Distributed Surface Reconstruction

Patrick Komon, BSc[*]
Supervised by: Projektass.in Diana Marin, BSc MEng [†]

Institute of Computer Graphics
Vienna University of Technology
Vienna / Austria

Abstract

As 3D scanning technology is advancing, both quality and quantity of available point cloud data is increasing. Many applications require the reconstruction of the surface of a scanned object as a 3D model. As scans become exceedingly detailed, point clouds become larger and surface reconstruction more computationally challenging. A fast and scalable solution for the reconstruction problem is needed. We constructed and implemented a scalable distributed surface reconstruction algorithm called DISTRIBUTEDBALLFILTER based on the recently developed BALLFILTER algorithm. We executed our implementation on the VSC3+ high performance computing cluster and empirically analysed its speedup as well as its parallel scaling behaviour. In our tests for DISTRIBUTEDBALLFILTER we achieved running times around five times faster than with BALLFILTER.

Keywords: Surface reconstruction, Point cloud, Distributed memory, Parallel computing

1 Introduction

In recent years, 3D scanning technology has become easier to obtain and use. As a consequence, more scanned data is made available as part of private or public projects. One such project is “Wien gibt Raum” [19] in which more than 100 terabytes of point cloud data was collected. It is often favourable or necessary to work with 3D models consisting of points, edges and faces that represent real-world objects rather than point clouds. Thus the challenge is to create a model from the point cloud that most accurately represents the scanned object. In computer graphics, this problem is known as *surface reconstruction*. Introducing noise and outliers, as they are present in most real-life scans, makes solving this problem even more challenging. Large point clouds require fast and scalable surface reconstruction algorithms so that a 3D model can be calculated in reasonable time.

[*]e11808210@student.tuwien.ac.at
[†]dmarin@cg.tuwien.ac.at

1.1 Related work

There are several approaches for surface reconstruction. You et al. [21] categorize the existing surface reconstruction algorithms based on their methodology into *interpolation*, *approximation* and *learning-based* approaches. While *soft-computing* approaches are mentioned separately, they will not be further explained as they are not relevant to this work.

Interpolation approaches (also called *combinatorial* approaches) try to reconstruct the surface that (exactly) goes through all sampled points. Usually they use the Delaunay complex or the Voronoi diagram of the point cloud. The CRUST algorithm introduced by Amenta et al. [1, 2] is a well known Delaunay-based combinatorial approach. In their work, they also introduced the notion of ε-sampling, relating surface features with the sampling density. It inspired multiple variants, each improving reconstruction quality for specific cases, for example POWERCRUST, which improves in noisy and under-sampled regions [3].

Approximation approaches try to find the surface by finding a function that best agrees with all sampled points, similar to curve fitting. Widely used in practice, SCREENEDPOISSON surface reconstruction [7] applies Poisson‘s equation to solve the reconstruction problem. However, it requires knowledge of the surface normal for each sampled point.

Learning-based approaches facilitate some form of machine learning. One recent example is POINTS2SURF [6], which managed to reduce the reconstruction error by 30% compared to SCREENEDPOISSON.

1.2 BALLFILTER algorithm and limitations

Recently, Ohrhallinger [12] developed a new Delaunay-based reconstruction algorithm called BALLFILTER. As the corresponding paper is not published yet, we cannot explain its inner workings. In general, it calculates the Delaunay complex and filters its triangles. By design, BALLFILTER is a serial algorithm and therefore is limited by the physical capabilities of a single machine. This effectively limits the size of data sets it can process in reasonable time.

1.3 Distributed approach

In this paper, we address these limitations from a parallel computing perspective by introducing a distributed-memory parallel version of BALLFILTER. DISTRIBUTEDBALLFILTER first subdivides the input into a three-dimensional grid with overlapping cells. Then BALLFILTER is run on each grid cell separately. Finally, all results are collected and merged together to create the final result.

This removes the memory restriction imposed by using a single machine. Moreover, it enables scaling the input size while still maintaining acceptable run times. We aim at utilizing parallel infrastructure, as is present in state-of-the-art high-performance computing (HPC) clusters, to execute BALLFILTER on much larger data sets than previously possible. We evaluate its performance and scaling behaviour from a parallel computing perspective and compare it against the original algorithm. Specifically, DISTRIBUTEDBALLFILTER will has been tested and run on the VSC-3+ cluster [5].

