Space Partitioning for Distributed Surface Reconstruction

Project in Visual Computing

—

E193 – Institute of Visual Computing and Human-Centered Technology

Lukas Brunner[*]
TU Wien
Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
(Advisor)
TU Wien
Projektass.in Diana Marin, BSc MEng.
(Assistance)
TU Wien
Projektass.(FWF) Dr. Stefan Ohrhallinger, B.A.
(Assistance)
TU Wien

Abstract

Delaunay triangulation, as one of the fundamental problems in computational geometry, has been vastly explored in scientific research as it often constitutes the bottleneck in composite algorithms if no acceleration structures are used. In recent years however, data sets grew successively larger due to technological advancements such as in photogrammetry, 3D laser scanning or scientific simulation methods. Thus, distributed algorithms gained increasing attention as the main method to deal with data quantities in the terra- or petabyte range in a reasonable amount of time.

This work investigates prospects of speeding up a state-of-the-art surface reconstruction algorithm (not yet published) by splitting up the underlying required Delaunay tetrahedralization using efficient uniform space partitioning. In a distributed execution environment, the constraints that a valid Delaunay graph must satisfy typically lead to the expensive exchange of points among the distributed compute nodes. By exploiting traits of the surface reconstruction algorithm, the implementation can relax the constraints for the distributed Delaunay tetrahedralization to gain performance and trade formal correctness with a minor and visually acceptable error bound. Moreover, the implementation deals with the technical challenges of incorporating the complete storage hierarchy to support data sets that exceed main memory capacities, as well as being usable in both local and distributed execution environments.

The evaluation focuses on an analysis of the runtime performance of the space partitioned implementation for large planetary surface reconstruction with respect to latency, storage efficiency and summed processing time.

Keywords: Surface Reconstruction, Delaunay Triangulation, Space Partitioning

1 Introduction

When executing algorithms on large datasets, problems commonly arise because of super-linear space and/or time complexities. Many optimal algorithms suitable for practical applications on large datasets belong to the $O(n \log n)$ category due to relying on efficient sorting algorithms for sovling the most complex parts. However in practice, large constant factors can outweigh the asymptotic complexity for practical applications. Especially below $O(n^2)$ other factors, such as I/O latency or data distribution and lock contention can significantly alter the real runtime behaviour.

This work focuses on the surface reconstruction algorithm BALLFILTER, a version of BALLMERGE, both developed by Stefan Ohrhallinger [2022] and also worked on in [Komon 2022]. It relies on the computation of the full Delaunay tetrahedralization as a first step. The application uses the asymptotically optimal CGAL implementation which runs theoretically in $O(n \log n)$ [Devillers et al. 2022]. Although already using shared memory parallel programming[1], this step constitutes the major part of surface reconstruction time, as Section 5 will show (see Fig. 8). With an overall $O(n \log n)$ complexity, for further speedups we focused mainly focused on improving latency instead of overall execution time by adapting the solution to distributed parallel execution. Space partition-

[*]Visual Computing (UE 066932) – Matr. Nr.: 11909464 – Email: lukas.brunner@student.tuwien.ac.at

[1]When building CGAL with TBB (`https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.htm`)

ing is an effective and convenient way to achieve this, as it also naturally accommodates large datasets that exceed the main memory of a single compute node.

1.1 Problem

Dividing a large dataset into smaller spatially coherent subsets (we use rectangular tiles) and processing them by the same surface reconstruction algorithm theoretically increases efficiency due to the super-linear time complexity. Yet, this quantity turns out to be negligible in practice, because distributed processing incurs some non-negligible overhead for work distribution and merging of results. Especially in the case of the distributed computation of the Delaunay tetrahedralization, each node may need information from a possibly extended neighborhood of surrounding tiles [Peterka et al. 2015]. If all tiles can be processed in parallel at the same time, this kind of communication can be quickly realized with libraries such as MPI. However, these dependencies pose an architectural challenge once the dataset is large enough to exceed the working memory of a cluster. Resorting to complex scheduling of individual tiles or out-of-core algorithms incurs a significant performance hit [Caraffa et al. 2021]. Due to these factors, a distributed Delaunay tetrahedralization may decrease latency while in practice actually increasing the overall processing time. As will be shown in this report, the presented implementation is an extreme example where for large data less subdivision is ultimately more efficient because latency only decreases slowly or almost not at all, depending on the problem size.

1.2 Idea

In light of the above it is fortunate that one property of BALLFILTER allows to circumvent the exchange of points among the tiles, making the processing of tiles completely independent from each other. By including all points within a certain area around each tile, a visually acceptable or even indistinguishable result can be obtained. Komon [2022] states the size of this margin based on a padding that is calculated as:

$$\text{padding}(l, \delta) = \frac{2l}{\sqrt{4\delta - \delta^2}}$$

where l was chosen based on the length of the bounding box diagonal and δ is a parameter of BALLFILTER. Section 5.2 starts with a short discussion of this value.

At the time of subdividing a dataset into tiles, we can already include those points that are potentially necessary for the correctness of the critical parts of the Delaunay graph. This is done by including with the data of each tile all points that fall within a certain margin around the tile before processing them independently with BALLFILTER. According to Komon, the margin is calculated for each side of the tile as follows:

$$\text{margin}(s, l, \delta) = \begin{cases} \text{padding}(l, \delta) + l, & \text{if } s > 0 \\ \text{padding}(l, \delta) \quad , & \text{if } s < 0 \end{cases}$$

where s is positive for the margin in positive direction along any dimension and negative for the margin in negative direction. Note that the margin is the same across all dimensions, always larger in direction of the respective basis vector and independent of the size of the tile.

In summary, the subdivision scheme has two main benefits:

- Independent distributed processing: It allows serial and parallel processing of parts of the data in any order.
- Under the assumption of a sufficiently uniform sample density, it allows to choose the subdivision parameters such that the data of a tile is guaranteed to fit into main memory. This simplifies the implementation of algorithms and opens more potential for program optimization.

