -83.8967;168.3733 DEM elev.: 4416 m – real elev.: 4528 m ILP elev.: 4416 m
Mt. Hope (Antarctica)	75	1113.5	203.4	No data found in other sources
Dome Charlie (Antarctica)	92	971.8	248.6	found ILP at -76.3783;116.3708 DEM elev.: 3265 m – real elev.: 3233 m ILP elev. 3266 m
Mauga Silisili (Samoa)	23	2245.1	2502.8	DEM elev.: 1854 m – real elev.:1863 m PB ILP elev.: 1879 m found ILP elev.: 1836 m
Cocos Islands High Point	94	961.4	947.4	Found ILP at Enggano Island

Table 4.4 Most isolated peaks on Mars and the Moon according to the DEMs listed in Table 4.2.

Name	Coordinates		Elevation [m]	ILP		Isolation [km]
Mars						
Olympus Mons	17.33°N	133.42°W	21 226	-		∞
Cruls crater wall	42.27°S	163.78°E	4331	46.79°S	147.63°W	2037.43
Near Cyclopia	6.34°S	129.19°E	3806	18.59°N	149.99°E	1913.59
Huygens crater wall	10.06°S	54.62°E	4604	10.50°S	85.09°E	1776.76
Ascraeus Mons	11.76°N	104.53°W	18 321	17.89°N	131.64°W	1593.92
The Moon						
Engel'gardt crater wall	5.41°N	158.63°W	10 783.3	-		∞
Between Tacitus and Fermat craters	18.93°S	19.17°E	4832.33	60.0°S	72.65°W	2261.5
Near Lewis crater	21.04°S	112.33°W	9459.83	5.26°N	157.98°W	1574.56
Near Calippus crater	39.09°N	9.46°E	3625.68	9.63°S	2.06°E	1492.13
Dellinger crater wall	7.25°S	142.0°E	7561.73	12.91°S	170.22°W	1434.95

4.8 Conclusions

We have presented SweepIso, a scalable and efficient algorithm to compute the isolation of peaks. SweepIso considerably outperforms the previous, more brute-force, state-of-the-art approach. The performance gains also enable more accurate distance calculations at decisive places resulting in higher accuracy. SweepIso is able to process the entire Earth for currently publicly available data within minutes. This is relevant as it indicates that SweepIso can also handle higher-resolution data that is available commercially or that will be available in the future. SweepIso's two-level semi-external sweeping architecture may also be an interesting design pattern for other computations on massive DEM data. Furthermore, SweepIso could serve as a benchmark for dynamic nearest neighbor search data structures.

Future Work. From an application perspective, it would be interesting to compute not only isolations for a given, necessarily imprecise data set but to compute confidence bounds that take into account error margins in the input data. This would be possible at a moderate increase in cost. Peaks could be replaced by enclosing boxes/circles while vertical errors could be handled by having "may-be-there" and "must-be-there" insertion events and sweep-plane data structures. Geographically most interesting would be those isolations that change a lot depending on how high exactly particular pairs of peaks are. Those pairs could then be valuable targets for additional data cleaning or new measurements.

For algorithm engineering, it would be interesting to close the gap between theory and practice with respect of nearest neighbor data structures. We have reasonable empirical performance of simple data structures like k-D trees but no well-fitting performance guarantees applicable to SweepIso. For example, one could look for a more general characterization of inputs where cover trees [BKL06] work provably well.

Appendix

Acronyms

ADT	abstract data type. 5
BW algorithm	Bowyer-Watson incremental insertion algorithm. 20
CGAL	Computational Geometry Algorithms Library. 14
CUDEM	continuously updated digital elevation model. 90
D&C	divide-and-conquer. v, 2, 11, 12
DEM	digital elevation model. vi, 1, 3, 89
DT	Delaunay triangulation. 2, 11
EMST	Euclidean minimum spanning tree. 2, 3, 13
GMP	GNU Multiple Precision Arithmetic Library. 4
GNG	geographic neighborhood graph. 72
GPGPU	general-purpose graphics processing unit. 16
ILP	isolation limit point. 3, 89
LEDA	Library of Efficient Data types and Algorithms. 4
MPFR	GNU Multiple Precision Floating-Point Reliably Library. 4
NCA	nearest partition center assignment. 37
NNG	nearest neighbor graph. 3, 13
NSA	nearest sample point assignment. 37
OVD	oriented Voronoi diagram. 72
PE	processing element. 22
PQ	priority queue. 6
RDG	random Delaunay graph. 61
RGG	random geometric graph. 61
RHG	random hyperbolic graph. 61
RNG	relative neighborhood graph. 13
SRTM	Shuttle Radar Topography Mission. 91
TIG	topographic isolation graph. 3
w.h.p.	with high probability. 62

List of Algorithms

List of Figures

List of Tables

Publications and Supervised Theses

In Conference Proceedings

- D. Funke, N. Hüning, and P. Sanders. "A Sweep-Plane Algorithm for Calculating the Isolation of Mountains". en. In: *31st Annual European Symposium on Algorithms (ESA 2023)*. Edited by I. L. Gørtz, M. Farach-Colton, S. J. Puglisi, and G. Herman. Volume 274. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 51:1–51:17. DOI: `10.4230/LIPIcs.ESA.2023.51`
- D. Funke and P. Sanders. "Efficient Yao Graph Construction". In: *21st International Symposium on Experimental Algorithms (SEA 2023)*. Edited by L. Georgiadis. Volume 265. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 20:1–20:20. ISBN: 978-3-95977-279-2. DOI: `10.4230/LIPIcs.SEA.2023.20`
- D. Funke, P. Sanders, and V. Winkler. "Load-Balancing for Parallel Delaunay Triangulations". In: *Proceedings of Euro-Par 2019*. Edited by R. Yahyapour. Göttingen, Germany: Springer-Verlag, 2019, pages 156–169. DOI: `10.1007/978-3-030-29400-7_12`
- D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. "Communication-Free Massively Distributed Graph Generation". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2018, pages 336–347. DOI: `10.1109/ipdps.2018.00043`
- D. Funke and P. Sanders. "Parallel *d*-D Delaunay Triangulations in Shared and Distributed Memory". In: *2017 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics (SIAM), 2017, pages 207–217. DOI: `10.1137/1.9781611974768.17`
- I. Shapoval, M. Clemencic, B. Hegner, D. Funke, D. Piparo, and P. Mato. "Graph-based decision making for task scheduling in concurrent Gaudi". In: *2015 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*. IEEE, Nov. 2015. DOI: `10.1109/nssmic.2015.7581843`
- M. Clemencic, D. Funke, B. Hegner, P. Mato, D. Piparo, and I. Shapoval. "Gaudi components for concurrency: Concurrency for existing and future experiments". In: *16th International workshop on Advanced Computing and Analysis Techniques in physics research (ACAT2014)*. Volume 608. Journal of Physics: Conference Series. IOP Publishing, May 2015, page 012021. DOI: `10.1088/1742-6596/608/1/012021`
- D. Funke, T. Hauth, V. Innocente, G. Quast, P. Sanders, and D. Schieferdecker. "Parallel track reconstruction in CMS using the cellular automaton approach". In: *20th International Conference on Computing in High Energy and Nuclear Physics (CHEP2013)*. Volume 513. Journal of Physics: Conference Series 5. IOP Publishing, June 2014, page 052010. DOI: `10.1088/1742-6596/513/5/052010`