2 Background

In this chapter, we will be revisiting some theoretical basics. First, we will cover the Delaunay complex. Then, we will briefly mention important aspects of the BALLFILTER algorithm. Lastly, we will go over typical assumptions of the distributed-memory parallel computing model.

2.1 Delaunay complex

Many surface reconstruction algorithms are based on the Delaunay complex (commonly referred to as Delaunay triangulation). In particular, they operate by examining the tetrahedrons obtained from calculating the Delaunay complex for the input point cloud. The challenge then becomes finding the subset of tetrahedrons that most closely reconstructs the original object.

The Delaunay complex of a three-dimensional point set S consists of (non-overlapping) tetrahedrons as well as all of their vertices, edges, triangles. These tetrahedrons must be constructed from points of S and cover its entire convex hull. Furthermore, they must fulfil the *empty-sphere property*, that is, the circumsphere of each tetrahedron must not contain any (other) vertices. For every set of points the Delaunay complex can be calculated in $O(n \log n)$ time [9].

2.2 BALLFILTER reconstruction method

Because the paper presenting the BALLFILTER algorithm is not published yet, we cannot provide details about its function. The most important steps are calculating the Delaunay complex and filtering its triangles. It can reconstruct open surfaces and runs in $O(n \log n)$ time for n points.

2.3 Distributed-memory parallel computing

In the distributed-memory parallel computing model there is a number of p processes that are connected via a communication network. Each process can only access its own local memory. Processes can only interact with each other by communicating over the communication network. There are several paradigms, standards and frameworks providing means for communication. For high-performance computing, the most used is the Message Passing Interface (*MPI*) [10]. It utilizes the paradigm of message passing. Processes communicate with each other by explicitly sending and receiving messages [17]. Costs for such operations are determined by the concrete topology of the communication network.

We will be using the notion of speedup. The *(absolute) speedup* S_{abs} of a parallel algorithm is its improvement over a baseline sequential version, in our case BALLFILTER [15]. The *relative speedup* S_{rel} measures the improvement of multiple processes over using a single one.

3 Method

In the following, we will explain in detail how the parallelization of the algorithm works. First, we will explore how the input is subdivided. Next, we will discuss how those parts are distributed among multiple processes. Then, we will examine the parallel reconstruction step and derive its parallel runtime complexity. Lastly, we will discuss how the resulting partial meshes are merged back together.

3.1 Splitting into overlapping tiles

The approach for input subdivision was developed by Brunner [4] and is thoroughly explained in their dedicated work. We will only explain the most important aspects.

Splitting the input into independent parts is necessary to enable parallelism. We will be splitting the point set along a regular 3D grid into cells. The grid is axis-aligned, covers the entire point cloud and the number of grid cells along each axis is given as input parameters x, y and z, creating a total of $s = x \times y \times z$ cells. The grid cell dimensions are calculated based on the minimal and maximal coordinates present in the point cloud.

However, the union of the Delaunay triangulations of all grid cells is not equal to the Delaunay triangulations of the entire point clouds, as triangles constructed from vertices in different, neighbouring cells will be missing. Visually, this leads to cuts along the 3D grid in the resulting model. Put differently, we have to make sure not to miss triangles, which have vertices in different grid cells.

We solve this problem by considering points in an area around every grid cell in addition to the points within it. This area is given by the so-called *padding*. We call the set of all points within a grid cell and its surrounding padding

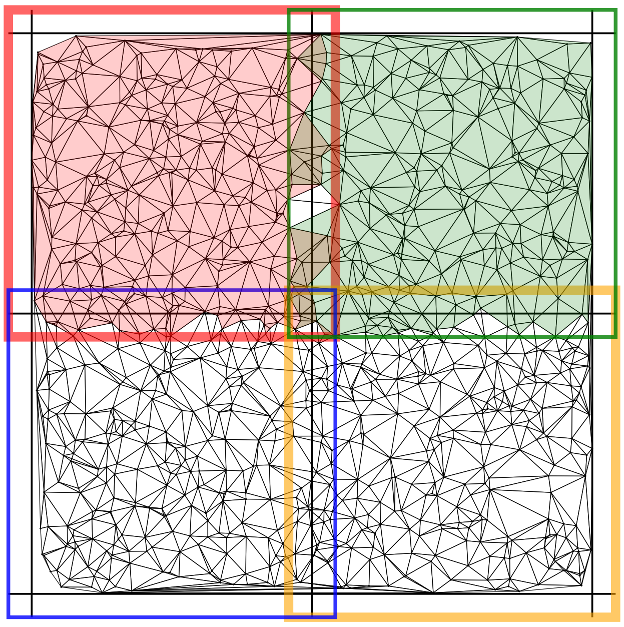
Figure 1: Delaunay-triangulated 2D point cloud, subdivided into a regular grid. Tiles are highlighted by coloured borders with varied thickness for readability.

