Load-Balancing for Parallel Delaunay Triangulations

Daniel Funke, Peter Sanders, and Vincent Winkler

Karlsruhe Institute of Technolgy, Karlsruhe, Germany
{funke, sanders}@kit.edu, vincent.winkler@student.kit.edu

Abstract. Computing the Delaunay triangulation (DT) of a given point set in $\mathbb{R}^D$ is one of the fundamental operations in computational geometry. Recently, Funke and Sanders [11] presented a divide-and-conquer DT algorithm that merges two partial triangulations by re-triangulating a small subset of their vertices – the *border* vertices – and combining the three triangulations efficiently via parallel hash table lookups. The input point division should therefore yield roughly equal-sized partitions for good load-balancing and also result in a small number of border vertices for fast merging. In this paper, we present a novel divide-step based on partitioning the triangulation of a small sample of the input points. In experiments on synthetic and real-world data sets, we achieve nearly perfectly balanced partitions and small border triangulations. This almost cuts running time in half compared to non-data-sensitive division schemes on inputs exhibiting an exploitable underlying structure.

1 Introduction

The Delaunay triangulation (DT) of a given point set in $\mathbb{R}^D$ has numerous applications in computer graphics, data visualization, terrain modeling, pattern recognition and finite element methods [15]. Computing the DT is thus one of the fundamental operations in geometric computing. Therefore, many algorithms to efficiently compute the DT have been proposed (see survey in [23]) and well implemented codes exist [13, 20]. With ever increasing input sizes, research interest has shifted from sequential algorithms towards parallel ones.

Recently, we presented a novel divide-and-conquer (D&C) DT algorithm for arbitrary dimension [11] that lends itself equally well to shared and distributed memory parallelism and thus hybrid parallelization. While previous D&C DT algorithms suffer from a complex – often sequential – divide or merge step [8, 17], our algorithm reduces 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 efficiently via hash table lookups. All steps required for the merging – identification of relevant vertices, triangulation and combining the partial DTs – are performed in parallel.

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. In this paper 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 input point set accordingly.

The paper is structured as follows: we review the problem definition, related work on partitioning for DT algorithms and our D&C DT algorithm from [11] in Section 2. Subsequently, our proposed divide-step is described in Section 3, along with a description of fast intersection tests for the more complexly shaped partition borders and implementation notes. We evaluate our algorithms in Section 4 and close the paper with conclusions and an outlook on future work in Section 5.

2 Preliminaries

2.1 Delaunay Triangulations

Given a D-dimensional point set $\mathbf{P} = \{p_1, p_2, \dots, p_n\} \subset \mathbb{R}^D$ for all $i \in \{1, \dots, n\}$, a triangulation $T(\mathbf{P})$ is a subdivision of the convex hull of $\mathbf{P}$ into D-simplices such that the set of 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 $\mathbf{P}$. 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 can be computed in $\mathcal{O}(n \log n)$ time for $D = 2$ and $\mathcal{O}(n^{\lceil \frac{D}{2} \rceil})$ time for $D \geq 3$.

2.2 Related Work

Many algorithms for the parallel construction of the DT of a given point set have been proposed in the literature. They generally fall into one of two categories: parallel incremental insertion and D&C approaches. We will focus on a review of the divide-step of the latter. A more comprehensive discussion of both algorithm types is given in [11].

Aggarwal et al. [1] propose the first parallel D&C DT algorithm. They partition the input points along a vertical line into blocks, which are triangulated in parallel and then merged sequentially. The authors do not prescribe how to determine the location of the splitting line. Cignoni et al. [8] partition the input along cutting (hyper)planes and firstly construct the simplices of the triangulation crossing those planes before recursing on the two partitions. The remaining simplices can be created in parallel in the divided regions without further merging. The authors mention that the regions should be of roughly equal cardinality, but do not go into the details of the partitioning. Chen [5] and Lee et al. [17] explicitly require splitting along the median of the input points. Whereas the former uses classical splitting planes, the latter traces the splitting line with Delaunay edges, thus eliminating the need for later merging.

The subject of input partitioning has received more attention in the meshing community. 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. Chrisochoides [6] 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 [7] propose an algorithm that meshes the subproblems via incremental insertion using the Bowyer-Watson algorithm.