- D. Funke, F. Brosig, and M. Faber. "Towards Truthful Resource Reservation in Cloud Computing". In: *Proceedings of the 6th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2012)*. IEEE, Dec. 2012. DOI: 10.4108/valuetools.2012.250173
- O. Tveretina and D. Funke. "Deciding Reachability for 3-Dimensional Multi-Linear Systems". In: *Proceedings of 2nd Int Symposium on Games, Automata, Logics and Formal Verification (GandALF 2011)*. Volume 54. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, June 2011, pages 250–262. DOI: 10.4204/eptcs.54.18
- D. Funke and F. Kerschbaum. "Privacy-Preserving Multi-Objective Evolutionary Algorithms". In: *Parallel Problem Solving from Nature, PPSN XI. PPSN 2010*. Edited by R. Schaefer, C. Cotta, J. Kołodziej, and G. Rudolph. Volume 6239. LNCS. Berlin Heidelberg: Springer, Sept. 2010, pages 41–50. DOI: 10.1007/978-3-642-15871-1_5

Journal Articles

- D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. von Looz. "Communication-Free Massively Distributed Graph Generation". In: *Journal of Parallel Distributed Computing* 131.C (Sept. 2019), pages 200–217. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2019.03.011

Book Chapters

- F. Kerschbaum and D. Funke. "Protecting Privacy by Secure Computation". In: *Privacy Protection Measures and Technologies in Business Organizations*. Edited by G. O. Yee. IGI Global, Dec. 2012. Chapter 11, pages 268–284. DOI: 10.4018/978-1-61350-501-4.ch011

Technical Reports

- D. Funke, N. Hüning, and P. Sanders. *A Sweep-plane Algorithm for Calculating the Isolation of Mountains*. Technical report. Karlsruhe Institute of Technology, May 2023. DOI: 10.48550/ARXIV.2305.08470. arXiv: 2305.08470 [cs.DS]
- D. Funke and P. Sanders. *Efficient Yao Graph Construction*. Technical report. Karlsruhe Institute of Technology, Mar. 14, 2023. DOI: 10.48550/arXiv.2303.07858. arXiv: 2303.07858v1 [cs.CG]
- D. Funke, P. Sanders, and V. Winkler. *Load-Balancing for Parallel Delaunay Triangulations*. Technical report. Karlsruhe Institute of Technology, Feb. 20, 2019. DOI: 10.48550/arXiv.1902.07554. arXiv: 1902.07554v1 [cs.DS]
- D. Funke, S. Lamm, U. Meyer, P. Sanders, M. Penschuck, C. Schulz, D. Strash, and M. von Looz. *Communication-free Massively Distributed Graph Generation*. Technical report. Karlsruhe Institute of Technology, Oct. 20, 2017. DOI: 10.48550/arXiv.1710.07565. arXiv: 1710.07565v3 [cs.DC]

Theses

- D. Funke. "Parallel Triplet Finding for Particle Track Reconstruction". Master's thesis. Karlsruhe Institute of Technology, June 2013. DOI: 10.5445/IR/1000048585
- D. Funke. "Secure Evolutionary Algorithms for Cooperative Supply Chain Planning". Bachelor's thesis. Baden-Württemberg Cooperative State University (DHBW), Jan. 2009

Supervised Theses

- N. Hüning. “A Sweepline Algorithm for Calculating the Isolation of Mountains”. Bachelor’s thesis. Karlsruhe Institute of Technology, Oct. 2022. DOI: 10.5445/IR/1000165010
- J. Ebbing. “Graph-based Tracking for the Belle II CDC”. Master’s thesis. Karlsruhe Institute of Technology, May 2019
- V. Winkler. “Load Balancing für Parallele Delaunay-Triangulierung”. Bachelor’s thesis. Karlsruhe Institute of Technology, Mar. 2018. DOI: 10.5445/IR/1000092121
- F. Schroedter. “Graph-theoretische Charakterisierung von Metropolregionen”. Master’s thesis. Karlsruhe Institute of Technology, Sept. 2017
- P. Zander. “A Practical, Compact Representation for Plane Triangulations”. Bachelor’s thesis. Karlsruhe Institute of Technology, Sept. 2017
- D. Merkel. “Graph-based Approach to Particle Track Reconstruction”. Master’s thesis. Karlsruhe Institute of Technology, Nov. 2015

Bibliography

[AB98] M. Akra and L. Bazzi. "On the Solution of Linear Recurrence Equations". In: *Computational Optimization and Applications* 10.2 (May 1998), pages 195–210. DOI: 10.1023/a:1018373005182 (cited on page 23).

[ACG88] A. Aggarwal, B. Chazelle, and L. Guibas. "Parallel computational geometry". In: *Algorithmica* 3.1 (1988), pages 293–327. DOI: 10.1007/bf01762120 (cited on pages 9, 14, 15, 17).

[Aga+08] P. K. Agarwal, L. Arge, T. Mølhave, and B. Sadri. "I/O-efficient efficient algorithms for computing contours on a terrain". In: *Proceedings of the twenty-fourth annual symposium on Computational geometry.* ACM, June 2008. DOI: 10.1145/1377676.1377698 (cited on pages 89, 96).

[AGP90] E. G. Anagnostou, L. J. Guibas, and V. G. Polimenis. "Topological sweeping in three dimensions". In: *Algorithms.* Edited by T. Asano, T. Ibaraki, H. Imai, and T. Nishizeki. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pages 310–317. ISBN: 978-3-540-47177-6. DOI: 10.1007/3-540-52921-7_80 (cited on pages 10, 92).

[Akh+17] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. "Engineering a direct k-way Hypergraph Partitioning Algorithm". In: *Workshop on Algorithm Engineering and Experiments, (ALENEX).* SIAM, 2017, pages 28–42. DOI: 10.1137/1.9781611974768.3 (cited on page 38).

[Akm+89] V. Akman, W. Franklin, M. Kankanhalli, and C. Narayanaswami. "Geometric computing and uniform grid technique". In: *Computer-Aided Design* 21.7 (1989), pages 410–420. ISSN: 0010-4485. DOI: 10.1016/0010-4485(89)90125-5 (cited on page 72).

[AL62] G. M. Adelson-Velsky and E. M. Landis. "An algorithm for organization of information". In: *Doklady Akademii Nauk.* Volume 146. Russian Academy of Sciences. 1962, pages 263–266 (cited on pages 7, 80).

[Ala08] Alaska Satellite Facility. *Radarset Antarctic Mapping.* Online; accessed 05-12-2023. 2008. URL: https://asf.alaska.edu/data-sets/derived-data-sets/radarsat-antarctic-mapping-project-ramp/ (cited on page 91).

[Ama+23] C. J. Amante, M. Love, K. Carignan, M. G. Sutherland, M. MacFerrin, and E. Lim. "Continuously Updated Digital Elevation Models (CUDEMs) to Support Coastal Inundation Modeling". In: *Remote Sensing* 15.6 (2023). ISSN: 2072-4292. DOI: 10.3390/rs15061702 (cited on page 90).

[Ary+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. "An optimal algorithm for approximate nearest neighbor searching fixed dimensions". In: *Journal of the ACM (JACM)* 45.6 (1998), pages 891–923. DOI: 10.1145/293347.293348 (cited on page 94).