a *tile*. The set of tiles is not a partition of the point cloud but a set of overlapping subsets. The number of tiles is equal to the number of grid cells. The padding is added to the grid cell dimensions, extending the cell along each axis in positive and negative directions.

The input subdivision is illustrated in Figure 1. Tiles are regions with coloured borders. The vertices of red-coloured triangles belong to the upper left tile. The vertices of green-coloured triangles belong to the upper right tile. In the overlap region, there are triangles with all vertices belonging to both tiles. These will be part of the Delaunay triangulations in each tile separately and thus be considered for reconstruction by BALLFILTER. While this cannot lead to holes in the resulting mesh, it may lead to redundant triangles. This may be fixed as part of a post-processing step, see 3.4.

Two vertices of a triangle can only be in two different tiles exclusively, if the edge between these vertices is longer than double the padding. This case is illustrated in Figure 1, where between green and red tile, there are two white triangles. These will be missing in the separate Delaunay triangulations of each tile. However, by choosing a padding relative to the parameters used for BALLFILTER, we ensure that all triangles that BALLFILTER considers are also present in the Delaunay triangulation of at least one tile.

Checking a single point for tile membership requires checking its coordinates against the grid as well as the padding. This can be done in $O(1)$, therefore calculating the memberships of all points can be done in $O(n)$.

The overlap between tiles leads to the duplication of some points of the input set. In particular, the maximal number of tiles that may be overlapping each other at any given location dictates the (maximal) factor of duplication. Choosing a padding smaller than half the tile dimensions in each direction would imply a maximum of four overlapping tiles in the 2D case (see Figure 1) and eight overlapping tiles in the 3D case.

Therefore, for DISTRIBUTEDBALLFILTER, any point of the input set may be part of up to eight tiles and thus be duplicated up to eight times. The total number of points n' after splitting is $n' \leq 8n$ and is in $O(n)$. From here on we assume $n' = cn$ with some constant factor $c < 8$.

3.2 Work distribution

The next step is to assign the tiles to a fixed number of p *processes* (also called *machines* or *nodes*). We assume all processes are capable of performing the same amount of work in the same time (*assumption of identical machines*). The running time of the entire program is determined by the running time of the slowest process. There is no assumption about the distribution of the points. There may be significant differences in the number of points between tiles. We call an execution of BALLFILTER for a single tile a *job*. The challenge is to assign jobs to processes in such a way that the total running time is minimized. This problem is well known as *load balancing* [8].

There are various ways of approaching this problem and generating optimal solutions can be computationally complex. Instead of trying to find optimal solutions, we will focus on efficiently finding a solution, that is reasonably close to the optimum, thus *approximating* it.

Formally, we want to assign jobs in such a way, that the maximum running time within all processes is minimized. The running time for processing a tile is dependent on the number of points it contains. Thus, the goal is to minimize the total number of points (that is, the sum of the tile sizes) any single process gets assigned.

To minimize this sum, we will be using *list-scheduling* with the *longest-processing-time-first (LPT) rule*. It is a simple and fast approximation algorithm for the scheduling problem, that guarantees a $4/3$-approximation [20], i.e. the calculated schedule is slower than the optimal by factor $4/3$ at the most. In each iteration, a job is assigned to the process with the least number of points to process yet. The jobs are assigned in order from highest number of points to lowest. Sorting the jobs can be done in $O(s \log s)$, with s being the number of tiles. The process assignment is done in $O(s \log p)$. As it only makes sense for p processes to process at least s tiles, it holds that $p \leq s$. Therefore, calculating the assignment is in $O(s \log s)$.