2.3 Parallel Divide-and-Conquer DT Algorithm

Recently, we presented a parallel D&C algorithm for computing the DT of a given point set [11]. Our algorithm recursively divides the input into two partitions which are triangulated in parallel. The contribution lies in a novel merging step for the two partial triangulations which re-triangulates a small subset of their vertices and combines the three triangulations via parallel hash table lookups. For each partial triangulation the *border* is determined, i. e. the simplices whose circumhypersphere intersects the bounding box of the other triangulation. The vertices of those border simplices are then re-triangulated to obtain the border triangulation. The merging proceeds by combining the two partial triangulations, stripping the original border simplices and adding simplices from the border triangulation iff i) they span multiple partitions; or ii) are contained within one partition but exist in the same form in the original triangulation.

The algorithm's sensitivity to the input point distribution is twofold: the partitions need to be of equal size for good load-balancing between the available cores and the number of simplices in the border needs to be minimized in order to reduce merging overhead. As presented in [11], the algorithm splits the input into two partitions along a hyperplane. Three strategies to choose the splitting dimension are proposed: i) constant, predetermined splitting dimension; ii) cyclic choice of the splitting dimension – similar to k-D trees; or iii) dimension with largest extend. This can lead to imbalance in the presence of non-homogeneously structured inputs, motivating the need for more sophisticated partitioning schemes.

3 Sample-based Partitioning

In this paper, we propose more advanced strategies for partitioning the input points than originally presented in [11]. The desired partitioning addresses both data sensitivities of our algorithm. The underlying idea is derived from sample sort: gain insight into the input distribution from a (small) sample of the input. Algorithm 1 describes our partitioning procedure. 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, where the sample triangulation is used to speed up point location queries [10].

Algorithm 1 partitionPoints($\mathbf{P}, k$): partition input into k partitions.

Input: points $\mathbf{P} = \{p_1, \ldots, p_n\}$ with $p_i \in \mathbb{R}^D$, number of partitions k
Output: partitioning $\big(\mathbf{P}_1 \ldots \mathbf{P}_k\big)$

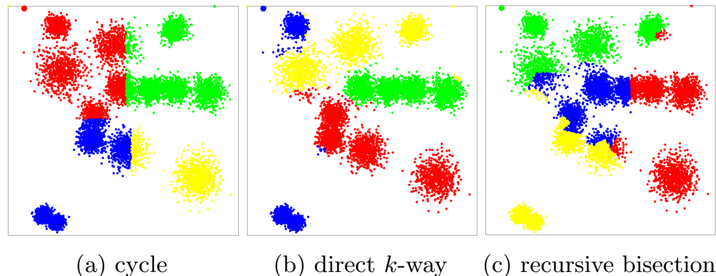
1: $\mathbf{P}_S \leftarrow$ choose $\eta(n)$ from $\mathbf{P}$ uniformly at random ▷ $\eta(n)$ sample size
2: $T \leftarrow$ Delaunay($\mathbf{P}_S$)
3: $G = (V, E, \omega)$ with $V = \mathbf{P}_S$, $E = T$ and weight function ω
4: $\big(V_1 \ldots V_k\big) \leftarrow$ partition(G) ▷ partition graph
5: $\big(\mathbf{P}_1 \ldots \mathbf{P}_k\big) \leftarrow \big(\varnothing \ldots \varnothing\big)$
6: **parfor** $p \in \mathbf{P}$ **do**
7: $\quad v_n \leftarrow \arg\min_{v \in \mathbf{P}_S} ||p - v||$ ▷ find nearest sample point to p
8: $\quad \mathbf{P}_i \cup= p$ with $i \in [1 \ldots k] : v_n \in V_i$ ▷ assign p to v_n's partition
9: **return** $\big(\mathbf{P}_1 \ldots \mathbf{P}_k\big)$

Instead, we transform the 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 tool.

The choice of the weight function ω influences the quality of the resulting partitioning. As mentioned in Section 2.3, 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. In Section 4.1 we evaluate several suitable edge weight functions.

Given the partitions of the sample vertices $(V_1 \ldots V_k)$, the partitioning needs to be extended to the entire input point set. The dual of the Delaunay triangulation of the sample point set – its Voronoi diagram – defines a partitioning of the Euclidean space $\mathbb{R}^D$ in the following sense: each point $p_{S,i}$ of the sample is assigned to a partition $j \in [1 \ldots k]$. Accordingly, its Voronoi cell with respect to $\mathbf{P}_S$ defines the sub-space of $\mathbb{R}^D$ associated with partition j. In order to extend the partitioning to the entire input point set, each point $p \in \mathbf{P}$ is assigned to the partition of its containing Voronoi cell.