[ASS18] Y. Akhremtsev, P. Sanders, and C. Schulz. "High-Quality Shared-Memory Graph Partitioning". In: *Euro-Par 2018: Parallel Processing.* Edited by M. Aldinucci, L. Padovani, and M. Torquati. Springer, 2018, pages 659–671. DOI: 10.1109/tpds.2020.3001645 (cited on pages 38, 40, 45).

[Axt+17] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. “In-Place Parallel Super Scalar Samplesort (IPSSSSo)”. In: *25th Annual European Symposium on Algorithms*. Volume 87. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 9:1–9:14. DOI: `10.4230/LIPIcs.ESA.2017.9` (cited on page 99).

[Bar+15] L. Barba, P. Bose, M. Damian, R. Fagerberg, W. L. Keng, J. O’Rourke, A. Van Renssen, P. Taslakian, S. Verdonschot, and G. Xia. “New and improved spanning ratios for Yao graphs”. In: *Journal of Computational Geometry* 6.2 (2015), pages 19–53. DOI: `10.20382/JOCG.V6I2A3` (cited on pages 2, 69).

[Bar+16] M. Barker, E. Mazarico, G. Neumann, M. Zuber, J. Haruyama, and D. Smith. “A new lunar digital elevation model from the Lunar Orbiter Laser Altimeter and SELENE Terrain Camera”. In: *Icarus* 273 (2016), pages 346–355. ISSN: 0019-1035. DOI: `10.1016/j.icarus.2015.07.039` (cited on pages 90, 100).

[Bat+10] V. H. Batista, D. L. Millman, S. Pion, and J. Singler. “Parallel geometric algorithms for multi-core computers”. In: *Computational Geometry* 43.8 (2010), pages 663–677. DOI: `10.1016/j.comgeo.2010.04.008` (cited on pages 12, 14, 29, 31, 40, 67).

[BC08] P. Bhattacharyya and B. K. Chakrabarti. “The mean distance to the nth neighbour in a uniform distribution of random points”. In: *European Journal of Physics* 29.3 (2008), pages 639–645. DOI: `10.1088/0143-0807/29/3/023` (cited on page 63).

[Ben75] J. L. Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (Sept. 1975), pages 509–517. DOI: `10.1145/361002.361007` (cited on pages 7, 17, 94, 96).

[Ben79] J. L. Bentley. “Decomposable searching problems”. In: *Information Processing Letters* 8.5 (June 1979), pages 244–251. DOI: `10.1016/0020-0190(79)90117-0` (cited on page 7).

[Ben80] J. L. Bentley. “Multidimensional divide-and-conquer”. In: *Communications of the ACM* 23.4 (Apr. 1980), pages 214–229. DOI: `10.1145/358841.358850` (cited on page 9).

[BHS80] J. L. Bentley, D. Haken, and J. B. Saxe. “A general method for solving divide-and-conquer recurrences”. In: *ACM SIGACT News* 12.3 (Sept. 1980), pages 36–44. DOI: `10.1145/1008861.1008865` (cited on page 23).

[BKL06] A. Beygelzimer, S. Kakade, and J. Langford. “Cover Trees for Nearest Neighbor”. In: *Proceedings of the 23rd International Conference on Machine Learning*. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2006, pages 97–104. DOI: `10.1145/1143844.1143857` (cited on page 110).

[Ble+99] E. G. Blelloch, L. G. Miller, C. J. Hardwick, and D. Talmor. “Design and Implementation of a Practical Parallel Delaunay Algorithm”. In: *Algorithmica* 24.3 (1999), pages 243–269. DOI: `10.1007/pl00008262` (cited on pages 12, 15).

[BM70] R. Bayer and E. McCreight. “Organization and maintenance of large ordered indices”. In: *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET ’70*. ACM Press, 1970, pages 107–141. DOI: `10.1145/1734663.1734671` (cited on page 7).

[BO79] J. L. Bentley and T. A. Ottmann. “Algorithms for reporting and counting geometric intersections”. In: *IEEE Transactions on Computers* (1979), pages 643–647. DOI: `10.1109/tc.1979.1675432` (cited on pages 10, 71, 92).

[Bon+10] N. Bonichon, C. Gavoille, N. Hanusse, and D. Ilcinkas. “Connections between Theta-Graphs, Delaunay Triangulations, and Orthogonal Surfaces”. In: *Graph Theoretic Concepts in Computer Science*. Springer Berlin Heidelberg, 2010, pages 266–278. DOI: `10.1007/978-3-642-16926-7_25` (cited on page 3).

[Bor19] K. Borna. “Sweep Line Algorithm for Convex Hull Revisited”. In: *Journal of Algorithms and Computation* 51.1 (2019), pages 1–14. ISSN: 2476-2776. DOI: `10.22059/jac.2019.71276` (cited on page 10).

[Bos+07] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. “Space-efficient geometric divide-and-conquer algorithms”. In: *Computational Geometry* 37.3 (Aug. 2007), pages 209–227. DOI: `10.1016/j.comgeo.2006.03.006` (cited on page 9).

[Bow81] A. Bowyer. “Computing Dirichlet Tessellations”. In: *The Computer Journal* 24.2 (1981), pages 162–166. DOI: `10.1093/comjnl/24.2.162` (cited on pages 16, 20).

[Brö+22] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Pion, and S. Schirra. “2D and 3D Linear Geometry Kernel”. In: *CGAL User and Reference Manual*. 5.5.1. CGAL Editorial Board, 2022. URL: `https://doc.cgal.org/5.5.1/Manual/packages.html#PkgKernel23` (cited on page 79).

[Bun22] Bundesamt für Landestopografie swisstopo. *swissALTI 3D Das hoch aufgelöste Terrainmodell der Schweiz*. Mar. 2022 (cited on page 91).

[Cao+14] T.-T. Cao, A. Nanjappa, M. Gao, and T.-S. Tan. “A GPU Accelerated Algorithm for 3D Delaunay Triangulation”. In: *18th SIGGRAPH Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2014, pages 47–54. ISBN: 978-1-4503-2717-6. DOI: `10.1145/2556700.2556710` (cited on page 16).

[Car+19] L. Caraffa, P. Memari, M. Yirci, and M. Bredif. “Tile & Merge: Distributed Delaunay Triangulations for Cloud Computing”. In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, Dec. 2019. DOI: `10.1109/bigdata47090.2019.9006534` (cited on pages 67, 68).

[CDS12] S.-W. Cheng, T. K. Dey, and J. Shewchuk. *Delaunay mesh generation*. CRC Press, 2012. DOI: `10.1201/b12987` (cited on pages 1, 11, 13, 15, 20, 23).

[CF06] D. Chakrabarti and C. Faloutsos. “Graph mining: Laws, generators, and algorithms”. In: *ACM Computing Surveys* 38.1 (2006), page 2. DOI: `10.1145/1132952.1132954` (cited on page 61).

[CG12] R. Chen and C. Gotsman. “Localizing the Delaunay Triangulation and its Parallel Implementation”. In: *International Symposium on Voronoi Diagrams in Science and Engineering (ISVD)*. IEEE, June 2012, pages 24–31. DOI: `10.1109/isvd.2012.9` (cited on pages 12, 14, 15).

[Cha20] T. M. Chan. “Dynamic Geometric Data Structures via Shallow Cuttings”. In: *Discrete & Computational Geometry* 64.4 (July 2020), pages 1235–1252. ISSN: 1432-0444. DOI: `10.1007/s00454-020-00229-5` (cited on page 94).

