TU WIEN Informatics

Distributed Surface Reconstruction

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Patrick Michael Komon
Matrikelnummer 11808210

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass.(FWF) Dr. Stefan Ohrhallinger
Projektass.in Diana Marin, BSc MEng.

Wien, 15. September 2022

Patrick Michael Komon   Michael Wimmer

Informatics

Distributed Surface Reconstruction

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Patrick Michael Komon
Registration Number 11808210

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass.(FWF) Dr. Stefan Ohrhallinger
Projektass.in Diana Marin, BSc MEng.

Vienna, 15th September, 2022

Patrick Michael Komon   Michael Wimmer

Erklärung zur Verfassung der Arbeit

Patrick Michael Komon

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. September 2022

Patrick Michael Komon

Danksagung

Hiermit möchte ich allen Menschen danken, die mich beim Schreiben dieser Arbeit und damit beim Erreichen meines Abschlusses unterstützt haben. Besonders möchte ich meiner wunderbaren Betreuerin, Diana Marin, danken. Sie hat mich ermutigt, an meine Arbeit zu glauben, regelmäßig meinen Fortschritt überprüft, nützliches Feedback gegeben und immer schnell auf Fragen geantwortet. Darüber hinaus wäre diese Arbeit ohne die emotionale und manchmal technische Unterstützung meiner Eltern und meiner Freunde nicht möglich gewesen. Zuletzt möchte ich meine Dankbarkeit für die Möglichkeit ausdrücken, eine verteilte Implementierung für einen State-of-the-art-Algorithmus zu schreiben und diese auf einem echten HPC-Cluster auszuführen.

Acknowledgements

Hereby I want to thank all the people who supported me in writing this thesis and finishing my bachelors degree. Especially I would like to thank my wonderful supervisor, Diana Marin, who encouraged me to believe in my work, regularly checked on my progress and gave me insightful feedback while always responding very quickly. Furthermore, this work would not have been possible without the emotional and sometimes technical support of my parents and friends. Lastly, I am very grateful for the opportunity to write a distributed implementation for a state-of-the-art algorithm and a run it on a real HPC cluster.

Kurzfassung

Während hochqualitative 3D-Scans leichter zugänglich werden, entstehen neue Herausforderungen für die Verarbeitung der erzeugten Daten. 3D-Scanner erzeugen sehr große, unstrukturierte Mengen von 3D-Punkten, sogenannte Punktwolken. Um diese Daten sinnvoll weiterverwenden zu können, ist es notwendig die Oberfläche des gescannten Objekts als 3D-Modell zu rekonstruieren. Dieses Problem heißt 3D-Oberflächenrekonstruktion. Um mit dem Fortschritt der 3D-Scanning-Technologie mitzuhalten, ist es erforderlich, sehr großen Punktwolken in akzeptabler Zeit verarbeiten zu können.

In dieser Bachelorarbeit konstruieren, implementieren und evaluieren wir einen parallelen, verteilten Algorithmus zur Oberflächenrekonstruktion namens DistributedBallFilter, der auf dem kürzlich entwickelten BallFilter-Algorithmus [Ohr22] basiert. Mithilfe eines 3D-Gitters wird die Punktwolke in unabhängige Teile (sogenannte *Tiles*) unterteilt. Zur Sicherstellung der Korrektheit überlappen sich einzelne Tiles an den Rändern. Die Tiles werden anschließend an p Prozesse verteilt. Die Tile-Prozess-Zuweisung wird mithilfe von *Longest-Processing-Time-First List Scheduling* berechnet. Alle Prozesse rekonstruieren gleichzeitig die Oberflächen in den ihnen zugewiesenen Tiles. Dann werden die Einzelergebnisse in einem einzigen 3D-Modell zusammengeführt, welches die rekonstruierte Oberfläche der gesamten Punktwolke enthält. Die asymptotische Laufzeitkomplexität ist in $O(n \log n)$ im schlechtesten Fall und in $O(n + \frac{n \log n}{p})$ im besten Fall, je nach Verteilung der Punkte in der Punktwolke.

Der Algorithmus wurde in C++ implementiert. Das Zerteilen des Inputs wird mithilfe von *CUDA* auf einer GPU durchgeführt und wird in [Bru22] detailiert behandelt. Für jedes Tile wird eine einzelne Datei erzeugt, die mithilfe eines verteilten Dateisystems an die einzelnen Prozesse kommuniziert wird. *MPI* wird verwendet, um die Resultate aller Prozesse an einen Prozess zu senden, welcher auch für die Zusammenführung der Einzelergebnisse und die Ausgabe des 3D-Modells zuständig ist.