For all that, one pays the time for running the splitting step, described in Sections 3 and 4, and duplicate points being processed multiple times during reconstruction, as discussed by [Komon 2022]. Smaller tiles exhibit decreasing storage efficiency and causes overhead during surface reconstruction due to more duplicated points. Komon assumes the number of duplicate points to be asymptotically negligible which works in practice for a low number of subdivisions. But since *margin* is constant across increasing values for the *split* parameter and assuming a uniform distribution of points, the number of points in the margin is proportional to the summed border length of all tiles:

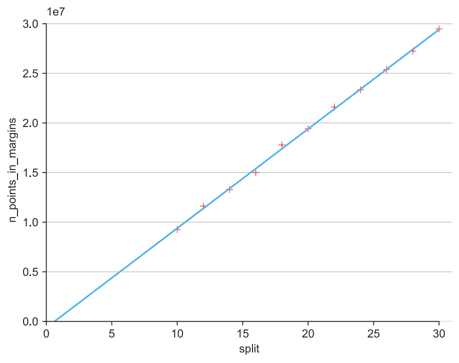
$$N_{margin} \propto split * 2d * a^d \qquad (*)$$

for a subdivision of a $(d+1)$-dimensional hypercube with a units of d-dimensional border into *split* parts along each axis. Fig. 2 illustrates this.

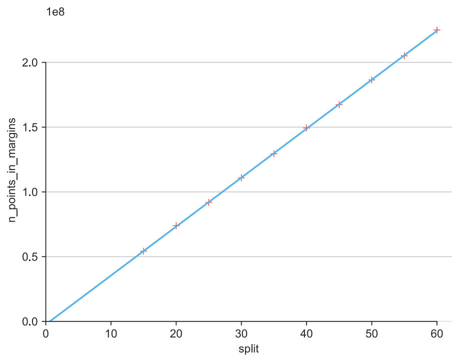
This means that the number of points in the margins proportionally increases not only with the sample density, but also with the *split* value (see Fig. 1). The number of points added per additional subdivision in x- and y-direction amounts to $1.00e^6$ Points for the Birmingham example and $3.78e^6$ Points for the Cambridge example, introduced in Section 5.1. For 60 subdivisions on two axes this accounts already for roughly 10% of all data in the Cambridge example (see Fig. 11). These considerations underline the importance of this overhead when choosing parameters in a practical deployment of such subdivision schemes.

1.3 Definitions

"Latency" will refer to the wall elapsed real time when processing tiles in parallel with a theoretically infinite



(a) Birmingham



(b) Cambridge

Figure 1: The total number of duplicated points in the margins increases linearly with the *split* parameter.

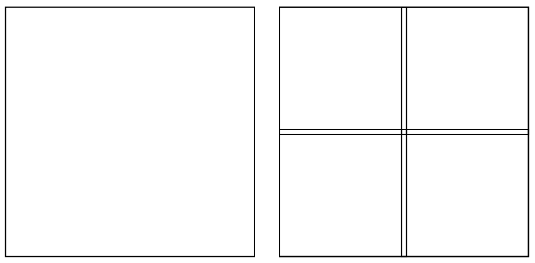


Figure 2: Schematic illustration of adding one subdivision to a 2D tile in both dimensions. Let the side length of a square-shaped dataset be a units. *Left*: the single tile has $4a$ units of border; *Right*: The borders of the four smaller tiles sum up to $4a + 4a$ units. Any further subdivision with another horizontal and vertical split adds another $4a$ units.

number of nodes (full parallelization). The "total time" or "overall time" will refer to the sum of the processing time spent on these hypothetical nodes. Since experiments presented in Section 5 ran sequentially on a single PC, this quantity is roughly equal to the actual duration of the experiments. "Individual time" (e. g. "individual reconstruction times") will refer to the time needed to process one individual tile.

2 Previous Work

Komon [2022] evaluated a distributed version of BALLFILTER on an HPC cluster, using the implementation for the subdivision explained in Sections 3 and 4. The results presented in Section 5 add a new contribution on top of those in [Komon 2022], because on the one hand the size of datasets and the core count per node differ, and on the other hand the referenced paper focuses on evaluating mainly latency, including the overhead of distributed processing. This report takes a different approach and disaggregates the empirical runtimes into their algorithmic subcomponents, unveiling different overall scaling behaviour for below roughly 2M Points versus above. Next to data size, latency – as investigated in [Komon 2022] – is a major motivation for moving computation into a distributed parallel environment. Time cannot be altered, but computational resources can. Nonetheless, for large data under hard financial constraints, time may become the more flexible variable because the additional overall computational resources spent on the overhead of a distributed solution translates into hardware and especially energy costs. Hence, this report extends previous work to show the actual necessary effort for achieving improved latencies.

3 Concept

As already indicated, the implementation features two sequential steps:

1. The splitting step: The program responsible for this step is called *Splitter* and the algorithm it executes will be called SPLIT. Its input is the whole point cloud dataset. Its output is a set of files (*Tile* files), where each file contains the data of a rectangular subregion of the input dataset plus additional data from the surrounding neighborhood as indicated in Section 1.2.

2. The reconstruction step: The is step executes BALLFILTER independently on every individual Tile file. It maps every input Tile file to one output PLY file with the reconstructed surface.

As the central practical contribution, this section of the report will focus on the Splitter. Both conceptually as well as the implementation can be decomposed into two substeps:

Scatter: This step contains several subparts: file input, sort points (GPU), cache to disk.

Resolve: This step contains two subparts: resolve tile, distribute margin points.

3.1 Scatter

The Splitter processes the input dataset by reading the dataset file in chunks. The chunks can be read in any order, which theoretically allows any arbitrary input data stream to work just as well. Every chunk consists of a number of points that will be sorted into the cells of a regular grid where each cell represents one tile. Within each cell, points are additionally sorted into 3^d subregions as shown in Fig. 3. Note that the larger margins are towards the negative direction of each dimension. It owes to the fact that these regions contain the points that will be added to the opposite side of the neighboring tiles in the Resolve step. The overall indexing scheme used for sorting and re-

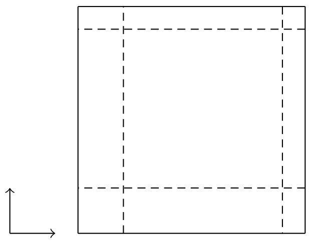


Figure 3: Subregions within a cell for $d = 2$.

trieving the respective data again in the Resolve step is the linearized cell index times 3^d plus the linearized subregion index. The sorted data is appended to one intermediate file per tile, leaving out tiles that do not contain any points. These files carry the extension `.data` and contain, after the Scatter step, one slice of data (potentially of zero size) per each chunk from the input dataset. There will also be one `.bins` file per `.data` file containing 3^d offsets for each slice of input data to identify neighborhoods of margin areas during the Resolve step.