[Che10] M.-B. Chen. “The Merge Phase of Parallel Divide-and-Conquer Scheme for 3D Delaunay Triangulation”. In: *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2010, pages 224–230. DOI: `10.1109/ispa.2010.71` (cited on pages 14, 15, 19).

[Chr06] N. Chrisochoides. “Parallel Mesh Generation”. In: *Numerical Solution of Partial Differential Equations on Parallel Computers*. Edited by A. M. Bruaset and A. Tveito. Springer, 2006, pages 237–264 (cited on page 15).

[CHT90] M. S. Chang, N.-F. Huang, and C.-Y. Tang. "An optimal algorithm for constructing oriented Voronoi diagrams and geographic neighborhood graphs". In: *Information Processing Letters* 35.5 (1990), pages 255–260. ISSN: 0020-0190. DOI: 10.1016/0020-0190(90)90054-2 (cited on pages 2, 10, 69, 71, 72, 74, 76, 79, 80).

[Cig+93] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. "Parallel 3D Delaunay Triangulation". In: *Computer Graphics Forum* 12.3 (1993), pages 129–142. DOI: 10.1111/1467-8659.1230129 (cited on pages 14, 15).

[CMS98] P. Cignoni, C. Montani, and R. Scopigno. "DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d". In: *Computer-Aided Design* 30.5 (1998). DOI: 10.1016/s0010-4485(97)00082-1 (cited on pages 12, 14, 15).

[CN00] N. Chrisochoides and D. Nave. "Simultaneous mesh generation and partitioning for Delaunay meshes". In: *Mathematics and Computers in Simulation* 54.4 (2000), pages 321–339. DOI: 10.1016/s0378-4754(00)00192-0 (cited on page 16).

[COM17] COMNAP: Council of Managers of National Antarctic Programs. *Antarctic Station Catalogue.* Online; accessed 22-04-2023. 2017. URL: https://www.comnap.aq/s/COMNAP_Antarctic_Station_Catalogue.pdf (cited on page 109).

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* 3rd edition. The MIT Press, 2009. ISBN: 0262033844 (cited on page 6).

[CS96] N. Chrisochoides and F. Sukup. "Task parallel implementation of the Bowyer-Watson algorithm". In: *International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields.* North-Holland, 1996, pages 773–782 (cited on page 14).

[CSN09] A. Clauset, C. R. Shalizi, and M. E. Newman. "Power-law distributions in empirical data". In: *SIAM review* 51.4 (2009), pages 661–703. DOI: 10.1137/070710111 (cited on page 61).

[Dam18] M. Damian. "Cone-based spanners of constant degree". In: *Computational Geometry* 68 (Mar. 2018), pages 48–61. DOI: 10.1016/j.comgeo.2017.05.004 (cited on pages 2, 69).

[dBer+08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry.* Berlin Heidelberg: Springer, 2008. DOI: 10.1007/978-3-540-77974-2 (cited on pages 3–5, 7–9, 13).

[dBT11] M. de Berg and C. Tsirogiannis. "Exact and approximate computations of watersheds on triangulated terrains". In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, Nov. 2011. DOI: 10.1145/2093973.2093985 (cited on page 89).

[Del34] B. Delaunay. "Sur la sphère vide. A la mémoire de Georges Voronoi". In: *Bulletin de l'Académie des Sciences de l'URSS. Classe des Sciences Mathématiques et Naturelles* 6 (1934), pages 793–800 (cited on pages 1, 11, 13, 20).

[Dev02] O. Devillers. "The Delaunay Hierarchy". In: *International Journal of Foundations of Computer Science* 13.02 (2002), pages 163–180. DOI: 10.1142/s0129054102001035 (cited on page 34).

[dFer11] J. de Ferranti. *Digital elevation data.* Online; accessed 05-12-2023. 2011. URL: http://viewfinderpanoramas.org/dem3.html (cited on pages 91, 99, 100).

[DGJ09] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The shortest path problem: Ninth DIMACS implementation challenge.* Volume 74. American Mathematical Soc., 2009 (cited on page 83).

[DGM09] L. Devroye, J. Gudmundsson, and P. Morin. “On the expected maximum degree of Gabriel and Yao graphs”. In: *Advances in Applied Probability* 41.4 (Dec. 2009), pages 1123–1140. ISSN: 1475-6064. DOI: `10.1239/aap/1261669589` (cited on pages 2, 70).

[DKS08] R. Dementiev, L. Kettner, and P. Sanders. “STXXL: Standard Template Library for XXL Data Sets”. In: *Software Practice & Experience* 38.6 (2008), pages 589–637. DOI: `10.1002/spe.844` (cited on page 96).

[DN17] M. Damian and N. Nelavalli. “Improved bounds on the stretch factor of Y4”. In: *Computational Geometry* 62 (Apr. 2017), pages 14–24. DOI: `10.1016/j.comgeo.2016.12.001` (cited on page 69).

[DP02] O. Devillers and S. Pion. *Efficient Exact Geometric Predicates for Delaunay Triangulations*. Technical report RR-4351. INRIA, Jan. 2002. URL: `https://inria.hal.science/inria-00072237` (cited on page 4).

[EPY97] D. Eppstein, M. S. Paterson, and F. F. Yao. “On Nearest-Neighbor Graphs”. In: *Discrete and Computational Geometry* 17.3 (Apr. 1997), pages 263–282. DOI: `10.1007/pl00009293` (cited on page 13).

[ER59] P. Erdős and A. Rényi. “On Random Graphs I.” In: *Publicationes Mathematicae (Debrecen)* 6 (1959), pages 290–297 (cited on page 61).

[ETW92] H. Edelsbrunner, T. S. Tan, and R. Waupotitsch. “An $O(n^2 \log n)$ Time Algorithm for the Minmax Angle Triangulation”. In: *SIAM Journal on Scientific and Statistical Computing* 13.4 (July 1992), pages 994–1008. DOI: `10.1137/0913058` (cited on page 13).

[Euc56] Euclid. *The Elements of Euclid*. Edited by S. T. L. Heath. 2nd. New York: Dover Publications Inc., 1956. URL: `https://archive.org/details/euclid_heath_2nd_ed` (cited on page 1).

[FB74] R. A. Finkel and J. L. Bentley. “Quad trees a data structure for retrieval on composite keys”. In: *Acta Informatica* 4.1 (1974), pages 1–9. DOI: `10.1007/bf00288933` (cited on pages 7, 94, 96).

[FBF77] J. H. Friedman, J. L. Bentley, and R. A. Finkel. “An algorithm for finding best matches in logarithmic expected time”. In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pages 209–226. DOI: `10.1145/355744.355745` (cited on page 98).

[FHS23a] D. Funke, N. Hüning, and P. Sanders. “A Sweep-Plane Algorithm for Calculating the Isolation of Mountains”. en. In: *31st Annual European Symposium on Algorithms (ESA 2023)*. Edited by I. L. Gørtz, M. Farach-Colton, S. J. Puglisi, and G. Herman. Volume 274. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 51:1–51:17. DOI: `10.4230/LIPIcs.ESA.2023.51` (cited on pages 3, 89, 118).

[FHS23b] D. Funke, N. Hüning, and P. Sanders. *A Sweep-plane Algorithm for Calculating the Isolation of Mountains*. Technical report. Karlsruhe Institute of Technology, May 2023. DOI: `10.48550/ARXIV.2305.08470`. arXiv: `2305.08470 [cs.DS]` (cited on pages 3, 89, 119).