Wir haben unsere Implementation auf dem High-Performance-Computing-Cluster VSC3+ mithilfe einiger Datensätze evaluiert. Dazu zeigen wir einige Visualierungen der Laufzeiten und analysieren das Verhalten der Laufzeit, wenn die Anzahl der Prozesse oder der Input-Größe variiert wird. Wir konnten in unseren Tests eine Verbesserung der Laufzeit von DistributedBallFilter gegenüber jener von BallFilter um einen Faktor von ungefähr fünf, je nach Anzahl Prozesse und Input-Größe beobachten. Der höchste beobachtete Speedup belief sich auf 5.89.

Abstract

As the accessibility of high-quality 3D scans increases, processing the scanned data becomes more challenging. 3D scanners obtain very large, unstructured sets of points, so called point clouds. To be able to use the data in a meaningful way it is necessary to reconstruct the surface of the scanned object from the point cloud, resulting in a 3D model. This is called the problem of 3D surface reconstruction. Processing very large point clouds (in a reasonable time) is necessary in order to keep up with ever increasing scanning technology.

In this thesis, we construct, implement and evaluate a distributed surface reconstruction algorithm called DistributedBallFilter. It is a distributed-memory parallel version of the recently developed BallFilter algorithm [Ohr22]. Firstly, the input point cloud is subdivided into chunks called *tiles* using a 3D grid. To ensure the correctness of the results, tiles are slightly overlapping on their borders. After splitting the input, each tile can be processed independently from each other. The tiles are assigned and distributed to a number of p processes. The assignment of tiles to processes is calculated using *longest-processing-time-first list scheduling.* Then all processes reconstruct the 3D surface of all their assigned tiles in parallel. After all tiles are processed, the result is merged back together into a single 3D model, containing the reconstructed surface of the entire input point cloud. The asymptotic run time complexity is $O(n \log n)$ in the worst case (same as BallFilter) and $O(n + \frac{n \log n}{p})$ in the best case, depending on the distribution of points within the input data.

Furthermore, we implemented the algorithm in C++. The input splitting is run on a GPU using *CUDA* and discussed thoroughly in its dedicated paper [Bru22]. For each tile a single file is output, which is communicated to each process via a distributed file system. The *MPI* standard is used for sending all local results to a single process, which is also responsible for merging and outputting the final 3D model.

Finally we executed the algorithm on the VSC3+ cluster, a high-performance cluster based in Vienna. It was run against several test data sets. We visualize the results and analysed the behavior of the running time when scaling the number of processes as well as the input size. In our tests, DistributedBallFilter managed to be up to around five times faster than BallFilter depending on the number of nodes used and the input size. The largest observed speedup was by a factor of 5.89 compared to BallFilter.

Contents

CHAPTER 1

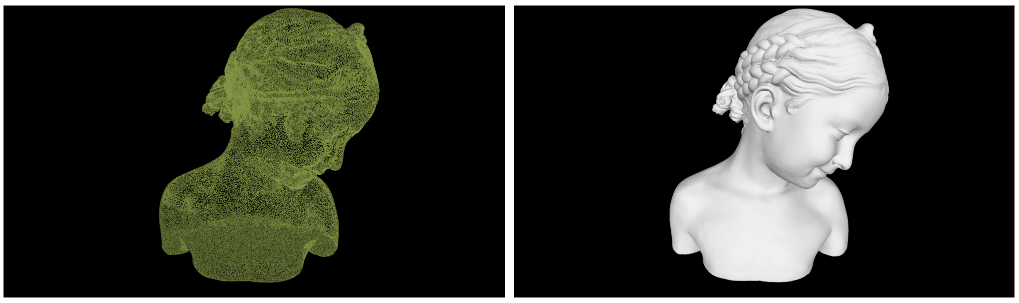
Introduction

In recent years, 3D scanning technology has become considerably easier to obtain and use. As a consequence, more and more scanned data is made available as part of private or public projects. One such project is Wien gibt Raum [Wie22]. Conducted by the government of the City of Vienna, the project aims to increase efficiency and transparency of the administration of public space as well as planning and approval of new projects. For the project, over 100 terabytes of data was collected by vehicles, surveying the streets of Vienna with 360° cameras and laser (LiDAR) scanners. Usually, the result of such surveys is an unstructured set of 3D points, commonly called point cloud. While point clouds can be viewed and manipulated, they only have limited use when it comes to visualization. It is more intuitive to work with 3D models consisting of points, edges and faces that represent real objects, rather than a floating set of points. Models can easily be visualized, manipulated or used with other tools and algorithms. For example, a model created from the Wien gibt Raum data could be used to visualize new construction projects. Therefore, the challenge is to create a model from a point cloud that most accurately represents the scanned object. In computer graphics, this problem is well known as surface reconstruction.