All steps in Algorithm 1 can be efficiently parallelized. Sanders et al. [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. [2] 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 Algorithm 1.



(a) cycle (b) direct k-way (c) recursive bisection

Fig. 1: Example of a two-dimensional partitioning with four partitions for 10 000 points and a sample size of 1000.

3.1 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 [22] prove that recursive bisection can lead to arbitrarily bad partitions and Kernighan and Lin [14] confirm the superiority of direct k-way partitioning experimentally. However, recursive bisection is still widely – and successfully – used in practice (e. g. for initial partitioning in KaHIP [19]). We therefore consider both strategies to obtain k partitions for our DT algorithm.

The partitioning schemes originally proposed in [11] can be seen as recursive bisection: the input is recursively split along the median. The splitting dimension is chosen in a cyclic fashion, similiar to k-D trees. Figure 1a shows an example.

Similarly, our new partitioning algorithm 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 1c.

Using direct k-way partitioning, only one partitioning and one merge step is required. 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 1b depicts an example partitioning.

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.

3.2 Geometric Primitives

Our D&C algorithm [11] 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 can be performed using the fast box-sphere overlap test of Larsson et al. [16]. However, using the more advanced partitioning algorithms presented in this paper, 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 intersection tests:[1] i) each partition is crudely approximated with an axis-aligned *bounding box* and the fast intersection test of Larsson et al. [16] is used 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. ii) for each partition it is determined which cells of a *uniform grid* are occupied by points from that partition. This allows for a more accurate test whether a given simplex s of partition i intersects with partition j by determining whether any of j's occupied grid cells are intersected by the circumhypersphere of s, again using the box-sphere intersection test [16]. To further accelerate the intersection test we build an AABB tree [4] on top of the grid data structure. iii) to *exactly* determine the necessary points for the border triangulation we use the previous test to find the grid cells of partition j intersected by the circumhypersphere of s and then use an adaptive precision *inSphere*-test [21] for all points contained in these cells to test whether s violates the Delaunay property and thus its vertices need to be added to the border triangulation.

4 Evaluation

Batista et al. [3] 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. Furthermore, Lee et al. [17] suggest normally distributed input points around d) the center of the unit cube; and e) several "bubble" centers, distributed uniformly at random within the unit cube. We furthermore test our algorithm with a real world dataset from astronomy. The Gaia DR2 catalog [9] 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 the image next to Table 1 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

[1] For a more detailed description of the primitives we refer to the technical report [12].

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

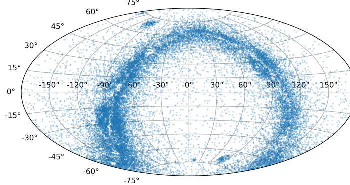


Table 1: Input point sets and their resulting triangulations. Running times are reported for $k = t = 16$, parallel KaHIP, $\eta(n) = \sqrt{n}$, grid-based intersection test with $c_{\mathcal{G}} = 1$ and logarithmic edge weights. The image on the right shows an Aitoff projection of a random sample of 25 000 sources from the Gaia DR2 dataset.

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

Table 2: Parameters of our algorithm (top) and conducted experiments (bottom).

performed in three-dimensional space ($D = 3$). Table 1 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.

Implementation Notes: We integrated our divide-step into the implementation of [11], which is available as open source.[2] We use KaHIP [19] and its parallel version [2] as graph partitioning tool. The triangulation of the sample point set is computed sequentially using CGAL [13] with exact predicates.[3]

4.1 Parameter Studies

The parameters listed in Table 2 can be distinguished into configuration parameters of our algorithm and parameter choices for our experiments. In our parameter

[2] `https://git.scc.kit.edu/dfunke/DelaunayTriangulation`
[3] `CGAL::Exact_predicates_inexact_constructions_kernel`
** unless otherwise stated in Table 1

study we examine the configuration parameters of our algorithm 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 times, we use it as indicator for our parameter tuning. We use the uniform, normal, ellipsoid and random bubble distribution for our parameter tuning and compare against the originally proposed cyclic partitioning scheme for reference. Due to space constrains we refer to the technical report [12] for an in-depth discussion of each parameter individually and only present a short summary here.