3.3 Processing tiles

Each process executes BALLFILTER on every tile it was assigned. The result of BALLFILTER is a set of triangles, represented by the indices of the vertices in the original point cloud. The run time of a single process is the sum of the run times of BALLFILTER for the set of all tiles *LT* assigned to this process, thus

$$O\left(\sum_{t_i \in LT} |t_i| \log |t_i|\right).$$

The term with the largest tile size dominates this sum, so the largest tile of each process dictates its asymptotic run time. The overall run time is determined by the slowest process, which is the one with the largest tile. Therefore, the overall bound can be expressed as

$$O(|t_i| \log |t_i|) \quad \text{with} \quad \forall t_j \in T : |t_i| \geq |t_j|.$$

Hence, the running time of DISTRIBUTEDBALLFILTER is dependent on the distribution of points within the tiles, which itself depends on the distribution in the point cloud, the chosen grid and the padding.

In the best case, every process is assigned exactly an equal part of all points (after duplication), i.e. a number of $\frac{cn}{p}$ points. Therefore, the largest tile t_i can contain at most this many points, formally $|t_i| \leq \frac{cn}{p}$. Since *LPT list-scheduling* guarantees a 4/3-approximation and BALLFILTER can be done in $O(n \log n)$, this implies a bound for the parallel running time in

$$\begin{aligned} O\left(\frac{4}{3}|t_i| \log |t_i|\right) &= O\left(\frac{4}{3}\frac{cn}{p} \log \frac{cn}{p}\right) \\ &= O\left(\frac{4c}{3}\frac{n}{p}(\log c + \log n - \log p)\right) \\ &= O\left(\frac{n \log n}{p}\right). \end{aligned}$$

In the worst case, almost all points are assigned to a single process. This may happen when the tile sizes are extremely unbalanced, e.g. if two tiles with sizes $cn - 1$ and 1 should be assigned to two processes. As the largest tile may be of size cn, the run time of the slowest process and thus the parallel running time is in

$$\begin{aligned} O(|t_i| \log |t_i|) &= O(cn \log cn) \\ &= O(n(\log c + \log n)) \\ &= O(n \log n) \end{aligned}$$

which is (asymptotically) equal to running BALLFILTER on the original point cloud. Due to duplicated points and communication overhead, the actual run times of DISTRIBUTEDBALLFILTER are expected to be higher than BALLFILTER in this case. However, such cases may be mitigated most of the time by carefully choosing the parameters for splitting.

3.4 Merging results and output

Upon finishing executing BALLFILTER on all assigned tiles, each process sends the resulting triangles to a single process, which joins all received sets together and outputs the final model. The padding causes all triangles that are considered by BALLFILTER to be entirely contained within at least one tile. Therefore, there is no need to explicitly connect the meshes of neighbouring tiles, the result is simply the union of all triangle sets.

However, as mentioned in 3.1, reconstructed triangles may be part of more than one tile. In this case, merging the results generates redundant geometry in the resulting mesh. Removal of those triangles is possible as a post-processing step. We will not take redundant triangle removal into account for running time considerations and because it does not interfere with the visual quality of the reconstructed model.

3.5 Summary

To summarize, DISTRIBUTEDBALLFILTER processes a set of *n* points using the following steps. Note that the point duplication caused by tile overlap is assumed to be a constant factor as discussed in 3.1.

1. Split the input point cloud into tiles in $O(n)$ (on a single node).
2. Calculate schedule in $O(s \log s)$ (on a single node).
3. Perform BALLFILTER in parallel worst case $O(n \log n)$, best case $O(\frac{n \log n}{p})$ (on *p* nodes).
4. Merge results from nodes in $O(n)$ (on a single node).

The overall asymptotic run time complexity is $O(n \log n)$ in the worst and $O(n + \frac{n \log n}{p})$ in the best case.

4 Implementation

In this chapter, we will briefly mention the technology used as well as discuss the implementation structure.

4.1 Technology

For the implementation, C++ has been used as it offers both, performance and high-level abstractions. As input splitting involves a large number of simple operations, has been use *CUDA* to utilize the large scale shared-memory parallelism possible on GPUs [11]. For distribution of the tiles to the processes, we use the distributed file system of the VSC-3+ cluster, *BeeGFS*. For the reconstruction step performed on each tile, the original implementation of BALLFILTER has been. For the merging step, each process' outputs are sent back to a single node using *Open-MPI* [14].