Generally, the problem of 3D surface reconstruction is concerned with approximating the boundary of an object given an unstructured set of points sampled on its surface. Introducing noise and outliers, as they are present in most real-life scans, makes solving this problem even more challenging. Large point clouds such as the one collected by Wien gibt Raum require fast and scalable surface reconstruction algorithms so that a 3D model can be calculated in reasonable time.

1.1 Related work

There are many algorithms for surface reconstruction on three-dimensional point sets.



(a) Input point cloud (b) Reconstructed 3D model

Figure 1.1: The problem of 3D surface reconstruction: From an unordered set of points reconstruct the surface of the object the points were sampled from.

There are several approaches for surface reconstruction. Following You et al. [YLL$^+$20], we categorize the existing surface reconstruction algorithms based on their methodology into *interpolation*, *approximation* and *learning-based* approaches. While *soft-computing* approaches are mentioned separately by You et al., 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. Because DistributedBallFilter is an interpolation approach (and also based on the Delaunay complex), we will explore this category in more detail. Edelsbrunner and Mücke [EM94] utilized the Delaunay complex to extend their previously introduced α-shapes to 3D. Unfortunately, α-shapes assume point clouds to be uniformly sampled, making it difficult to apply to real-world data. Local estimation of the Delaunay complex proved to be hard for non-uniformly sampled data sampling and also introduced the need for complex corrections of inconsistencies caused by bad estimation. Sculpting [Boi84], which greedily removes tetrahedra from the Delaunay complex to obtain the shape, can only reconstruct objects without holes (genus zero) while also suffering from artifacts at regions that are under-sampled. With the Crust algorithm, Amenta et al. [ABE98, ACDL00] moved away from the assumption of uniform sampling and introduced the notion of ϵ-sampling. The ϵ-sampling-condition relates surface features with sampling density, requiring less points in regions with less features. Crust's output contains artifacts that require post-processing and performs poorly in under-sampled regions. It inspired multiple variants, each improving reconstruction quality for specific cases, most notably PowerCrust, which provided improvements in noisy and under-sampled regions [ACK01].

Furthermore, there are some interpolation approaches that do not rely on the Delaunay complex. For example, an approach by Hornung and Kobbelt [HK06] transforms the reconstruction problem into the problem of finding a minimum graph cut. [Ede03] and [GO08] use other types of simplicial complexes, the flow complex and the witness complex respectively.

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 [KH13] reduces the problem of surface reconstruction to a Poisson problem that can be solved as a system of linear equations. It requires knowledge of the surface normal for each sampled point.

Learning-based approaches facilitate some form of machine learning. Many learning-based algorithms estimate surface normals of point clouds, as this is helpful or even necessary for some interpolation/approximation approaches. There are also some algorithms in this category that directly reconstruct surfaces. One recent example is Points2Surf [EGO+20], which managed to reduce the reconstruction error by 30% compared to ScreenedPoisson.

1.2 BallMerge algorithm and limitations

Recently, Ohrhallinger [Ohr22] developed a new Delaunay-based surface reconstruction algorithm called BallMerge. After calculating the Delaunay complex, the resulting tetrahedra are grouped based on the overlap of their circumspheres. The reconstructed object is the largest group of tetrahedra. BallMerge drastically improved upon the required ϵ-sampling condition thus requiring less dense sampling. It is relatively simple and remains robust against noise and outliers while also producing a watertight surface.

A variant of BallMerge called BallFilter [Ohr22] can reconstruct open surfaces and has been shown to work well with real-life scan data. It behaves similar to BallMerge in terms of run times and memory requirements. Empirically, BallMerge had a considerably lower run time than other algorithms such as PowerCrust [ACK01], ScreenedPoisson [KH13] and Points2Surf [EGO+20]. Its memory consumption was below all but ScreenedPoisson.

While BallMerge and BallFilter look very promising, there are still limitations to them. By design, they are serial algorithms and therefore limited by the physical capabilities of a single machine. Despite their good theoretical bounds, practical run times and memory requirements, this effectively limits the size of data sets they 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 the original BallFilter algorithm 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 us to scale 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 (commonly abbreviated to HPC) clusters, to execute BallFilter on much larger data sets than previously possible. While doing so, we evaluate its performance and scaling behavior from a parallel computing point of view and compare it against the original algorithm. Specifically, DistributedBallFilter will be tested and run on the VSC-3+ cluster [Clu22].