Our experiments show, that a sample size of $\eta(n) = \sqrt{n}$ balances the approximation quality of a partitioning of the final triangulation with the runtime for the sample triangulation. Considering edge weights, dense regions of the input point set are reflected by *many* short edges in the sample triangulation. Therefore, even constant edge weights result in a sensible partitioning. However, logarithmic edge weights[4] are better when there is an exploitable structure in the input points. For KaHIP we chose the parallel configuration as default as it requires a similar runtime to the ECO configuration while achieving a cut only slightly worse then STRONG. The grid-based intersection test with a cell size of $c_{\mathcal{G}} = 1$ shows the best trade-off between accuracy – i. e. only essential simplices are included in the border triangulation – and runtime for the geometric primitive itself.

4.2 Partitioning Quality

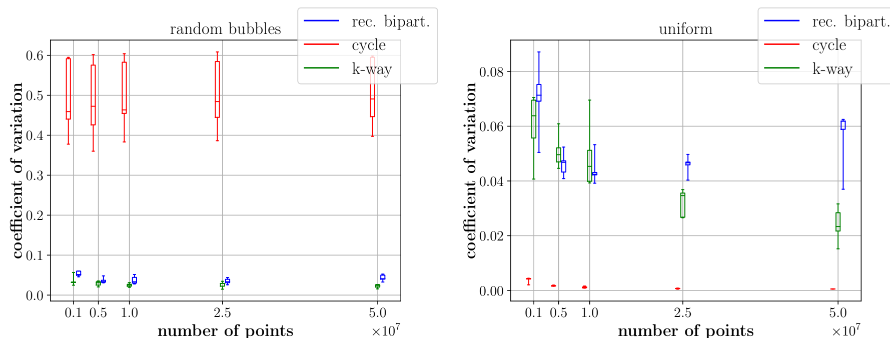
Given a graph partitioning $\left(V_1 \ldots V_k\right)$, its quality is defined by the weight of its cut, $\sum_{e \in C} \omega(e)$ for $C := \{e = (u,v), e \in E \text{ and } u \in V_i, v \in V_j \text{ with } i \neq j\}$. As mentioned in Section 3, the balance of the graph partitioning is ensured 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: i) the deviation from the ideal partition size and ii) the coefficient of variation of the partition sizes. Due to space constraints we only discuss the latter measure for two of our input distributions here. We refer to the technical report [12] for the full discussion.

The coefficient of variation c_v of the partition sizes p_i, $i \leq k$, is given by

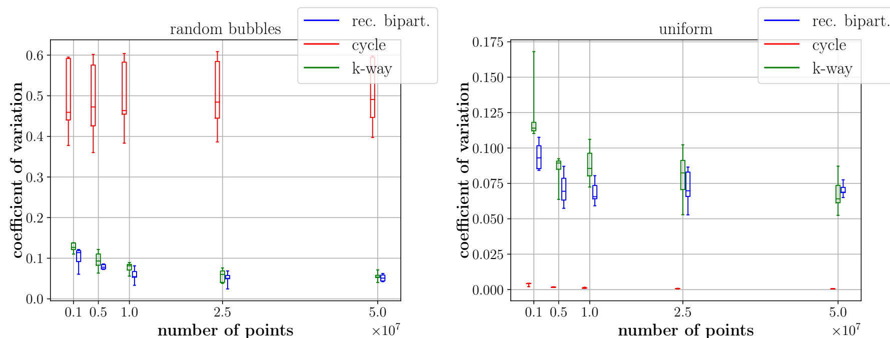
$$c_v = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{\sum_{i \leq k} (p_i - \mu)^2}{k-1}}}{\frac{\sum_{i \leq k} p_i}{k}}.$$

Figure 2 shows c_v for two different sample sizes and two of our input distributions.. For all distributions, our sample-based partitioning scheme robustly achieves a c_v of ≈6 % and ≈12 % for sample sizes $\sqrt{n}$ and $0.01n$, respectively. 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

[4] $\omega(e = (v,w)) = -\log d(v,w)$ width $d(\cdot)$ denoting the Euclidean distance.