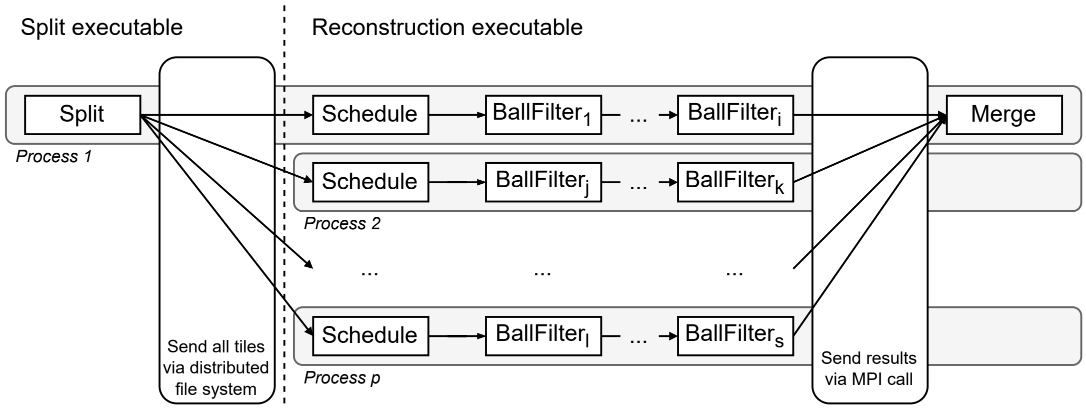

Figure 2: Implementation of DISTRIBUTEDBALLFILTER

4.2 Structure

Splitting the input point cloud and performing surface reconstruction are implemented in separate executables. The output of the splitting step, which is also the input of the reconstruction step, consists of a set of files, each representing a tile (*tile file*). This promotes decoupling of input splitting and reconstruction and allows exchanging implementations as well as introducing additional processing steps. On the VSC3+, tile files are written to the distributed file system, making all tiles available to all nodes after the splitting step.

Then the reconstruction executable is started on all nodes in parallel. Each process calculates the job schedule (tile file assignment) independently. Compared to calculating it on a single node and communicating it to all others, it is still faster because no communication is required. Then each process runs BALLFILTER on each of its assigned tiles. When finished, all nodes send their results to the node with the least load. This node then merges the results and outputs the model. The entire process is visualized in Figure 2. Notice, the structure differs slightly from the one proposed in 3.5. We deliberately chose this approach due to the availability of a distributed file system.

5 Results and Evaluation

In this chapter, we will discuss the results of our implementation on specific data sets and parameter combinations. First, we will give an overview of the hardware environment and explain the data sets and parameter combinations selected for testing. Moreover, we will visualize the running times and reason about the effect of scaling the number of processes p or the input size n. Finally, we compare these times to the original implementation of BALLFILTER.

5.1 Hardware environment and Datasets

The algorithm has been run on the VSC-3+ cluster. It consists of various types of nodes with different hardware specifications [5]. We used two different node types. For the splitting step, we used a single node with a NVIDIA Pascal GeForce GTX 1080 GPU. For the reconstruction step, we used a number of identical nodes, containing two Intel Xeon E5-2660v2 2.2GHz processors with 10 cores each (so 20 cores in total) and 64GiB of RAM. The reconstruction step has been run multiple times while varying the number of nodes used in order to analyse the scaling behaviour.

As input point clouds for testing, two data sets have been selected, both obtained via photogrammetry provided by *Pix4D* [13]. They were chosen based on their large size and real world relevance. The point clouds were truncated to create inputs of various sizes n. In order to observe the running times of DISTRIBUTEDBALLFILTER on scaled input, n is varied while keeping the number of nodes p fixed.

5.2 Parameters

The parameters for BALLFILTER only influence the quality of the resulting 3D model and generally do not significantly impact performance. For this reason, we keep them fixed for all runs at the values recommended in [12].

Splitting the input requires three parameters, x, y and z, denoting the number of tiles along each axis, respectively. Correctly choosing these parameters is vital for work distribution and in turn has a great impact on the overall run time. Although not strictly needed, information about the distribution of points in the input point cloud is helpful for picking concrete values. During testing, these values will be picked based on the number of nodes to employ.