[FLP14] V. Fuetterling, C. Lojewski, and F.-J. Pfreundt. “High-Performance Delaunay Triangulation for Many-Core Computers”. In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association, 2014 (cited on pages 12, 14, 15, 28).

[FM70] W. D. Frazer and A. C. McKellar. “Samplesort: A Sampling Approach to Minimal Storage Tree Sorting”. In: *J. ACM* 17.3 (July 1970), pages 496–507. DOI: `10.1145/321592.321600` (cited on page 34).

[For87] S. Fortune. “A sweepline algorithm for Voronoi diagrams”. In: *Algorithmica* 2.1 (1987), pages 153–174. DOI: `10.1007/bf01840357` (cited on pages 10, 13, 23, 71, 92).

[Fou+07] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. “MPFR: A multiple-precision binary floating-point library with correct rounding”. In: *ACM Transactions on Mathematical Software* 33.2 (June 2007), page 13. DOI: `10.1145/1236463.1236468` (cited on page 4).

[FS17] D. Funke and P. Sanders. “Parallel *d*-D Delaunay Triangulations in Shared and Distributed Memory”. In: *2017 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics (SIAM), 2017, pages 207–217. DOI: `10.1137/1.9781611974768.17` (cited on pages 2, 11, 118).

[FS23a] D. Funke and P. Sanders. *Efficient Yao Graph Construction.* Technical report. Karlsruhe Institute of Technology, Mar. 14, 2023. DOI: `10.48550/arXiv.2303.07858`. arXiv: `2303.07858v1 [cs.CG]` (cited on pages 3, 69, 119).

[FS23b] D. Funke and P. Sanders. “Efficient Yao Graph Construction”. In: *21st International Symposium on Experimental Algorithms (SEA 2023).* Edited by L. Georgiadis. Volume 265. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 20:1–20:20. ISBN: 978-3-95977-279-2. DOI: `10.4230/LIPIcs.SEA.2023.20` (cited on pages 3, 69, 118).

[FSW19a] D. Funke, P. Sanders, and V. Winkler. “Load-Balancing for Parallel Delaunay Triangulations”. In: *Proceedings of Euro-Par 2019.* Edited by R. Yahyapour. Göttingen, Germany: Springer-Verlag, 2019, pages 156–169. DOI: `10.1007/978-3-030-29400-7_12` (cited on pages 2, 11, 118).

[FSW19b] D. Funke, P. Sanders, and V. Winkler. *Load-Balancing for Parallel Delaunay Triangulations.* Technical report. Karlsruhe Institute of Technology, Feb. 20, 2019. DOI: `10.48550/arXiv.1902.07554`. arXiv: `1902.07554v1 [cs.DS]` (cited on pages 2, 11, 119).

[Fun+17] D. Funke, S. Lamm, U. Meyer, P. Sanders, M. Penschuck, C. Schulz, D. Strash, and M. von Looz. *Communication-free Massively Distributed Graph Generation.* Technical report. Karlsruhe Institute of Technology, Oct. 20, 2017. DOI: `10.48550/arXiv.1710.07565`. arXiv: `1710.07565v3 [cs.DC]` (cited on pages 2, 11, 119).

[Fun+18] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. “Communication-Free Massively Distributed Graph Generation”. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, May 2018, pages 336–347. DOI: `10.1109/ipdps.2018.00043` (cited on pages 2, 11, 118).

[Fun+19] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. von Looz. “Communication-Free Massively Distributed Graph Generation”. In: *Journal of Parallel Distributed Computing* 131.C (Sept. 2019), pages 200–217. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2019.03.011` (cited on pages 2, 11, 61, 64, 119).

[FW98] P. Fisher and J. Wood. “What is a mountain? Or the Englishman who went up a Boolean geographical concept but realised it was fuzzy”. In: *Geography* (1998), pages 247–256 (cited on page 92).

[FWC04] P. Fisher, J. Wood, and T. Cheng. “Where is Helvellyn? Fuzziness of multi-scale landscape morphometry”. In: *Transactions of the Institute of British Geographers* 29.1 (2004), pages 106–128. DOI: `10.1111/j.0020-2754.2004.00117.x` (cited on page 92).

[Gai18] Gaia Collaboration. “Gaia Data Release 2 - Summary of the contents and survey properties”. In: *Astronomy & Astrophysics* 616 (2018), A1. DOI: `10.1051/0004-6361/201833051` (cited on pages 41, 83).

[Gil59] E. N. Gilbert. “Random Graphs”. In: *Annals of Mathematical Statistics* 30.4 (Dec. 1959), pages 1141–1144. DOI: `10.1214/aoms/1177706098` (cited on page 61).

[GR69] J. C. Gower and G. J. S. Ross. “Minimum Spanning Trees and Single Linkage Cluster Analysis”. In: *Applied Statistics* 18.1 (1969), page 54. DOI: `10.2307/2346439` (cited on page 13).

[Gri04] P. Grimm. *Die Gebirgsgruppen der Alpen: Ansichten, Systematiken und Methoden zur Einteilung der Alpen*. Deutscher Alpenverein, 2004 (cited on pages 3, 89).

[GS69] K. R. Gabriel and R. R. Sokal. “A New Statistical Approach to Geographic Variation Analysis”. In: *Systematic Zoology* 18.3 (Sept. 1969), page 259. DOI: `10.2307/2412323` (cited on page 13).

[GS78] L. J. Guibas and R. Sedgewick. “A dichromatic framework for balanced trees”. In: *19th Annual Symposium on Foundations of Computer Science*. IEEE, Oct. 1978, pages 8–21. DOI: `10.1109/sfcs.1978.3` (cited on page 7).

[GU93] L. H. Graff and E. L. Usery. “Automated classification of generic terrain features in digital elevation models”. In: *Photogrammetric Engineering and Remote Sensing* 59.9 (1993), pages 1409–1417 (cited on page 92).

[Gut84] A. Guttman. “R-trees”. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD ’84*. ACM Press, 1984, pages 47–57. DOI: `10.1145/602259.602266` (cited on page 7).

[Gwi+10] K. Gwinner, F. Scholten, F. Preusker, S. Elgner, T. Roatsch, M. Spiegel, R. Schmidt, J. Oberst, R. Jaumann, and C. Heipke. “Topography of Mars from global mapping by HRSC high-resolution digital terrain models and orthoimages: Characteristics and performance”. In: *Earth and Planetary Science Letters* 294.3-4 (2010), pages 506–519 (cited on pages 90, 100).

[Hel05] A. Helman. *The Finest Peaks-Prominence and Other Mountain Measures*. Trafford Publishing, 2005 (cited on page 92).

[HJB85] M. T. Heideman, D. H. Johnson, and C. S. Burrus. “Gauss and the history of the fast Fourier transform”. In: *Archive for History of Exact Sciences* 34.3 (1985), pages 265–277. DOI: `10.1007/bf00348431` (cited on page 9).

[HS15] S. Hert and M. Seel. “dD Convex Hulls and Delaunay Triangulations”. In: *CGAL User and Reference Manual*. 4.7. CGAL Editorial Board, 2015 (cited on pages 12, 14, 29, 40, 65).

[HST22] K. Hanauer, C. Schulz, and J. Trummer. “O’Reach: Even Faster Reachability in Large Graphs”. In: *ACM Journal of Experimental Algorithmics* 27 (Oct. 2022), pages 1–27. DOI: `10.1145/3556540` (cited on page 68).