3.2 Resolve

The Resolve step is sequenced after the Scatter step and begins with resolving the slices from each the `.data` file into the the first part of the final Tile file (with `.tile` extension) as shown in Fig. 4. The `.tile` file will contain the data per each subregion in a contiguous data section and add the according information in the Tile file header.

The second Resolve step distributes the margin points among the tiles. It simply iterates over the neighboring tiles for each tile and copies the relevant subregions to the neighbors.

4 Implementation

The implementation assumes an upper limit on the sample density of the source dataset, and that a *split* value exists such that each tile resulting from SPLIT is guaranteed to fit into main memory. This means in practice that the maximum data per tile should be a constant fraction of the available main memory as SPLITTER needs to maintain

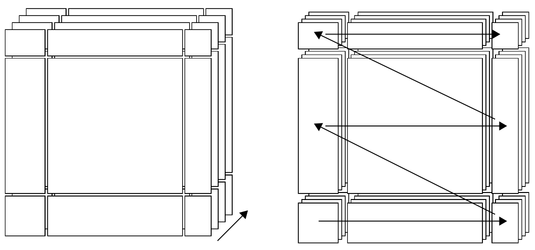


Figure 4: Schematic depiction of the Resolve step. The arrows indicate the order in which data occurs in the file. *Left*: The `.data` file contains consecutive slices. Each slice contains 3^d (possibly empty) parts of the points for the respective subregions; *Right*: The `.tile` file contains (after the header section) 3^d (possibly empty) regions with all data aggregated for the respective subregion.

additional information per tile as well as potential copies of the data. The additional information per tile, e. g. the structure of intermediate results necessary for the Scatter step, may take up a small fraction of memory in practice, but it will scale in $O(\frac{n}{split^d})$ in the worst case – only one sample ending up in each tile for every slice of the input dataset.

Since SPLIT is sequenced before BALLFILTER, its execution time linearly increases both the overall processing time as well as the latency of a parallelized processing of a monolithic input dataset. Naturally, the relative impact is expected to be larger on the theoretically shorter latency than on the longer overall execution time, despite the practical results in Section 5.2. Thus, Splitter tries to exploit as much parallelism as possible, while keeping the implementation efforts reasonable for a proof of concept. See Section 6 for further improvements.

4.1 Multithreading

The Intel Thread Building Blocks library(*TBB*) was used to manage most of the algorithmic complexity arising from processing a large dataset that exceeds the working memory in a parallel shared memory environment. The limited main memory requires scheduling and managing dependencies among partial results. TBB Flow Graph allows the specification of such dependencies and gives control over concurrency limits for parts of the data flow graph through which a maximum total main memory consumption can be established.

TBB is non-preemptive in the sense that its scheduler may only cancel pending tasks or wait for a task to finish before assigning a new task for execution on one of its threads. Therefore, blocking waits (e. g. file IO using the C++ Standard Library or waiting for the result of a CUDA kernel call) within TBB threads can significantly waste CPU cycles that could be spent on other tasks. On the one hand, the OS scheduler is yet free to preempt TBB tasks running on the underlying OS threads and a large TBB thread pool (exceeding the number of hardware threads) can thus serve as a hasty performance optimization, especially when scheduled tasks wait on synchronization primitives (usually for accessing a non-threadsafe resource) or explicitly yield their thread (typically also as part of waiting on a synchronization primitive). On the other hand however, this behaviour cannot be well controlled and oversubscribing will lead to frequent context switches that significantly deteriorate performance in situations where blocking waits are not an issue and threads would normally spend most of their time on their actual task. The solution to this problem is to introduce the asynchronous notion of the problematic processes explicitly using TBB concepts. The async_node decouples the API for processing input data from the API for providing an output. In the example of executing a CUDA kernel, it passes control back to the TBB runtime after asynchronously submitting a kernel call and remembering a so-called *gateway* object. This proxy object can then be used from a CUDA callback (`cudaLaunchHostFunc`) enqueued directly after kernel call submission to notify TBB of the result. By a restriction of how CUDA host callbacks work, the callback itself may not directly retrieve the result. It may only pass back to the TBB runtime the necessary information for retrieving the data in a downstream node.

In the other identified bottleneck, file I/O, the async_node cannot solve the problem, for it needs an asynchronous API that could be integrated with the TBB. The C++ Standard Library does not expose such API. Even though the implementation uses thread-local instances of `std::ifstream`, at least with the Microsoft STL, reading from those streams in parallel still synchronizes – and thus serializes – the calls under the hood. The integration of a library that provides asynchronous file I/O is kept for future work due to additional implied implementation costs (see Section 6).

Assuming that the input file might be stored in a contiguous physical location, the implementation chose to read continuous regions for each thread, spread equally along the file. The rationale was to potentially spread intensive reads over multiple NAND flash memory chips. Due to the aforementioned serialization of reads, the effectivity of this approach could not be determined. Future research might be able to evaluate the performance difference between asynchronous ”pseudo-sequential” reads (every thread asynchronously requesting the globally next slice, where every thread might have to seek) and spread out sequential reads (every thread reading continuous data without seeking).

4.2 GPU sorting

The actual sorting is implemented in CUDA as it parallelizes well on the GPU. The sorting step can be sped up by several orders of magnitude using COUNTINGSORT as initially introduced by Hoetzlein [2014]. While not implemented, this approach opens the door to easily apply additional point-wise transformations (e. g. affine transformations to relocate the frame of reference) in parallel on the fly while sorting the data.

4.3 Caching of Intermediate Results

Eventually, intermediate results must be buffered outside the main memory. Values reported in Section 5 are based on caching to a local SSD. Nevertheless, network drives or fast cloud storage guaranteed to reside within the same high-speed network would just work as well – with a respective impact on the runtime performance. In either case, the critical factor will be the random write performance as it typically provides the worst throughput and results on e. g. HDDs have shown prohibitively bad performance during the Scatter step of the Splitter.