The number of processes p is closely related to the number of tiles. Generating less tiles than there are nodes allocated would leave some nodes idle, waiting for the others to finish, so $p \leq s$. Having more tiles allows for better work distribution and may enable faster overall running times. This also depends on how balanced the tiles are to begin with. However, using more tiles than nodes leads to overhead by point duplication. At some point, the benefit of better work distribution is exceeded by the additional work of processing duplicated points. In our tests, we will use $s = p$ and $s = 2p$.

5.3 Measured running times

For a closer examination, we chose the `eclepens` data set. The exact running times for various runs with different parameter combinations are listed in Table 1 as well as the absolute and relative speedup. T_{split} and $T_{reconstruct}$ are the times for running the split and reconstruct executables respectively as shown in Figure 2.

The running time of the split step are consistently very low. In comparison, the running times of the reconstruction step are dominating the total running time. As this paper's main focus is the reconstruction step, we will neglect T_{split} and refrain from analysing its impact on the total running time.

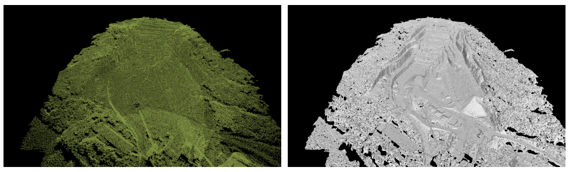

Figure 3: Input point cloud (left) and reconstruction (right) of the `eclepens` (8 million points) data set calculated using DISTRIBUTEDBALLFILTER with 16 processes and 32 tiles in 11.43s.

5.4 Scaling behaviour and speed-up

Figure 4 shows the running times of the original (shared-memory) implementation of BALLFILTER and our implementation of DISTRIBUTEDBALLFILTER ran with 16 nodes and 32 tiles for different versions of `eclepens`. The data set has been truncated to specific sizes (from $2^0 = 1$ to $2^5 = 32$ million points) to simulate growing input size.

BALLFILTER has an asymptotic running time bound of $O(n \log n)$. The asymptotic running time bound of DISTRIBUTEDBALLFILTER is $O(n \log n)$ for the worst and $O(n + \frac{n \log n}{p})$ for the best case. The distributed version is in the worst case (asymptotically) as fast as the original algorithm and faster in the best case. The factor by which it is faster is called absolute speedup S_{abs}. In reality, the absolute speedup will always be somewhere between 1 and p, depending on the balance of the tile set which in turn is based upon the distribution of points in the input point cloud. If p is considered constant, the asymptotic run time complexity for DISTRIBUTEDBALLFILTER is $O(n \log n)$ for all cases and matches the one of BALLFILTER. Therefore, the only difference between both running times lies

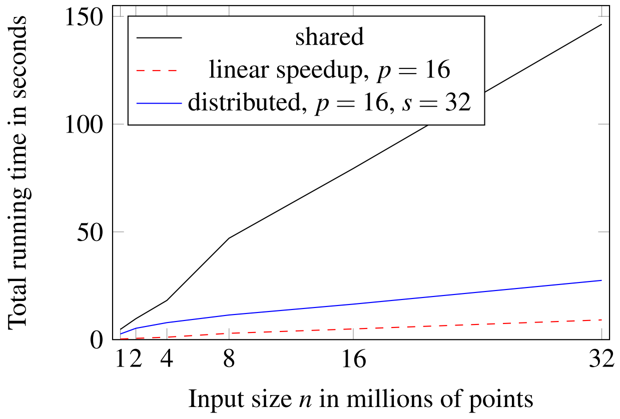

Figure 4: Running times of the original BALLFILTER vs DISTRIBUTEDBALLFILTER with increasing input size for `eclepens`

in the coefficient (which is ignored by asymptotic complexities).

In our example, the distributed version with 16 nodes was faster than the shared version for all input sets. Both curves look similar, as their asymptotic running times suggests. The absolute speedup increased with the size of the input up to a factor of 5.32. The best possible absolute speedup, i.e. linear speedup, would be 16, which would lead to running times that are $\frac{1}{16}$th of the original shared running times.

Now we will investigate how the running times change when the number of processes p is scaled up. We executed DISTRIBUTEDBALLFILTER on the `eclepens` data sets with 16 and 32 million points multiple times, each time doubling the number of nodes used. The number of tiles was set to twice the number of nodes ($s = 2p$), so that in cases of imbalance, the scheduling algorithm can balance out the work between the processes. The running times are listed in Table 1 and visualized in Figure 5.