(a) sample size $\eta(n) = 0.01n$



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

Fig. 2: Coefficient of variation of the partition sizes for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.

random seeds. Moreover, the deficits of the original cyclic 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.

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 determined by our D&C algorithm. For direct k-way partitioning, only one sample and one border triangulation are necessary; for recursive bisectitioning there are a total of $k-1$ of each. Figure 3 shows the overtriangulation factor for two different sample sizes and two of our input distributions. For all distributions, the larger sample size reduces the oversampling factor. As the partitioning of the larger sample DT more closely resembles the

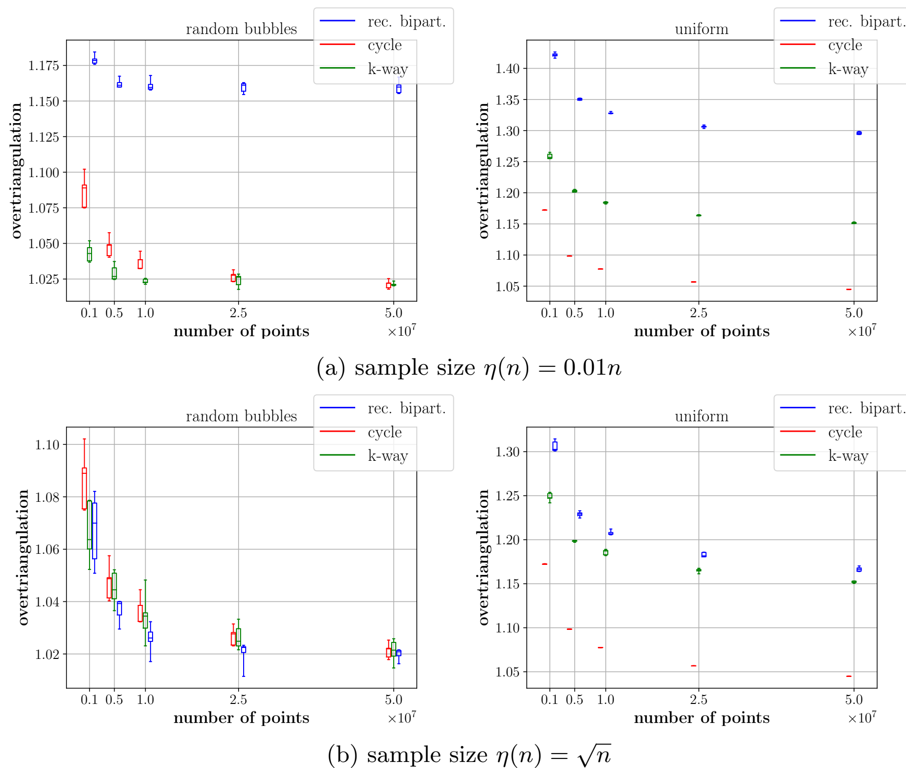


Fig. 3: Overtriangulation factor for $k = t = 16$, parallel KaHIP, logarithmic edge weights and grid-based intersection test with $c_{\mathcal{G}} = 1$.

partitioning of the full DT, the number of points in the border triangulation is reduced. For the random bubble distribution, the overtriangulation factor is on par or below that of the original cyclic partitioning scheme. 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 partitioning scheme. This results in more circumhyperspheres intersecting another partition and thus the inclusion of more points in the border triangulation. Our experiments with the exact intersection test primitive confirm this notion.

4.3 Runtime Evaluation

We conclude our experiments with a study of the runtime of our D&C algorithm with the new sample-based divide step against the originally proposed cyclic division strategy, its k-way variant – called “flat cycle” – as well as the parallel incremental insertion algorithm of CGAL. Figure 4 shows the total triangulation time for our fixed choice of configuration parameters.