While the Scatter step is conceptually easy to parallelize, as all chunks are independent, the Resolve step involves multiple sequential parts: Resolving `.data` and `.bins` files into Tile files as shown in Fig. 4 can be parallelized across tiles. This step and the next step, which distributes the margin points among the tiles, are inherently serial. Distributing the margin points involves a nested loop. The outer loop enumerates the $3^d - 1$ potential neighbors that each tile has. The inner loop iterates over all Tile files and can be parallelized. For each Tile file it appends the required margin points from the current *source* Tile file to the Tile file of the neighboring cell (the *target* File) and updates the necessary information in the *target* file header. It is possible to parallelize over all Tiles because:

- Point data copied from the *source* file by the *source* thread is appended to the *target* file and does not affect the data read by the *target* thread from the *target* file.
- The file header in the *target* file is written by only one *source* thread at a time because the mapping of cells to neighboring cells in a given direction is a bijection. Furthermore, all read-write pairs of any *source* thread from and to the *target* file header do not overlap with one another. The changes which any *source* thread applies to the header of a *target* tile file do not affect the parts of the header which the *target* thread may read and use.

The code for addressing cells, subregions, neighbors and identifying neighborhoods (both on the GPU and CPU) is written for the variable n-dimensional case. It uses compile-time evaluation of `constexpr` functions to precompute arrays of indices, offsets and respective array lengths to efficiently address and iterate regions within an n-D grid during runtime. One more complex case uses static initialization to compute an array of indices, to enumerate discontinuous ranges of n-D subregions touching the lower-dimensional hyperrectangles in the border of an n-D-tile, e. g. for a 3D tile: neighboring subregions coinciding with a face, along an edge or touching one of the corner vertices. The practical usage of this code is limited by an $O(C^d)$ complexity for the resulting executable size as well as compilation/initialization time. The discussion of the individual constructs and the underlying geometry is beyond the scope of this report.

5 Results

Experiments were run locally on a Windows 11 machine with the following configuration:

- AMD Ryzen 7 3700X 8-Core Processor (16 Threads) @ 3.59GHz (locked), Caches: 512KB L1, 4MB L2, 32MB L3
- 32 GB RAM @ 2666MHZ (4x Corsair CMK16GX4M2Z3600C18 DDR4)
- 1TB Samsung SSD 970 EVO Plus
- NVIDIA GeForce GTX 1080 Ti

Overclocking features were turned off, to avoid unpredictable effects due to dynamic behaviour or non-spec utilization of hardware. Additionally before running the Splitter and between the Splitter and the reconstruction stage, data potentially cached by the OS was flushed to remove ”magic speedups” through the Windows SuperFetch cache.

The evaluation does not run a fully distributed version of BALLFILTER, like DISTRIBUTEDBALLFILTER in [Komon 2022], but executes the same core BALLFILTER implementation used within DISTRIBUTEDBALLFILTER sequentially on the individual tiles resulting from a call to SPLIT without the additional scheduling, MPI communication and final merging step in [Komon 2022]. An approximation to latency in a theoretical parallel execution is subsequently reconstructed from the logged time measurements and does not take the runtime overhead of the a real distributed execution (e. g. final merging step etc.) into account.

5.1 Dataset

This report uses the SensatUrban dataset of [Hu et al. 2022] for the evaluation. It comes in multiple PLY files split across a training and a test set. For the purpose of this work they were combined into two monolithic PLY files, one for each geographic location, and used as input to SPLIT:

- Birmingham: 8.5 GB, 570M Points, 1273.69 m $\times$ 1306.75 m $\times$ 72.64 m, approx. $456\frac{Points}{m^2}$
- Cambridge: 31.8 GB, 2.139B Points, 2152.94 m $\times$ 2136.91 m $\times$ 105.81 m, approx. $657\frac{Points}{m^2}$

These two point clouds feature a realistic[2] point distribution and are suitable for this evaluation due to the following reasons:

- During development, it can be used in parts, while not introducing major changes in the point distributions.
- It is representative for a real world use-case of surface reconstruction: TIN meshes of planetary surfaces reconstructed from point clouds are used in products like Google Maps, Microsoft Flight Simulator or many professional geographic information systems.
- It is large enough to be represent data exceeding the working memory for both CPU and GPU, requiring streaming upon ingestion. It is small enough to still run tests sequentially on a single machine with consumer hardware.
- Other datasets using terrestrial LiDAR scanning, capture the scene from discrete points of view, like e.g. the Semantic3D dataset [Hackel et al. 2017]. They have unfavorable data distributions as most samples are concentrated in a small region around the capturing device with a linear falloff in world-space sampling frequency, heavily suffering from occlusion due to the perspective and frequently static capturing locations.

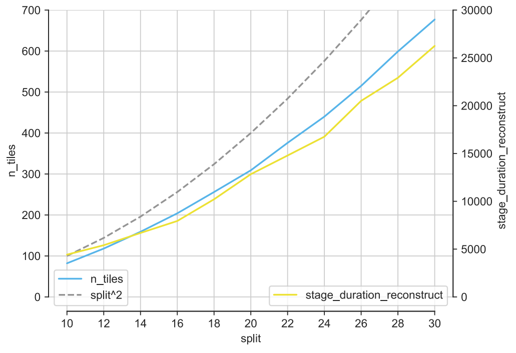
However in return, SensatUrban is not suitable for 3D subdivision as most data is spread across the xy-plane. While the implementation works for 3D tiles just fine, this evaluation focuses on the application in planetary surface reconstruction and thus poses a 2.5D scenario, where subdivision happens along 2 axes while the dataset occupies the third dimension as well. The two example input files have an approximate sample density ratio of 1.44[3], which good to keep in mind as it is a fundamental reason for the following performance figures to differ between the Birmingham and the Cambridge example.

5.2 Measurements

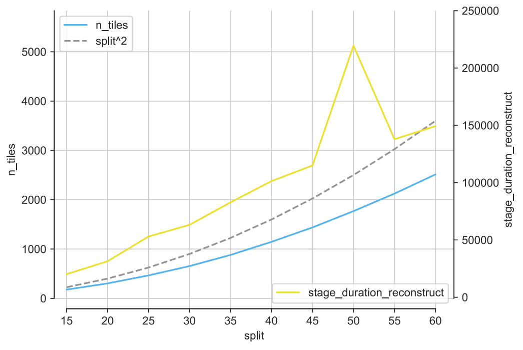
Fig. 5 shows the total running time for each *split* value. Ideally the total running time would stay constant. Despite some overhead for processing more files would be expected, the runtime increase in Fig. 5 is exceptionally large. In light of this visualization and as already mentioned in Section 1, less subdivision seems to result in a vastly more efficient use of electrical energy. Section 5.2.2 goes into detail to show where this unexpected behaviour is coming from.