[Hün22] N. Hüning. “A Sweepline Algorithm for Calculating the Isolation of Mountains”. Bachelor’s thesis. Karlsruhe Institute of Technology, Oct. 2022. DOI: `10.5445/IR/1000165010` (cited on pages 89, 120).

[Ise+06] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. “Streaming Computation of Delaunay Triangulations”. In: *ACM Transactions on Graphics* 25.3 (2006), pages 1049–1056. DOI: `10.1145/1179352.1141992` (cited on pages 19, 67).

[Jet19] Jet Propulsion Labratory. *Shuttle Radar Topography Mission (SRTM).* Online; accessed 05-12-2023. 2019. URL: `https://www2.jpl.nasa.gov/srtm/` (cited on page 91).

[Jia04] X. Jia. “Wireless Networks and Random Geometric Graphs”. In: *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2004).* IEEE Computer Society, 2004, pages 575–580. DOI: `10.1109/ISPAN.2004.1300540` (cited on page 61).

[Joe91] B. Joe. “Construction of three-dimensional Delaunay triangulations using local transformations”. In: *Computer Aided Geometric Design* 8.2 (1991), pages 123–142. DOI: `10.1016/0167-8396(91)90038-d` (cited on pages 13, 14).

[Jun+22] J. H. Jungclaus, S. J. Lorenz, H. Schmidt, V. Brovkin, N. Brüggemann, F. Chegini, T. Crüger, P. De-Vrese, V. Gayler, M. A. Giorgetta, et al. “The ICON earth system model version 1.0”. In: *Journal of Advances in Modeling Earth Systems* 14.4 (2022), e2021MS002813 (cited on page 96).

[KdF17] A. Kirmse and J. de Ferranti. “Calculating the prominence and isolation of every mountain in the world”. In: *Progress in Physical Geography* 41.6 (2017), pages 788–802. DOI: `10.1177/0309133317738163` (cited on pages 3, 90, 92, 98–100, 108).

[KK98] G. Karypis and V. Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pages 359–392. DOI: `10.1137/s1064827595287997` (cited on page 38).

[KKŽ05] J. Kohout, I. Kolingerová, and J. Žára. “Parallel Delaunay triangulation in E2 and E3 for computers with shared memory”. In: *Parallel Computing* 31.5 (2005), pages 491–522. DOI: `10.1016/j.parco.2005.02.010` (cited on pages 1, 11, 12, 14, 15).

[KL70] B. W. Kernighan and S. Lin. “An Efficient Heuristic Procedure for Partitioning Graphs”. In: *The Bell System Technical Journal* 49.2 (Feb. 1970), pages 291–307. DOI: `10.1002/j.1538-7305.1970.tb01770.x` (cited on page 38).

[KM17] E. Keogh and A. Mueen. “Curse of Dimensionality”. In: *Encyclopedia of Machine Learning and Data Mining.* Springer US, 2017, pages 314–315. DOI: `10.1007/978-1-4899-7687-1_192` (cited on page 8).

[Kri+10] D. V. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá. “Hyperbolic Geometry of Complex Networks”. In: *CoRR* abs/1006.5169 (2010). DOI: `10.1103/physreve.82.036106` (cited on page 61).

[LAL07] T. Larsson, T. Akenine-Möller, and E. Lengyel. “On Faster Sphere-Box Overlap Testing”. In: *Journal of Graphics, GPU, and Game Tools* 12.1 (2007), pages 3–8. DOI: `10.1080/2151237x.2007.10129232` (cited on pages 29, 39, 40).

[Lam17] S. Lamm. “Communication Efficient Algorithms for Generating Massive Networks”. Master’s thesis. Karlsruhe Institute of Technology, 2017 (cited on pages 11, 12, 61, 62).

[Lo12] S. Lo. “Parallel Delaunay triangulation in three dimensions”. In: *Computer Methods in Applied Mechanics and Engineering* 237-240 (2012), pages 88–106. DOI: `10.1016/j.cma.2012.05.009` (cited on pages 14, 19, 64).

[LPP01] S. Lee, C.-I. Park, and C.-M. Park. "An Improved Parallel Algorithm For Delaunay Triangulation On Distributed Memory Parallel Computers". In: *Parallel Processing Letters* 11 (2001), pages 341–352. DOI: `10.1109/apdc.1997.574023` (cited on pages 12, 14, 15, 29, 41, 60).

[Lum+07] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. "Challenges in parallel graph processing". In: *Parallel Processing Letters* 17.1 (2007), pages 5–20. DOI: `10.1142/S0129626407002843` (cited on page 61).

[Mau84] A. Maus. "Delaunay triangulation and the convex hull of n points in expected linear time". In: *BIT Numerical Mathematics* 24.2 (June 1984), pages 151–163. DOI: `10.1007/bf01937482` (cited on page 64).

[Mea82] D. Meagher. "Geometric modeling using octree encoding". In: *Computer Graphics and Image Processing* 19.2 (June 1982), pages 129–147. DOI: `10.1016/0146-664x(82)90104-6` (cited on page 8).

[MN89] K. Mehlhorn and S. Näher. "LEDA a library of efficient data types and algorithms". In: *Lecture Notes in Computer Science*. Berlin Heidelberg: Springer, 1989, pages 88–106. DOI: `10.1007/3-540-51486-4_58` (cited on page 4).

[MN98] M. Matsumoto and T. Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (1998), pages 3–30. DOI: `10.1145/272991.272995` (cited on page 64).

[Mor66] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966 (cited on page 63).

[MP10] S. Muthukrishnan and G. Pandurangan. "Thresholding random geometric graph properties motivated by ad hoc sensor networks". In: *Journal of Computer and System Sciences* 76.7 (2010), pages 686–696. DOI: `10.1016/j.jcss.2010.01.002` (cited on page 61).

[MPR18] C. Marot, J. Pellerin, and J.-F. Remacle. "One machine, one minute, three billion tetrahedra". In: *International Journal for Numerical Methods in Engineering* (Dec. 2018). DOI: `10.1002/nme.5987` (cited on pages 67, 68).

[MSD16] T. Maier, P. Sanders, and R. Dementiev. "Concurrent Hash Tables: Fast and General?(!)" In: *Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2016, 34:1–34:2. DOI: `10.1145/3309206` (cited on pages 6, 29).

[Mur+10] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. "Introducing the Graph 500". In: *Cray Users Group (CUG)* 19 (2010), pages 45–74 (cited on page 61).

[Nat14] National Center for Geospatial Intelligence Standards. *World Geodetic System 1984: Its Definition and Relationships with Local Geodetic Systems*. Technical report NGA.STND.0036_1.0.0_WGS84. Department of Defense, 2014 (cited on pages 91, 98).

[NR20] C. M. Nguyen and P. J. Rhodes. "Delaunay triangulation of large-scale datasets using two-level parallelism". In: *Parallel Computing* 98 (Oct. 2020), page 102672. DOI: `10.1016/j.parco.2020.102672` (cited on pages 67, 68).

[NS07] G. Narasimhan and M. Smid. *Geometric spanner networks*. Cambridge University Press, 2007. DOI: `10.1017/cbo9780511546884` (cited on pages 3, 71).

[ORe13] G. O'Regan. "John von Neumann". In: *Giants of Computing*. Springer London, 2013, pages 205–208. DOI: `10.1007/978-1-4471-5340-5_44` (cited on page 9).