For one node, DISTRIBUTEDBALLFILTER was slower than the original implementation. That is to be expected because of the overhead required by the distributed implementation. Using two or more nodes, however, significantly decreased the running times compared to the original implementation.

As the absolute and relative speedup approach a value of around 6, the running times stay the same even when the number of nodes is increased. For larger numbers of tiles, the additional work caused by duplicate points eventually cancels out the performance gained by parallel execution.

To summarize, DISTRIBUTEDBALLFILTER performed well on scaling up input size n as well as scaling the number of nodes p. Specifically, on the original `eclepens` data set, when using 16 nodes, it was shown that DISTRIBUTEDBALLFILTER outperformed BALLFILTER by a factor of 5.66. We also observed that having a higher number of tiles can, and in many cases will, increase the overall running time because it enables better work distribution,

	p	$s = x \times y \times z$	T_{split}	$T_{reconstruct}$	T_{total}	S_{abs}	S_{rel}
`eclepens` 16 million points	1	$2 = 2 \times 1 \times 1$	1.35175	88.8745	90.22625	0.88	1.00
	2	$4 = 2 \times 2 \times 1$	1.59808	53.7387	55.33678	1.43	1.63
	4	$8 = 2 \times 2 \times 2$	1.34450	31.4786	32.82310	2.42	2.75
	8	$16 = 4 \times 2 \times 2$	1.25440	22.4666	23.72100	3.35	3.80
	16	$32 = 4 \times 4 \times 2$	1.56540	14.8993	16.46470	4.82	5.48
	32	$64 = 4 \times 4 \times 4$	1.66721	11.8065	13.47371	5.89	6.70
`eclepens` 32 million points	1	$2 = 2 \times 1 \times 1$	1.56998	168.3390	169.90898	0.86	1.00
	2	$4 = 2 \times 2 \times 1$	1.72504	96.3045	98.02954	1.49	1.73
	4	$8 = 2 \times 2 \times 2$	1.62856	61.7569	63.38546	2.31	2.68
	8	$16 = 4 \times 2 \times 2$	1.75069	36.3197	38.07039	3.84	4.46
	16	$32 = 4 \times 4 \times 2$	1.84338	25.6505	27.49388	5.32	6.18
	32	$64 = 4 \times 4 \times 4$	2.24963	23.7843	26.03393	5.62	6.53

Table 1: Parameter combinations and running times for `eclepens`

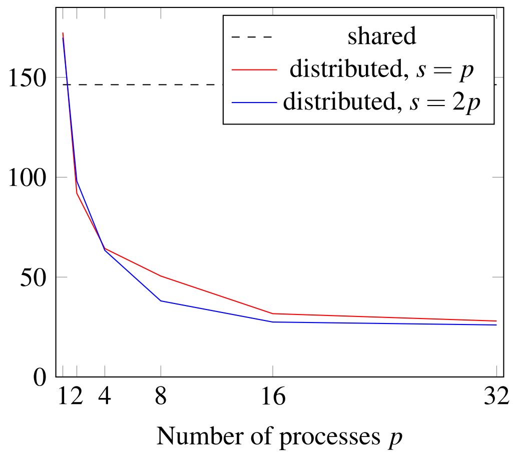

Figure 5: Absolute running times of the original BALLFILTER vs DISTRIBUTEDBALLFILTER with one tile per process and two tiles per process respectively, run on `eclepens` with 32 million points.

despite requiring more duplicated points. Figure 4 shows that DISTRIBUTEDBALLFILTER is a large improvement towards linear speedup.

6 Conclusion and Future Work

Finally, in this chapter we will briefly summarize this paper's results and give a short outlook on future work that could be done on DISTRIBUTEDBALLFILTER.

6.1 Summary

In this paper, we presented a distributed-memory parallel algorithm for 3D surface reconstruction. It works by splitting the input into overlapping chunks, reconstructing a mesh from each chunk in parallel using BALLFILTER and merging the chunk results back together.

We have shown the asymptotic run time complexity to be $O(n \log n)$ in the worst and $O(n + \frac{n \log n}{p})$ in the best case, depending on the distribution of points within the input point cloud. We implemented the algorithm in C++ and tested it on the VSC3+-cluster. In our test runs, we observed that DISTRIBUTEDBALLFILTER improves the running times considerably compared to the original BALLFILTER.