[2]As opposed to synthetic, since it had been produced through photogrammetry from aerial image data, see [Hu et al. 2022] for details.

[3]The sample density for both example input files (see beginning of Section 5.1) was calculated as the number of points devided by the summed area of all tiles which contained samples on the respective highest tested subdivision.



(a) Birmingham



(b) Cambridge

Figure 5: The number of Tile files (blue/left axis) plotted against the total reconstruction time for each *split* value (yellow/right axis).

5.2.1 Splitting

SPLIT was always run with equal values for the number of tiles in x- and y-direction and a constant 1 in z-direction. The parameter varying the subdivison in x- and y-direction is called *split*. The number of tiles resulting form SPLIT increases in $O(split^2)$ (see Fig. 5). In comparison to the total running time, the Splitter takes up only a minor fraction (see Fig. 6). Nevertheless, it can account for roughly 40 to 70 precent of overall latency (see Fig. 7 and 8). This sequential structure is what makes performance optimisations within the Splitter significant. The most important steps of the Splitter are unsurprisingly the Scatter and Resolve steps. Since I/O speed is known to be the bottleneck in the Splitter, the graphs in Fig. 7 coincide with the amount of I/O required in each step. The Scatter duration stays rather constant with a ratio between the Birmingham and the Cambridge example according to their file size. The Scatter duration dominates consistently with larger amounts of smaller tiles.

5.2.2 Reconstruction

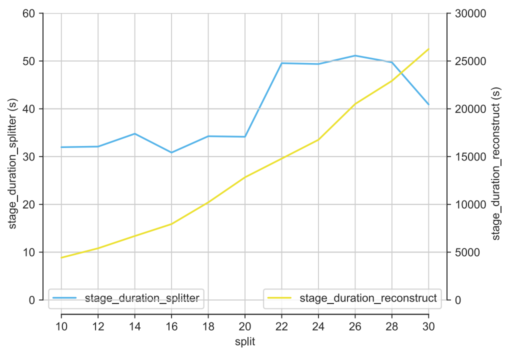
An essential parameter of BALLFILTER is the quantity l mentioned in 1.2. The original implementation establishes this value based on the length of the bounding box diagonal as $l = \frac{diagonal}{c}$, c being a constant. Komon chooses $c = 2000$. The following experiments use $c = 5000$. In fact, this value depends on the sampling frequency and thus using the diagonal of the bounding box is just a heuristic. Finding a value for c systematically, could start by computing the empirical cumulative distribution function of edge lengths $F_{\mathbf{X}}(x)$ ($\mathbf{X}$ the random variable describing the length of edges) and the average vertex degree $\deg_{\text{mean}}(G)$ in the full Delaunay graph G. Then one could choose the value c s.t. $F_{\mathbf{X}}(c) = \frac{6}{\deg_{\text{mean}}(G)}$, assuming that the average number of neighbors in a planar Delaunay graph asymptotically approaches 6 [Tanemura 2003]. A fully probabilistic model might even improve the value but is beyond the scope of this report. Also calculating a perfect value for c involves iterating or sampling the full Delaunay graph which can add significant processing time. A more efficient way is just estimating c with previous knowledge about the sampling rate based on the respective capturing process.

Regarding the odd scaling behaviour shown in Fig. 5 we start with a simple overview, since it emerges from multiple factors:

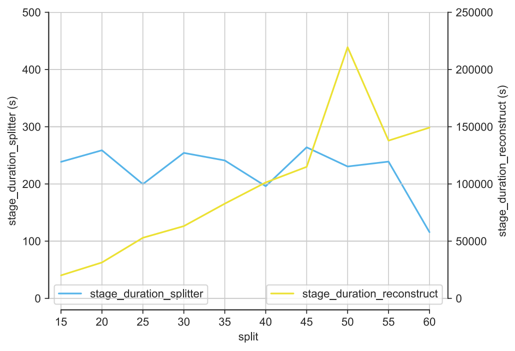
If the individual reconstruction times do not decrease with increasing subdivision, then there is no advantage of distributing work, other than subdividing data size enough to fit tiles into main memory. This would be the case if Fig. 5 would show quadratic behaviour for the total reconstruction time, because any constant factor C, which the total reconstruction time might grow less than the number of tiles, becomes negligible in practice when spread across the quadratically increasing number of tiles. However, Fig. 5 adumbrates linear (5b) or slightly super-linear (5a) behaviour for the total running time. This means that eventually one gets a return on the increased efforts for parallelization in the form of decreased latency, albeit possibly little and in this example more for 5b than 5a as can be seen in Fig. 8.

Despite decreasing latency has been shown above, Fig. 5 and 8 do not show a scaling behaviour that might naïvely be expected from an $O(n \log n)$ algorithm. Going more into the detailed measurements, multiple components contribute to the emerge:

1. Increasing lock contention on smaller tiles prevents the Delaunay tetrahedralization to show $O(n \log n)$ scaling in practice. Fig. 9 rather shows an $\Omega(\sqrt{n})$ shape in the lower range which grows significantly faster than $O(n \log n)$ with very low problem sizes. the average tetrahedralization time reaches a peak just below 200M Points for both the Birmingham and the larger Cambridge example.