[Pea23] PeakVisor. *PeakVisor*. Online; accessed 22-04-2023. 2023. URL: `https://peakvisor.com/panorama.html` (cited on page 109).

[PY85] C. Puech and H. Yahia. "Quadtrees, octrees, hyperoctrees". In: *Proceedings of the first annual symposium on Computational geometry - SCG '85*. ACM Press, 1985, pages 272–280. DOI: `10.1145/323233.323268` (cited on page 8).

[Rah+15] Z. Rahmati, M. A. Abam, V. King, S. Whitesides, and A. Zarei. "A simple, faster method for kinetic proximity problems". In: *Computational Geometry* 48.4 (May 2015), pages 342–359. DOI: `10.1016/j.comgeo.2014.12.002` (cited on page 3).

[Rak+21] M.-J. Rakotosaona, P. Guerrero, N. Aigerman, N. Mitra, and M. Ovsjanikov. "Learning Delaunay Surface Elements for Mesh Reconstruction". In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2021. DOI: `10.1109/cvpr46437.2021.00009` (cited on page 16).

[RLA17] M. Rav, A. Lowe, and P. K. Agarwal. "Flood Risk Analysis on Terrains". In: *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, Nov. 2017. DOI: `10.1145/3139958.3139985` (cited on page 89).

[Rod+05] E. Rodriguez, C. Morris, J. Belz, E. Chapin, J. Martin, W. Daffer, and S. Hensley. *An Assessment of the SRTM Topographic Products*. Technical report JPL D-31639. Jet Propulsion Laboratory, 2005 (cited on page 91).

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0201502550 (cited on page 8).

[San+18] P. Sanders, S. Lamm, L. Hübschle-Schneider, E. Schrade, and C. Dachsbacher. "Efficient Random Sampling – Parallel, Vectorized, Cache-Efficient, and Online". In: *ACM Transactions on Mathematical Sofware* 44.3 (Jan. 2018), 29:1–29:14. DOI: `10.1145/3157734` (cited on pages 38, 61, 62).

[San+19] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures*. Springer International Publishing, 2019. DOI: `10.1007/978-3-030-25209-0` (cited on pages 5–7).

[SD95] P. Su and R. L. S. Drysdale. "A Comparison of Sequential Delaunay Triangulation Algorithms". In: *Symposium on Computational Geometry (SCG)*. ACM, 1995, pages 61–70. DOI: `10.1145/220279.220286` (cited on page 12).

[Sei95] R. Seidel. "The upper bound theorem for polytopes: an easy proof of its asymptotic version". In: *Computational Geometry* 5.2 (Sept. 1995), pages 115–116. DOI: `10.1016/0925-7721(95)00013-y` (cited on pages 13, 22).

[SH76] M. I. Shamos and D. Hoey. "Geometric intersection problems". In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE, Oct. 1976, pages 208–215. DOI: `10.1109/sfcs.1976.16` (cited on page 10).

[She05] R. Shewchuk. "Star splaying". In: *Proceedings of the 21st Annual Symposium on Computational Geometry*. ACM, June 2005. DOI: `10.1145/1064092.1064129` (cited on pages 16, 68).

[She96] J. Shewchuk. "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator". In: *Applied Computational Geometry Towards Geometric Engineering* 1148 (1996), pages 203–222. DOI: `10.1007/bfb0014497` (cited on pages 12, 19, 28).

[She97] J. Shewchuk. "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates". In: *Discrete & Computational Geometry* 18.3 (Oct. 1997), pages 305–363. DOI: `10.1007/pl00009321` (cited on pages 4, 29, 40).

[Si+14] W. Si, Q. Tse, G. Mao, and A. Zomaya. "How well do Yao graph and theta graph support Greedy forwarding?" In: *2014 IEEE Global Communications Conference*. IEEE, Dec. 2014, pages 76–81. DOI: `10.1109/glocom.2014.7036787` (cited on pages 2, 69).

[Sin16] D. Sinclair. "A 3D Sweep Hull Algorithm for computing Convex Hulls and Delaunay Triangulation". In: *CoRR* abs/1602.04707 (2016). DOI: `10.48550/arXiv.1602.04707`. arXiv: `1602.04707` (cited on page 10).

[Sla23] G. Slayden. *Peakbagger*. Online; accessed 22-04-2023. 2023. URL: `https://peakbagger.com` (cited on page 108).

[Smi00] M. Smid. "Closest-point problems in computational geometry". In: *Handbook of Computational Geometry*. Elsevier, 2000, pages 877–935. DOI: `10.1016/b978-044482537-7/50021-8` (cited on page 94).

[Smi85] D. R. Smith. "The design of divide and conquer algorithms". In: *Science of Computer Programming* 5 (1985), pages 37–58. DOI: `10.1016/0167-6423(85)90003-6` (cited on page 9).

[SO20] N. Sharp and M. Ovsjanikov. "PointTriNet: Learned Triangulation of 3D Point Sets". In: *Computer Vision – ECCV 2020*. Springer International Publishing, 2020, pages 762–778. DOI: `10.1007/978-3-030-58592-1_45` (cited on page 16).

[SS11] P. Sanders and C. Schulz. "Engineering Multilevel Graph Partitioning Algorithms". In: *Algorithms – ESA 2011*. Edited by C. Demetrescu and M. M. Halldórsson. Berlin, Heidelberg: Springer, 2011, pages 469–480. DOI: `10.1007/978-3-642-23719-5_40` (cited on page 42).

[SS13] P. Sanders and C. Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Proc. of Int. Symp. on Experimental Algorithms (SEA'13)*. Volume 7933. LNCS. Springer, 2013, pages 164–175. DOI: `10.1007/978-3-642-38527-8_16` (cited on pages 38, 40, 42).

[SS16] P. Sanders and C. Schulz. "Scalable generation of scale-free graphs". In: *Information Processing Letters* 116.7 (2016), pages 489–491. DOI: `10.1016/j.ipl.2016.02.004` (cited on page 61).

[ŠT12] S. Štrbac-Savić and M. Tomašević. "Comparative performance evaluation of the AVL and red-black trees". In: *Proceedings of the Fifth Balkan Conference in Informatics*. ACM, 2012. DOI: `10.1145/2371316.2371320` (cited on page 81).

[ST97] H. D. Simon and S.-H. Teng. "How Good is Recursive Bisection?" In: *SIAM Journal on Scientific Computing* 18.5 (Sept. 1997), pages 1436–1445. DOI: `10.1137/s1064827593255135` (cited on page 38).

[STP22] W. Si, Q. Tse, and F. Paradis. "Cone-Based Spanners". In: *CGAL User and Reference Manual*. 5.4. CGAL Editorial Board, 2022. URL: `https://doc.cgal.org/5.4/Manual/packages.html#PkgConeSpanners2` (cited on pages 3, 69, 71, 79, 81).

[Stu12] A. Stukowski. "Structure identification methods for atomistic simulations of crystalline materials". In: *Modelling and Simulation in Materials Science and Engineering* 20.4 (2012), page 045021. DOI: `10.1088/0965-0393/20/4/045021` (cited on pages 29, 61).

[Su+20] T. Su, W. Wang, H. Liu, Z. Liu, X. Li, Z. Jia, L. Zhou, Z. Song, M. Ding, and A. Cui. "An adaptive and rapid 3D Delaunay triangulation for randomly distributed point cloud data". In: *The Visual Computer* 38.1 (Nov. 2020), pages 197–221. DOI: `10.1007/s00371-020-02011-3` (cited on page 67).