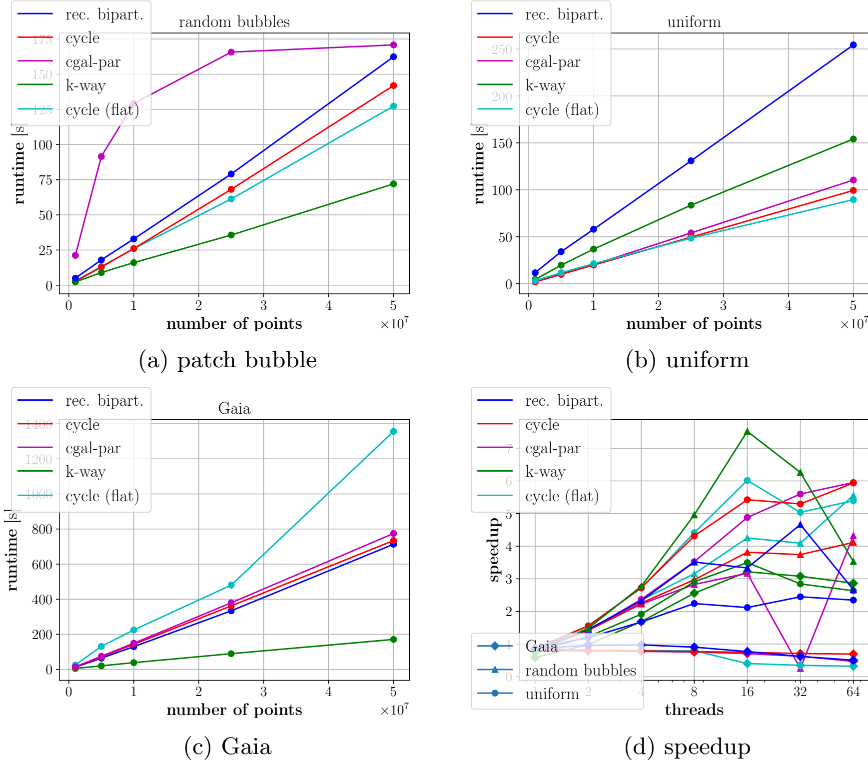
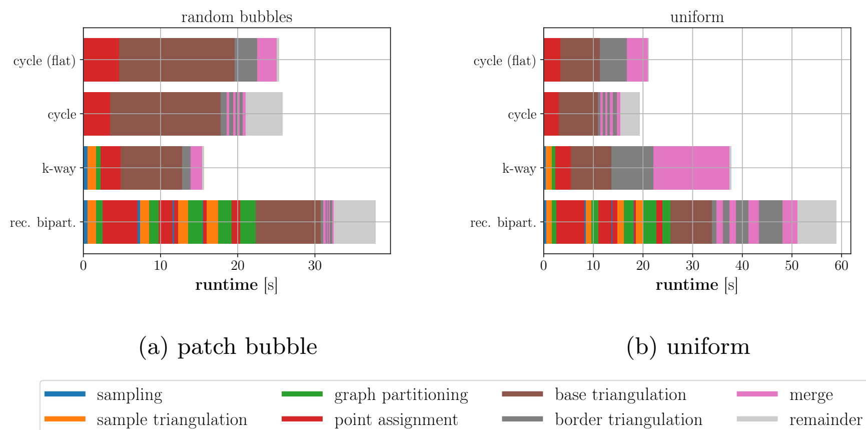


Fig. 4: Runtime evaluation for $k = t = 16$, parallel KaHIP, $\eta(n) = \sqrt{n}$, grid-based intersection test with $c_{\mathcal{G}} = 1$ and logarithmic edge weights. Absolute speedup over sequential CGAL for $k = t$ and all distributions tested with 50×10^6 points.

Direct k-way partitioning performs best on the random bubbles distribution, with a speedup of up to 50 % over the cyclic partitioning scheme. Considering the flattened cycle 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. 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 $o_{DT} = 1.15$ for k-way partitioning compared to $o_{DT} = 1.05$ for cyclic 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 5b.

Of particular interest is the scaling behavior of our algorithm with an increasing number of threads. Figure 4d shows a strong scaling experiment. The absolute



(a) patch bubble

(b) uniform

Fig. 5: 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.

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 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, it 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 size of the input is not sufficient to be efficiently split into 64 partitions.

Considering our real world dataset, the direct k-way partitioning scheme also exhibits the best scaling behavior. As illustrated in the image next to Table 1, 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.

Clearly, direct k-way partitioning outperforms recursive bisection in every configuration. Following the theoretical considerations in Section 3.1 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.

Figure 5 shows a breakdown of the algorithm runtime for our fixed choice of configuration parameters. 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.

5 Conclusions

We present a novel divide-step for the parallel D&C DT algorithm presented in [11]. 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 almost a factor of two speedup over our previous partitioning scheme 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. The sample-based divide step can also be integrated into our distributed memory algorithm presented in [11], where the improved load-balancing and border size reduces the required communication volume for favorable inputs.