CHAPTER 2
Theory

For designing a distributed-memory parallel version of BallFilter, it is necessary to understand its theoretical foundation as well the algorithm and its properties itself. Therefore we will be revisiting the basics of the Delaunay complex, a geometric structure constructed from the input point cloud. Then we will explain in detail how the BallFilter algorithm processes the Delaunay complex of the input point set to reconstruct the surface. As it will be needed for analysing and evaluating our parallel algorithm, we will be recalling fundamental terminology for parallel performance such as the notion of speed up. Further we will explain the aspects of the distributed-memory parallel computing model and communication.

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 (called *subcomplex*) of the *Delaunay complex* that reconstructs the original object closest.

In literature, there are two prevalent ways of defining the Delaunay complex. Firstly, it can be defined as a dual graph to the Voronoi diagram of a point set. Secondly, it can be defined as a simplicial complex, using convex hulls. We will be using the second definition as it more clearly conveys the structure as well as provides us with common terminology used with Delaunay-based surface reconstruction.

The Delaunay complex is a so called *simplicial complex.* A *simplicial complex* is a set of *simplices.* A *n-simplex* is the convex hull of a set of $n + 1$ points. A 0-simplex is called *vertex* and contains only a single point. A 1-simplex is called *edge* and consists of two points as well as all points on a (straight) line segment connecting them. Analogous, a

2-simplex is a *triangle* and a 3-simplex is a *tetrahedron.* A *face* of a simplex is the convex hull of a subset of the points the simplex was created from. For example, a 2-simplex (triangle) is the convex hull of three points. The convex hull of any subset of these three points is a face of that simplex (as well as a simplex itself).

The Delaunay complex of a three-dimensional point set S consists of (non-overlapping) tetrahedrons as well as all of their faces. Thus it contains 0-, 1-, 2- and 3-simplices. These tetrahedrons must be constructed from points taken from S and cover the entire convex hull of S. Furthermore they must fulfill the *empty-sphere property* (also called Delaunay property). It states that the circumsphere of each tetrahedron, that is, the sphere touching all of its vertices, must not contain any (other) vertices. For every set of points the Delaunay complex can be calculated in $O(n \log n)$ time [Lea92].

2.2 BallMerge reconstruction method

The BallMerge reconstruction method starts by calculating the Delaunay complex of the input point cloud and operates on the resulting tetrahedrons. It is based on the observation, that the circumspheres of tetrahedrons (in literature often called *medial balls*) are not overlapping along the boundary between interior and exterior of an object. Object surfaces can therefore be found by dividing the tetrahedrons into groups of interior and exterior. Formally, the overlap between the medial balls of two neighboring tetrahedrons is quantified by the *intersection ratio*, defined as follows:

$$ir = max(\frac{r_0 + r_1 - d}{r_0}, \frac{r_1 + r_0 - d}{r_1})$$

where r_0 and r_1 are the radii of the medial balls of the tetrahedrons and d is the distance between their circumcenters. Also it holds that $0 \leq ir \leq 2$, higher values meaning greater overlap [Ohr22].

The tetrahedrons are now grouped, given a threshold δ. Two neighboring tetrahedrons are considered δ-merged if their intersection ratio is less than δ or if there is another tetrahedron that is δ-merged with both of them. Applying this relation creates a partition of the tetrahedron set. Each set of the partition is called δ-merged component and the reconstructed surface is the boundary of the largest component.

Note that BallMerge always reconstructs closed, watertight surfaces. However, most real-life scans are open surfaces, like for example aerial laser scans. Therefore, Ohrhallinger formulated a variant of BallMerge called BallFilter [Ohr22]. BallFilter directly reconstructs the boundaries between all δ-merged components by including triangles when their neighboring tetrahedrons fulfill $ir > \delta$. Triangles having one or more edges longer than a certain fraction $\frac{1}{tlen}$ of the overall bounding box of the point cloud are excluded as well. For most real-world applications, $tlen = 200$ is recommended.

Both, BallMerge and BallFilter only need a single traversal of the Delaunay complex. As checking the intersection ratio and edge length can be done in $O(1)$ time, the entire traversal takes $O(n)$. The reconstruction is hence dominated by calculating the Delaunay complex, taking $O(n \log n)$.

Our approach involves splitting the input point cloud along a 3D grid. Even for point clouds sampled from closed surfaces, the resulting grid cells will contain samplings of open surface. Because of this observation and the fact, that BallFilter works better with real-life data, we will be using the BallFilter variant exclusively.

2.3 Distributed-memory parallel computing

We will be reasoning about the distributed version of BallFilter from the perspective of parallel computing. Therefore we will revisit some of the terminology and concepts used.