[Sut+15] P. Sutter, G. Lavaux, N. Hamaus, A. Pisani, B. Wandelt, M. Warren, F. Villaescusa-Navarro, P. Zivick, Q. Mao, and B. Thompson. "VIDE: The Void IDentification and Examination toolkit". In: *Astronomy and Computing* 9 (Mar. 2015), pages 1–9. DOI: `10.1016/j.ascom.2014.10.002` (cited on page 41).

[SV82] Y. Shiloach and U. Vishkin. "An $\mathcal{O}(n^2 \log n)$ parallel max-flow algorithm". In: *Journal of Algorithms* 3.2 (June 1982), pages 128–146. DOI: `10.1016/0196-6774(82)90013-x` (cited on page 23).

[SVZ07] C. Schindelhauer, K. Volbert, and M. Ziegler. "Geometric spanners with applications in wireless networks". In: *Computational Geometry* 36.3 (2007), pages 197–214. DOI: `10.1016/j.comgeo.2006.02.001` (cited on pages 2, 69).

[SZK13] E. Schubert, A. Zimek, and H.-P. Kriegel. "Geodetic Distance Queries on R-Trees for Indexing Geographic Data". In: *Advances in Spatial and Temporal Databases.* Berlin Heidelberg: Springer, 2013, pages 146–164. DOI: `10.1007/978-3-642-40235-7_9` (cited on page 92).

[Tac+11] T. Tachikawa, M. Hato, M. Kaku, and A. Iwasaki. "Characteristics of ASTER GDEM version 2". In: *2011 IEEE International Geoscience and Remote Sensing Symposium.* 2011, pages 3657–3660. DOI: `10.1109/IGARSS.2011.6050017` (cited on page 91).

[TMF19] R. N. Torres, F. Milani, and P. Fraternali. "Algorithms for mountain peaks discovery: a comparison". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing.* 2019, pages 667–674. DOI: `10.1145/3297280.3297343` (cited on page 92).

[TOG17] C. D. Toth, J. O'Rourke, and J. E. Goodman, editors. *Handbook of Discrete and Computational Geometry.* 3rd edition. Chapman and Hall/CRC, Nov. 2017. DOI: `10.1201/9781315119601` (cited on page 8).

[Tor+18] R. N. Torres, P. Fraternali, F. Milani, and D. Frajberg. "A deep learning model for identifying mountain summits in digital elevation model data". In: *IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE).* IEEE. 2018, pages 212–217. DOI: `10.1109/aike.2018.00049` (cited on page 92).

[Tou80] G. T. Toussaint. "The relative neighbourhood graph of a finite planar set". In: *Pattern Recognition* 12.4 (Jan. 1980), pages 261–268. DOI: `10.1016/0031-3203(80)90066-7` (cited on page 13).

[TPM20] C. Tzovas, M. Predari, and H. Meyerhenke. "Distributing Sparse Matrix/Graph Applications in Heterogeneous Clusters - an Experimental Study". In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC).* IEEE, Dec. 2020. DOI: `10.1109/hipc50609.2020.00021` (cited on page 68).

[TSJ19] X. Tan, C. Sakthip, and B. Jiang. "Improved Stretch Factor of Delaunay Triangulations of Points in Convex Position". In: *Combinatorial Optimization and Applications.* Springer International Publishing, 2019, pages 473–484. DOI: `10.1007/978-3-030-36412-0_38` (cited on page 13).

[vdBer97] G. van den Bergen. "Efficient Collision Detection of Complex Deformable Models using AABB Trees". In: *Journal of Graphics Tools* 2.4 (1997), pages 1–13. DOI: `10.1080/10867651.1997.10487480` (cited on page 39).

[Vit87] J. S. Vitter. "An efficient algorithm for sequential random sampling". In: *ACM Transactions on Mathematical Software* 13.1 (1987), pages 58–67. DOI: `10.1145/23002.23003` (cited on page 62).

[vLTM18] M. von Looz, C. Tzovas, and H. Meyerhenke. “Balanced k-means for Parallel Geometric Partitioning”. In: *Proceedings of the 47th International Conference on Parallel Processing*. ACM, Aug. 2018. DOI: 10.1145/3225058.3225148 (cited on page 68).

[Vor08] G. Voronoi. “Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier mémoire. Sur quelques propriétés des formes quadratiques positives parfaites.” In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* 1908.133 (Jan. 1908), pages 97–102. DOI: 10.1515/crll.1908.133.97 (cited on page 11).

[VS20] D. I. Vassilaki and A. A. Stamos. “TanDEM-X DEM: Comparative performance review employing LIDAR data and DSMs”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 160 (2020), pages 33–50. ISSN: 0924-2716. DOI: 10.1016/j.isprsjprs.2019.11.015 (cited on pages 89, 91).

[Wan+04] D. Wang, Y. Li, B. B. Sun, M. L. Sui, K. Lu, and E. Ma. “Bulk metallic glass formation in the binary Cu–Zr system”. In: *Applied Physics Letters* 84.20 (2004), pages 4029–4031. DOI: 10.1063/1.1751219 (cited on page 29).

[Wat81] D. Watson. “Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes”. In: *The Computer Journal* 24.2 (1981), pages 167–172. DOI: 10.1093/comjnl/24.2.167 (cited on pages 16, 20).

[WGG11] H. Wu, X. Guan, and J. Gong. “ParaStream: A parallel streaming Delaunay triangulation algorithm for LiDAR points on multicore architectures”. In: *Computers & Geosciences* 37.9 (2011), pages 1355–1363. DOI: 10.1016/j.cageo.2011.01.008 (cited on pages 19, 67).

[Win18] V. Winkler. “Load Balancing für Parallele Delaunay-Triangulierung”. Bachelor’s thesis. Karlsruhe Institute of Technology, Mar. 2018. DOI: 10.5445/IR/1000092121 (cited on pages 11, 120).

[Wyl+67] C. Wylie, G. W. Romney, D. C. Evans, and A. Erdahl. “Half-tone perspective drawings by computer”. In: *AFIPS Joint Computer Conferences ’67 (Fall)*. American Federation of Information Processing Societies, 1967. DOI: 10.1109/AFIPS.1967.131 (cited on page 10).

[Yao82] A. C.-C. Yao. “On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems”. In: *SIAM Journal on Computing* 11.4 (1982), pages 721–736. DOI: 10.1137/0211059 (cited on pages 2, 3, 69–72).

[YLX20] W. Yitao, Y. Lei, and S. Xin. “Route Mining from Satellite-AIS Data Using Density-based Clustering Algorithm”. In: *Journal of Physics: Conference Series* 1616.1 (Aug. 2020), page 012017. DOI: 10.1088/1742-6596/1616/1/012017 (cited on page 92).

[Žal05] B. Žalik. “An efficient sweep-line Delaunay triangulation algorithm”. In: *Computer-Aided Design* 37.10 (Sept. 2005), pages 1027–1038. DOI: 10.1016/j.cad.2004.10.004 (cited on page 10).

[Zha+17] X. Zhang, J. Yu, W. Li, X. Cheng, D. Yu, and F. Zhao. “Localized Algorithms for Yao Graph-Based Spanner Construction in Wireless Networks Under SINR”. In: *IEEE/ACM Transactions on Networking* 25.4 (2017), pages 2459–2472. DOI: 10.1109/TNET.2017.2688484 (cited on page 69).