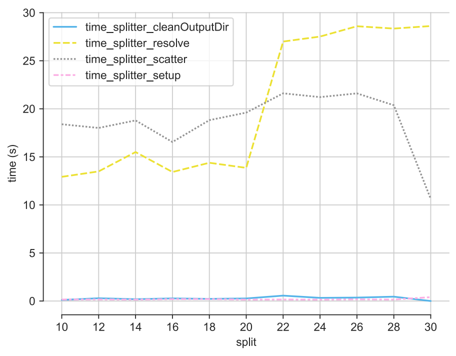


(a) Birmingham

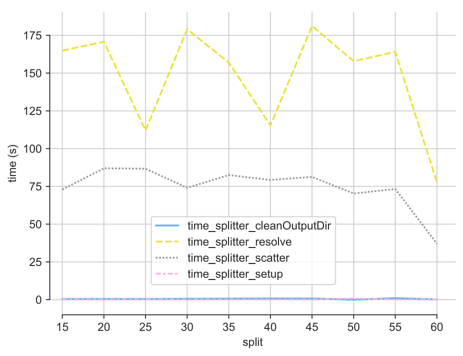


(b) Cambridge

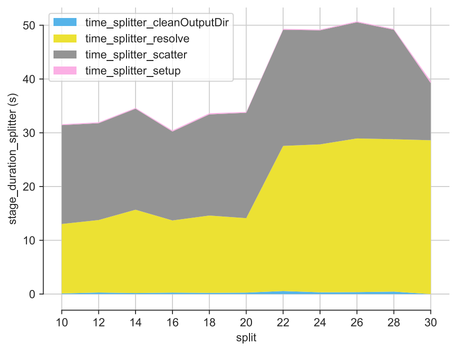
Figure 6: Timings for the total duration of the Splitter and the reconstruction step. The scale factor between the blue axis (timing for the Splitter) and the yellow axis (time spent for reconstruction all Tile files) is 50 for both figures.



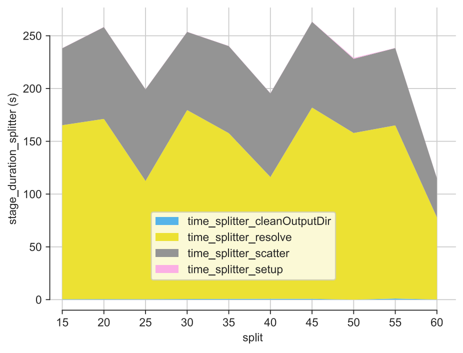
(a) Birmingham



(b) Cambridge

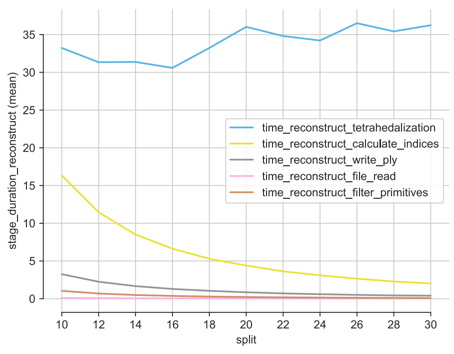


(c) Birmingham

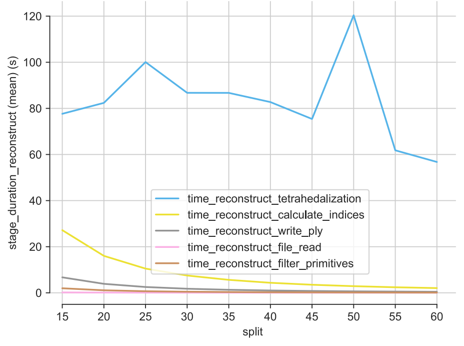


(d) Cambridge

Figure 7: Timings for the Splitter per each *split* value. *Top:* individual algorithmic components; *Bottom:* Stacked components.

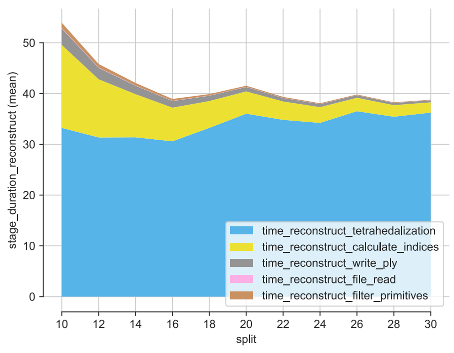


(a) Birmingham

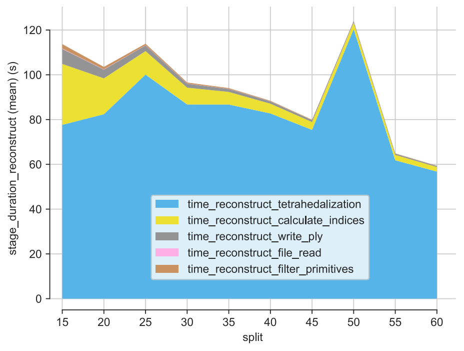


(b) Cambridge

Individual algorithmic components: The quadratic decreasing components are easily visible.



(c) Birmingham



(d) Cambridge

Stacked components: The mean reconstruction time per *split* value shows only moderate overall improvements due to the dominant tetrahedralization execution time.

Figure 8: Timings for the surface reconstruction averaged over all files per *split* value.

2. The ratio of inner points to margin points is decreasing with smaller tiles, see Fig. 10. When plotting the storage efficiency as:

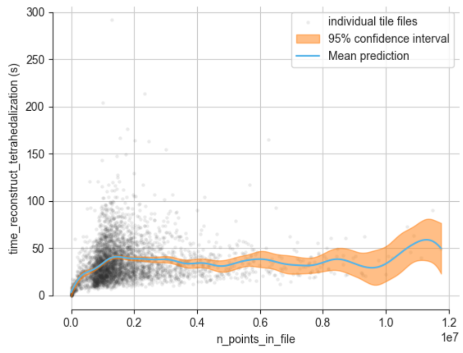
$$storage\_efficiency = \frac{n\_points\_in\_tile \text{ Points} * 16 \frac{\text{Bytes}}{\text{Point}}}{tile\_file\_size \text{ Bytes}}$$

then we also see the expected decreasing values in Fig. 11.

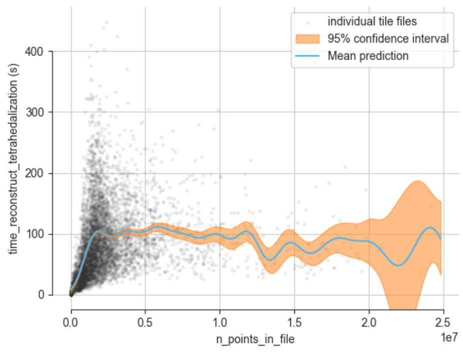
3. Other components of the reconstruction algorithms (I/O, Delaunay filtering, index buffer calculation) perfectly adhere to $\theta(n)$. They quadratically decrease with increasing *split* values (see Fig. 8) because the number of tiles subdividing the same volume is a quadratic function of the *split* value and thus the number of points per tile decreases in the shape of $\frac{C}{split^2}$.