2.3.1 Assumptions and model

In the distributed-memory parallel computing model there are a number of p processes that are connected via a communication network (sometimes called interconnect). 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. The most used and widely by HPC clusters widely supportedstandard is *MPI*, the Message Passing Interface. It utilizes the programming paradigm of message passing. In this paradigm, processes communicate with each other by explicitly invoking send and receive operations for exchanging messages [SGMHS17]. Costs for such operations are determined by the concrete topology of the communication network.

2.3.2 Terminology

Let $T^*(n)$ denote the execution time of the best known sequential algorithm for a problem. Furthermore, let $T(p, n)$ denote the execution time of some parallel algorithm solving the same problem, where p is the number of processes used. The *absolute speedup*, often simply called *speedup*, relates these two and is defined as

$$S_{abs}(p, n) = \frac{T^*(n)}{T(p, n)}.$$

It measures the improvement of parallel algorithms (solving the same problems) over a baseline sequential one. The best (absolute) speedup is called *linear speedup* and is p. Often the challenge is determining the number of processes, for which linear speedup can be achieved as this is the most efficient configuration. Similarly, *relative speedup* is given by

$$S_{rel}(p, n) = \frac{T(1, n)}{T(p, n)}.$$

It relates the execution time of a parallel algorithm executed with an arbitrary number processes against the execution time of the same algorithm when executed with just a single process. Both notions, absolute and relative speedup can be used formally (with asymptotic run times) as well as empirically (with measured run times) [RR10].

CHAPTER 3

Method

The idea of DistributedBallFilter is to simply split the point cloud into chunks along a 3D grid and run the original BallFilter algorithm on each chunk in parallel. The overall result is the union of all chunk results.

The essential steps performed by BallFilter are calculating the Delaunay complex of the entire point cloud and filtering the triangles of neighboring tetrahedrons by the δ-merged condition. DistributedBallFilter will need the additional steps of splitting the input, distributing the chunks as well as merging the outputs of BallFilter for each chunk back together to create the final result.

In designing DistributedBallFilter our goal is to be able to carry out as much work as possible in parallel while also maintaining clear interfaces between each step. In the following, we will discuss each step as well as our approach to parallization and highlight important theoretical considerations.

3.1 Splitting into overlapping tiles

The approach for input subdivision was developed by Brunner and is thoroughly explained in their dedicated work [Bru22]. Therefore 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 so-called *tiles*. Each tile is a set of points containing all points within a grid cell as well as points within a certain *padding* around the cell. The padding is necessary to ensure the correctness of the result. Tiles can be processed by the original BallFilter independently from each other and thus may be processed in parallel.

Let p be the input point set and $s = x \times y \times z$ the number of cells in the grid and x, y and z be number of grid cells along each dimension. The result of the splitting step is a set of tiles with the following properties:

$$T_p = Split(p) = \{t_0, t_1, t_2, ..., t_{s-1}\} \tag{3.1}$$

$$\bigcup T_p = \bigcup_{t \in Split(p)} t = p. \tag{3.2}$$

The union of all elements of the set of all tiles T_p must be equal to the original input point cloud.

3.1.1 Correctness

The union of the Delaunay complex of all grid cells (without padding) is not equal to the Delaunay complex of the entire point cloud as there cannot be edges between vertices in neighboring cells. Visually, it leads to cuts along the splitting grid in the resulting model. This problem is taken care of by adding the *padding*. Each tile contains points within a grid cell as well as within a certain padding around the grid cell. Therefore, the set of tiles is not a partition of the input, but a set of overlapping subsets.

The condition for *correctness* of the result of DistributedBallFilter may be expressed as

$$\bigcup_{t \in T_p} BF(t) = BF(p)$$

where $BF(p)$ is the result of executing BallFilter on a set of points p.

Correctness can be guaranteed by choosing the padding as

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

in all directions (to take into account all the triangles that could be grouped in the current simplices) and another padding of l in the positive direction (such that we do not miss any triangles with length of at most l), where $l = \frac{1}{2000} diagonal$. *diagonal* is referring to the diagonal of the axis-aligned bounding box that contains all points of the input point cloud. The proof for this correctness guarantee is outside of the scope of this paper.

For detailed reasoning about the padding size and its relation to correctness, please refer to [Bru22].

3.1.2 Tile membership and output size

To determine the tiles a point is a member of (*tile membership*), a single test against the 3D grid is needed and eight tests (one for each side) against the padding. Although there may be more optimal ways of doing this, it shows the theoretical bound of $O(1)$ for determining tile membership. Therefore, calculating the tile memberships of all points of a point set of size n can be done in $O(n)$.