6.2 Future Work

While we analysed mainly from an empirical perspective, DISTRIBUTEDBALLFILTER can be analysed in a more formal setting. Speedup and scaling properties can be argued and proven formally in order to evaluate our approach in a more theoretical sense. Also, the best- and worst-case asymptotic running time complexities could be expressed in more concrete ways, as coefficients oftentimes do matter in the practical comparison of algorithms. Formally taking into account the distribution or balance of the points within the input point cloud may yield further insights into the properties of DISTRIBUTEDBALLFILTER algorithm.

References

[1] Nina Amenta, Marshall Bern, and David Eppstein. The crust and the β-skeleton: Combinatorial curve reconstruction. *Graphical Models and Image Processing*, 60(2):125–135, 1998.

[2] Nina Amenta, Sunghee Choi, Tamal Dey, and Naveen Leekha. A simple algorithm for homeomorphic surface reconstruction. *International Journal of Computational Geometry & Applications*, 12, September 2000.

[3] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, SMA '01, page 249–266, New York, NY, USA, 2001. Association for Computing Machinery.

[4] Lukas Brunner. Technical report - work in progress. TU Wien, 2022.

[5] VSC Vienna Scientific Cluster. Vienna Scienfic Cluster (VSC) - website. `https://vsc.ac.at/systems/vsc-3/`, 2022. [Online; accessed 13-September-2022].

[6] Philipp Erler, Paul Guerrero, Stefan Ohrhallinger, Niloy J. Mitra, and Michael Wimmer. Points2Surf: Learning Implicit Surfaces from Point Clouds. In *Computer Vision – ECCV 2020*, pages 108–124. Springer International Publishing, 2020.

[7] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Trans. Graph.*, 32(3), July 2013.

[8] Jon Kleinberg and Éva Tardos. *Algorithm design*, chapter 11.1 Greedy Algorithms and Bounds on the Optimum: A Load Balancing Problem. Pearson Addison Wesley, Boston, Mass. [u.a.], internat. ed.. edition, 2006.

[9] Geoff Leach. Improving worst-case optimal delaunay triangulation algorithms. In *In 4th Canadian Conference on Computational Geometry*, page 15, 1992.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[11] NVIDIA Corporation. CUDA Toolkit - project website. `https://developer.nvidia.com/cuda-toolkit`, 2022. [Online; accessed 13-September-2022].

[12] Stefan Ohrhallinger. Personal communication. TU Wien, 2022.

[13] Pix4D SA. Pix4D - website. `https://www.pix4d.com/`, 2022. [Online; accessed 13-September-2022].

[14] The Open MPI Project. OpenMPI - project website. `https://www.open-mpi.org/`, 2022. [Online; accessed 13-September-2022].

[15] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*, chapter 4.2.1 Speedup and Efficiency. Springer Berlin Heidelberg, 2010.

[16] SchedMD LLC. Slurm Workload Manager - project website. `https://slurm.schedmd.com/overview.html`, 2021. [Online; accessed 13-September-2022].

[17] B. Schmidt, J. Gonzalez-Martinez, C. Hundt, and M. Schlarb. *Parallel Programming: Concepts and Practice*, chapter 9.1 Message Passing Interface. Elsevier Science, 2017.

[18] The CGAL Project. CGAL - project website. `https://www.cgal.org/`, 2022. [Online; accessed 13-September-2022].

[19] Stadt Wien. Wien Gibt Raum - project website. `https://digitales.wien.gv.at/projekt/wiengibtraum/`, 2022. [Online; accessed 13-September-2022].

[20] Xin Xiao. A direct proof of the 4/3 bound of LPT scheduling rule. In *Proceedings of the 2017 5th International Conference on Frontiers of Manufacturing Science and Measuring Technology (FMSMT 2017)*, pages 486–489. Atlantis Press, 2017/04.

[21] Cheng Chun You, Seng Poh Lim, Seng Chee Lim, Joi San Tan, Chen Kang Lee, and Yen Min Jasmina Khaw. A survey on surface reconstruction techniques for structured and unstructured data. In *2020 IEEE Conference on Open Systems (ICOS)*, pages 37–42. IEEE, 2020.