Factors1 and 2 deteriorate the efficiency of the distributed surface reconstruction for increasing *split* values and thus decreasing tile sizes. Factor 3 decreases in the naturally expected way for decreasing file sizes. The joint effect of these different trends can be seen in Fig. 8: the average reconstruction time decreases modestly since the Delaunay tetrahedralization looses efficiency with smaller tiles. The peak of the inefficiency of the CGAL tetrahedralization seems to be located around 500M Points where it diverges most from a hypothetical $n \log n$ scaling. This leads to the overall unfavorably increasing reconstruction time in Fig. 12. Fig. 13 shows the practically relevant latency which could be achieved on the two datasets. For a 100% reconstruction the latency even worsens with more subdivisions. If outliers are disregarded and incomplete results are tolerated, only then better latencies could be achieved – however just for a (large) subset of the data, as in the nature of the tradeoff.

An in-depth analysis by running BALLFILTER on a small set of Tile files of different sizes under Intel's VTune Profiler showed a tendency of decreasing spinning and overhead CPU time for larger Tile files. But also larger files showed a still increased overhead towards the end of building the Delaunay graph. Due to inserting points into the Delaunay tetrahedralization in the same order as they appear in the Tile file, this coincides with the insertion of the margin points. Because this manual profiling

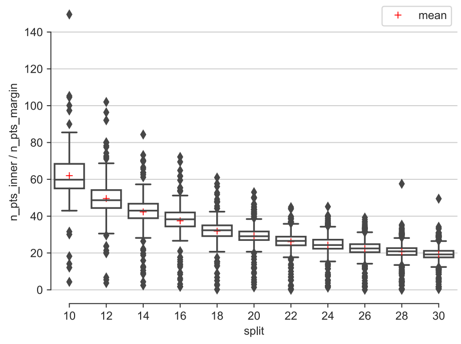


(a) Birmingham

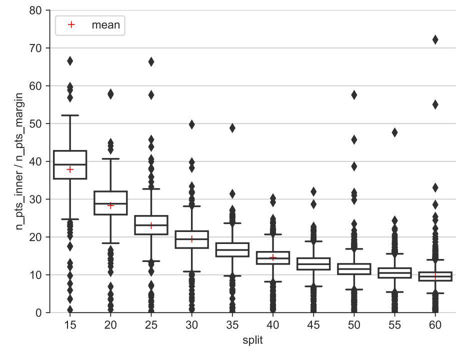


(b) Cambridge

Figure 9: Durations for the tetrahedralization in BALLFILTER of all tiles depending on the number of points per file. A Gaussian Process regression reveals the general trend. The predictions become noisy higher up the x-axis due to the non-uniform distribution of samples. But the empirical suboptimal behaviour of the CGAL tetrahedralization with a spatial lock grid of 50^3 cells for low numbers of points is clearly visible.

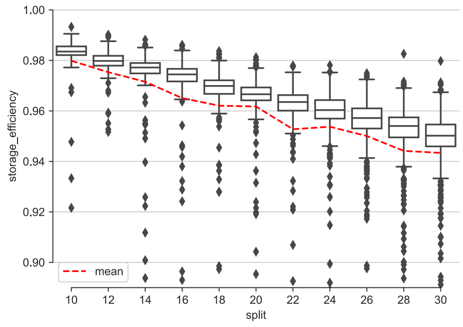


(a) Birmingham

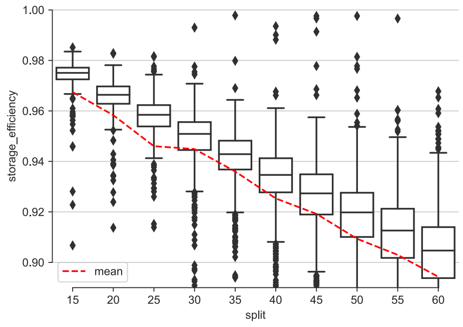


(b) Cambridge

Figure 10: Decreasing ration of number of points in the tile to point in the margin for each *split* value.

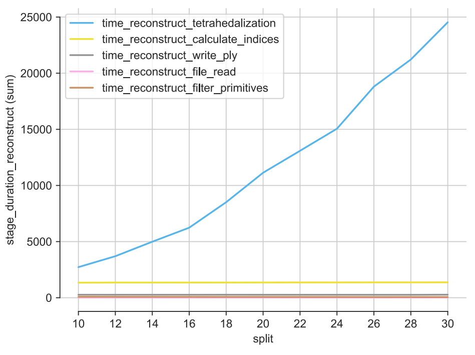


(a) Birmingham

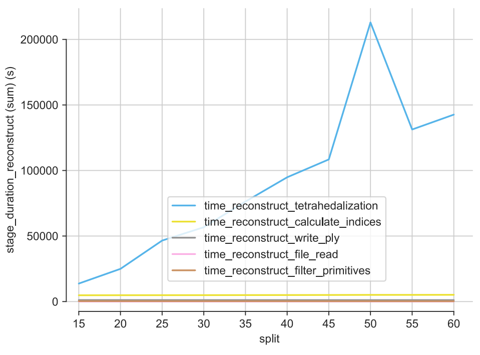


(b) Cambridge

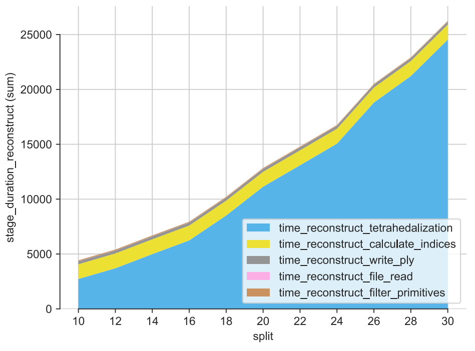
Figure 11: Storage efficiency for each *split* value.