The padding leads to the duplication of some points of the input point set. The number of points in all tiles $n' = |\bigcup T_p|$ is dependent on the chosen padding. In particular, the maximum number of tiles that may be overlapping each other dictates the factor of duplication. For example, let us assume the padding is very large. There may be regions, where all tiles overlap. As all points might be in these regions, each point might be contained in every tile. Thus their total size would be $n' \leq sn$, s being the number of cells in the grid. Another example would assume, the padding is rather small, covering only less than half of the neighboring cells. This way, each tile only overlaps directly neighboring tiles. The maximum area of overlap is around the corners of each tile, where eight tiles overlap. A single point can therefore be in up to eight tiles, creating an upper bound of $n' \leq 8n$.

For the sake of this work, we will assume the total number of points after splitting is an unknown, constant factor c of the number of points in the original point set and thus

$$n' = cn \in O(n).$$

3.2 Work distribution

The next step is to assign the independent chunks of work, the tiles, to a fixed number of p *processes* (commonly referred to as *machines* or *nodes*) and to communicate the tiles to those processes. We assume all processes are capable of performing the same amount of work in the same time (assumption of identical machines). The total running time is the running time of the slowest process. There is no assumption about the distribution of the points, the tiles might not be balanced. There may be significant differences in the number of points between tiles. The challenge is to assign tiles to processes in such a way that the total running time is minimized. This problem is well known and studied and called *Load balancing* and it is a type of *Scheduling problem* [KT06].

Using the terminology of load balancing, we call an execution of BallFilter for a single tile a *job*. Each job has a specific execution time. These execution times can be estimated by the number of points contained within the tile corresponding to the job. Therefore, balancing out the number of vertices across all processes is equivalent to balancing out load.

The problem of calculating an optimal assignment of jobs to processes is NP-hard. While finding exact solutions is computationally complex, there are several fast algorithms for approximating an optimal solution. One simple approximation algorithm is called *list-scheduling*. The list-scheduling algorithm keeps track of the sum of the execution times of all jobs that have been assigned to each process. In every step it assigns an unassigned job (in arbitrary order) to the process that currently has the lowest total execution time. This procedure is repeated until all jobs are assigned. Using a priority queue, list-scheduling can be done in $O(s \log p)$ (s being the number of jobs/tiles and p the number of processes) and was proven to be a 2-*approximation* of the load-scheduling problem, i.e. running the job schedule created by list-scheduling takes at most twice as long as the optimal schedule [KT06]. This bound can be drastically improved to 4/3 by pre-ordering the jobs by their execution time in descending order before assigning them. This is referred to as the *longest-processing-time-first (LPT) rule* [Xia04]. Sorting the jobs takes $O(s \log s)$ for a number of s jobs. Once sorted, assigning the jobs, takes $O(s \log p)$ for s jobs and p processes. The number of processes cannot exceed the number of jobs as otherwise there would be unnecessary idle processes. Therefore $s \geq p$ holds and implies $s \log s \geq s \log p$. Consequently, the total running time for LPT list-scheduling is in $O(s \log s)$.

We are using this method because it provides a good compromise between run time as well as theoretical bound for the quality of the solution. The number of tiles and processes will remain relatively small $p \leq s \leq 64$ for our evaluation. Calculating the assignment will not be done in parallel, since its running time is negligible compared to the other steps. While it may benefit the overall running time to employ parallelism for this step, implementing it is not in the scope of this paper. To avoid the cost of communication, each process computes the complete assignment for itself and picks out its own assigned jobs (tiles) for processing.

3.3 Processing tiles

Now 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 of the original point cloud.

As already discussed, BallFilter has a run time bounded by $O(n \log n)$ for n points. After work distribution each process has a set of local tiles. The run time of a single process is the sum of the run times of BallFilter for all tiles LT assigned to this process, thus

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

The term with the largest tile size dominates this sum, so the largest tile of each process dictates its run time. The overall run time is determined by the slowest processes. The slowest process is the one with the largest tile out of all process-local largest tiles, thus the largest tile out of all tiles. Therefore the overall bound can be expressed as

$$O(|t_i| \log |t_i|),$$

where for all $t_j \in T : |t_i| \geq |t_j|$.

This reveals one important property of DistributedBallFilter: its run time is entirely dependent on the distribution of points within the tiles, which itself depends on the distribution in the point cloud, the chosen grid and padding.