Acknowledgments The authors gratefully acknowledge the Deutsche Forschungsgemeinschaft (DFG) who partially supported this work under grants SA 933/10-2 and SA 933/11-1.

Bibliography

[1] Aggarwal, A., Chazelle, B., Guibas, L.: Parallel computational geometry. Algorithmica **3**(1), 293–327 (1988)
[2] Akhremtsev, Y., Sanders, P., Schulz, C.: High-quality shared-memory graph partitioning. In: Euro-Par, pp. 659–671, Springer (2018)
[3] Batista, V.H., Millman, D.L., Pion, S., Singler, J.: Parallel geometric algorithms for multi-core computers. Comp. Geometry **43**(8), 663–677 (2010)
[4] van den Bergen, G.: Efficient collision detection of complex deformable models using aabb trees. J. of Graphics Tools **2**(4), 1–13 (1997)
[5] Chen, M.B.: The Merge Phase of Parallel Divide-and-Conquer Scheme for 3D Delaunay Triangulation. In: Int. Symp. on Parallel and Distributed Processing with Applications (ISPA), pp. 224–230, IEEE (2010)
[6] Chrisochoides, N.: Parallel mesh generation. In: Num. Solution of PDEs on Parallel Computers, pp. 237–264, Springer (2006)
[7] Chrisochoides, N., Nave, D.: Simultaneous mesh generation and partitioning for delaunay meshes. Math. and Comp. in Sim. **54**(4), 321 – 339 (2000)
[8] Cignoni, P., Montani, C., Scopigno, R.: DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d. CAD **30**(5) (1998)

[9] Collaboration, G.: Gaia data release 2. summary of the contents and survey properties. arXiv (abs/1804.09365) (2018)

[10] Devillers, O.: The delaunay hierarchy. Int. J. of Foundations of Computer Science **13**(02), 163–180 (2002)

[11] Funke, D., Sanders, P.: Parallel d-d delaunay triangulations in shared and distributed memory. In: ALENEX, pp. 207–217, SIAM (2017)

[12] Funke, D., Sanders, P., Winkler, V.: Load-Balancing for Parallel Delaunay Triangulations. arXiv (abs/1902.07554) (2019)

[13] Hert, S., Seel, M.: dD convex hulls and delaunay triangulations. In: CGAL User and Reference Manual, CGAL Editorial Board, 4.7 edn. (2015)

[14] Kernighan, B.W., Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs. The Bell System Technical Journal **49**(2), 291–307 (Feb 1970)

[15] Kohout, J., Kolingerová, I., Žára, J.: Parallel Delaunay triangulation in E2 and E3 for computers with shared memory. Par. Comp. **31**(5), 491–522 (2005)

[16] Larsson, T., Akenine-Möller, T., Lengyel, E.: On Faster Sphere-Box Overlap Testing. J. of Graphics, GPU, and Game Tools **12**(1), 3–8 (2007)

[17] Lee, S., Park, C.I., Park, C.M.: An improved parallel algorithm for delaunay triangulation on distributed memory parallel computers. Parallel Processing Letters **11**, 341–352 (2001)

[18] Sanders, P., Lamm, S., Hübschle-Schneider, L., Schrade, E., Dachsbacher, C.: Efficient parallel random sampling – vectorized, cache-efficient, and online. ACM Trans. Math. Softw. **44**(3), 29:1–29:14 (Jan 2018)

[19] Sanders, P., Schulz, C.: Think Locally, Act Globally: Highly Balanced Graph Partitioning. In: Proc. of Int. Symp. on Experimental Algorithms (SEA'13), LNCS, vol. 7933, pp. 164–175, Springer (2013)

[20] Shewchuk, J.: Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. Applied Comp. Geometry Towards Geometric Engineering **1148**, 203–222 (1996)

[21] Shewchuk, J.: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Disc. & Comp. Geom. **18**(3), 305–363 (Oct 1997)

[22] Simon, H.D., Teng, S.H.: How good is recursive bisection? J. on Scientific Computing **18**(5), 1436–1445 (Sep 1997)

[23] Su, P., Drysdale, R.L.S.: A comparison of sequential delaunay triangulation algorithms. In: Symp. on Comp. Geometry (SCG), pp. 61–70, ACM (1995)