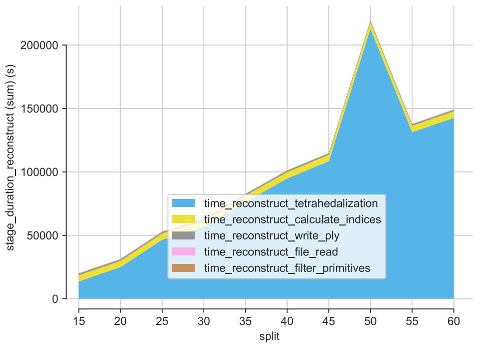
(a) Birmingham



(b) Cambridge

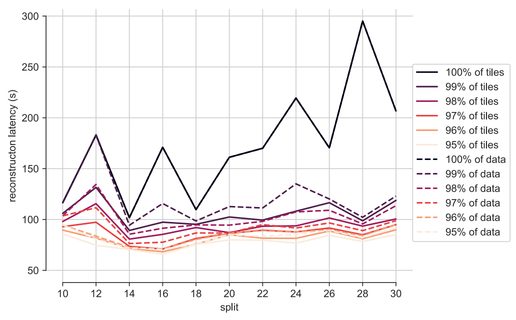


(c) Birmingham

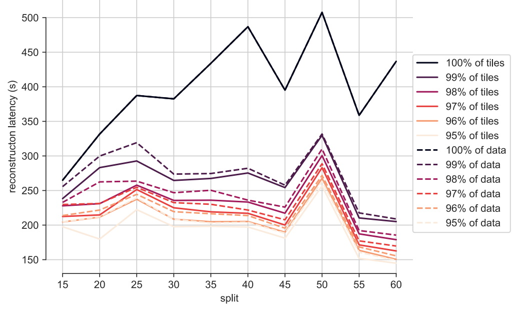


(d) Cambridge

Figure 12: Timings for the surface reconstruction summed over all files per *split* value. *Top:* individual algorithmic components; *Bottom:* Stacked components showing the overall reconstruction time for all files per *split* value.



(a) Birmingham



(b) Cambridge

Figure 13: Theoretical latencies for the reconstruction stage assuming full parallelization for each *split* value. The graphs show the minimum latencies with which a given percentage of tiles (solid lines) or percentage of overall data quantity (dashed lines) could be processed.

is time-consuming and only few Tile files could be investigated, no statement about an actual correlation can be made. Yet, it might be interesting to study the influence of the Tile files structure, for it is partially sorted, on the multithreaded CGAL Delaunay triangulation. Randomizing the input might yield improved runtime performance.

6 Conclusion and Future Work

This report reinforced the results of Komon [2022], showing an overall improvement when parallelizing the execution of BALLFILTER. Additionally, the practical scaling behaviour of the CGAL implementation suggests best real cost and electrical power efficiency for low *split* values that maximize Tile file size while still guaranteeing a distributed version of BALLFILTER as well as SPLIT to not run out of memory. Overly high subdivisions for which the average number of points per Tile file approaches the 200M Points mark, lose efficiency in the CGAL implementation for computing the Delaunay complex. Further work might be able to show a significant speedup in practice by

- either running as many single-threaded versions of BALLFILTER as hardware threads are available per node in parallel for multiple small tiles,
- or implementing a more efficient Delaunay tetrahedralization algorithm, following a divide-and-conquer paradigm which will not suffer from lock contention and may be implemented on the GPU, such as e. g. [Fuetterling et al. 2014].

Additional bottlenecks of the Splitter should be addressed, such as the unconditional serialization of multi-threaded read request on the same input file. As asynchronous IO is not covered by the C++ Standard Library, a cross-platform library, such as asio [4], may be used. However, directly switching to DirectStorage might be promising as it is as well a natural extension to using CUDA in the first processing step.

Lastly, there is an unexplained peak for $split = 50$ in the reconstruction time for the Cambridge example, see Fig. 5b. This was the one parameter instance of the parameter sweep that had to be repeated due to a timeout error caused by one tile crossing the 500s mark in the reconstruction stage, see Fig. 14.

[4]The implementation for the Splitter already includes asio as part of Boost.

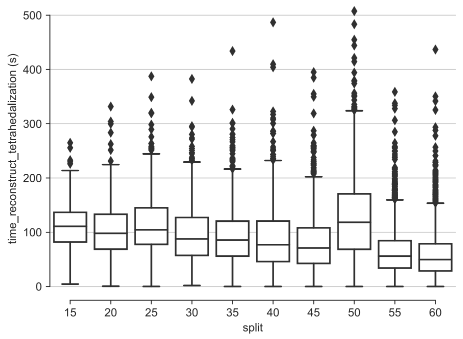


Figure 14: Reconstruction times for the Cambridge example per each *split* value.

It was rerun without a timeout on the reconstruction stage and subsequently resulted in this peak. Further investigation might be required to explain this anomaly and may lead to further research, investigating the effect of actual sample distributions occurring in the reconstruction of planetary surfaces.

References

CARAFFA, L., MARCHAND, Y., BREDIF, M., AND VALLET, B. 2021. Efficiently Distributed Watertight Surface Reconstruction. *International Conference on 3D Vision*, 1432–1441.

DEVILLERS, O., HORNUS, S., AND JAMIN, C. 2022. dD triangulations. In *CGAL User and Reference Manual*, 5.5.1 ed. CGAL Editorial Board. `https://doc.cgal.org/5.5.1/Triangulation/index.html#TriangulationSecPerf`.

FUETTERLING, V., LOJEWSKI, C., AND PFREUNDT, F. J. 2014. High-Performance Delaunay Triangulation for Many-Core Computers. *High-Performance Graphics*, January, 97–104.

HACKEL, T., SAVINOV, N., LADICKY, L., WEGNER, J. D., SCHINDLER, K., AND POLLEFEYS, M. 2017. Semantic3d.net: A new large-scale point cloud classification benchmark. In *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. IV-1-W1, 91–98.

HOETZLEIN, R. 2014. Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids. *GPU Technology Conference*.

HU, Q., YANG, B., KHALID, S., XIAO, W., TRIGONI, N., AND MARKHAM, A. 2022. Sensaturban: Learning semantics from urban-scale photogrammetric point clouds. *International Journal of Computer Vision 130*, 2, 316–343.

KOMON, P. M. 2022. *Distributed Surface Reconstruction*. Bachelor's thesis, TU Wien.

OHRHALLINGER, S., 2022. Personal communication. TU Wien.

PETERKA, T., MOROZOV, D., AND PHILLIPS, C. 2015. High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation. *International Conference for High Performance Computing, Networking, Storage and Analysis*, January, 997–1007.

TANEMURA, M. 2003. Statistical distributions of Poisson Voronoi cells in two and three dimensions. *Forma 18*, 221–247.