In the best case, the optimal job assignment is perfectly balanced, meaning every process is assigned exactly $\frac{n'}{p} = \frac{cn}{p}$ points. Note that n' is the number of points including duplicates after splitting the initial point cloud. Because the largest tile has to be assigned to some process and we assumed an optimal schedule where each process has $\frac{cn}{p}$ points (in one or more tiles), for the largest tile $|t_i| \leq \frac{cn}{p}$ holds. This directly implies a run time bound of

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

for the entire schedule.

In the worst case, the optimal solution to the load balancing problem assigns almost all points to a single process. This may happen when the tile sizes are extreme unbalanced and there are not enough tiles to compensate for this, e.g. if two tiles with sizes $cn - 1$ and 1 should be assigned to two processes, then the optimal schedule would assign one tile to each processes. As the largest tile may be of size cn, the run time of the slowest process and thus the whole schedule would be

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

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 choosing good parameters for input splitting.

3.4 Merging results and output

Upon finishing executing BallFilter on an assigned tile, the result is added locally to the set of all locally computed triangles. Subsequently, each process sends all triangles computed from their assigned tiles to a single process, which unions all received sets and outputs the final model.

As already mentioned, splitting the input leads to duplicate vertices within regions where multiple tiles overlap. When we take the union of the sets of triangles resulting from applying BallFilter to the tiles separately there may also be duplicate triangles because of those duplicate vertices. Removal of those triangles is possible as a simple post-processing step. However, they do not lead to artifacts and the number of duplicate triangles in the final result is relatively low. Thus, for now we deem it unnecessary to remove them.

3.5 Summary

To summarize, DistributedBallFilter performs the following steps when run on a set of n points. 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 $O(n)$, n number of sampled points
2. Calculate schedule in $O(|T|\log|T|)$ where T is the set of tiles.
3. Perform BallFilter in $O(|t_i|\log|t_i|)$, where t_i is the largest tile.
   Worst case $O(n\log n)$, best case $O(\frac{n\log n}{p})$, where p is number of processes.
4. Merge results from nodes in $O(n)$.

The overall asymptotic run time complexity is the sum of the complexities of the longest path in the DAG, that is $O(n + |T|\log|T| + |t_i|\log|t_i| + n)$. This is dominated by $O(n + |t_i|\log|t_i|)$ which in the worst case is $O(n + n\log n) = O(n + n\log n)$ and in the best case $O(n + \frac{n\log n}{p})$. Note, that the coefficient of n is expected to be very low, as splitting is done on the GPU and merging itself is a very simple operation.

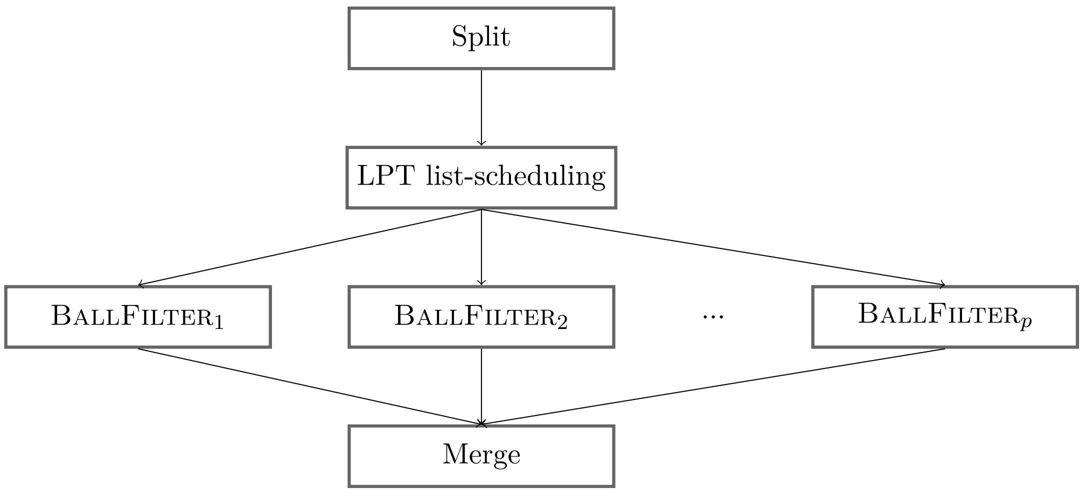


Figure 3.1: DAG of tasks performed in DistributedBallFilter

CHAPTER 4

Implementation

In this chapter we explain how the algorithm was implemented in software. First we discuss what technology was used and why. Then we explain how the existing implementation of BallFilter was adapted to achieve distributed-memory parallelism.

4.1 Technology

The most common framework for implementing distributed-memory parallelism in high-performance computing is the Message Passing Interface (MPI) [Mes21]. MPI is a standard that defines a large set of operations enabling processes running on different nodes of the cluster to communicate with each other via explicit function calls. We use C++ with the open source MPI implementation *OpenMPI* for communication [Pro22]. Only the operations for sending and receiving single message, `MPI_Send` and `MPI_Recv` are needed.

The splitting step of DistributedBallFilter involves a large number of simple operations on each points of the input point cloud. We already mentioned that the splitting step will not be distributed. Instead we implement this step using *CUDA*, to utilize the many cores of a graphics card. *CUDA* is a programming interface for NVIDIA GPUs, allowing code to be executed directly on the GPU (thus enabling large scale shared-memory parallelism) [NVI22].

We will be reusing parts of the original implementation of BallFilter as developed by Ohrhallinger [Ohr22]. This implementation uses the *CGAL* library. *CGAL* is a header-only C++ library that provides an extensive collection of data structures and algorithms for various mathematical problems [The22]. We will be using its shared-memory implementation for calculating the Delaunay complex of a given set of points.

The VSC-3+ cluster is managed by the *SLURM* cluster manager. It handles resource allocation and job queuing/scheduling on the cluster [Sch21]. We will use it to run DistributedBallFilter on a varying number of nodes to analyse its scaling behavior.

The VSC cluster also utilizes a high-performance distributed file system called *BeeGFS*. It allows multiple nodes to access a shared file-system in parallel and provides fast read and write performance. We will be using *BeeGFS* for communicating the tiles to the processes they were assigned to.

4.2 Necessary changes to BallFilter implementation

Splitting the input point cloud and performing surface reconstruction are independent of each other. This independence is reflected by implementing both processing steps in separate executables. The representation of tiles as files provides a common interface for splitting output and surface reconstruction input. This way, both parts may be swapped out with different implementations independently and it is easily possible to add pre-processing steps on a per-tile basis.

4.2.1 Splitting

The executable responsible for splitting the input takes the point cloud as well as the number of cells along each dimension as parameters. It outputs a set of files, each of which represents a single tile (*Tile file*), which contains a set of points. Each tile file is identified by an unique ID that corresponds to the tiles position in the 3D grid and is named accordingly. The tile files are written to the shared file system thus making all tiles available to all nodes. The implementation of the splitting step using *CUDA* is explored in [Bru22]. We will instead focus on performing surface reconstruction in parallel.

4.2.2 Distributed-memory parallel execution of BallFilter

The executable for surface reconstruction takes a set of tile files as input. Every process has access to all files via the distributed file system. Computing the schedule on a single node requires communicating the IDs of the assigned tiles to each process. Instead, the communication overhead is saved by letting each process compute the schedule itself and then read the tiles files it needs from the distributed file system. Also one node with the lowest load (according to the calculated schedule) is recognized as the *root node*. This node will be responsible for merging and writing the final result.

Then every node runs the original BallFilter implementation on the assigned tiles. Calculating the Delaunay complex is the bottleneck of BallFilter, and thus benefits most from (shared-memory) parallelism. The original BallFilter already uses a shared-memory parallel implementation of the Delaunay complex calculation supplied by *CGAL*. The filtering based on the δ-merged property on the Delaunay simplices is done in serial utilizing just a single core. Implementing this using shared-memory parallelism, while beneficial to the running time, is not in the scope of this paper. The resulting sets of triangles, represented by indices of vertices in the input point cloud, are merged in memory.

When the root node is done with reconstructing the surface within all its assigned tiles, it writes the vertices of the input point cloud to the output file as well as the locally calculated triangles. Upon completion of all of their assigned tiles, each non-root node sends its local results to the root node, which immediately writes them to the output file. After the root node received results from all other nodes, DistributedBallFilter is completed and the output file contains a 3D model of the reconstructed surface from the original input point cloud.

CHAPTER 5

Results and Evaluation

In this chapter, we will discuss the results of our implementation of DistributedBallFilter on specific data sets and parameter combinations. First we will give an overview of the hardware environment the algorithm is run in as well as explain the data sets and parameter combinations selected for testing. Moreover we will list and visualize the running times for specific data sets and parameter combinations and reason about the behavior when scaling the number of processes p or the input size n. Furthermore we will draw comparisons to the shared-memory implementation of BallFilter this paper is based upon.

5.1 Datasets and run-configurations

In this section the environment for the timed runs of BallFilter and DistributedBallFilter is presented. This includes the exact hardware, data sets and parameters used for the runs.

5.1.1 Hardware configuration

The VSC-3+ cluster consists of various types of nodes with different hardware specifications [Clu22]. We will be using two node types for the purpose of empirically analysing the performance of our implementation of DistributedBallFilter. For splitting and outputting the tile files to the distributed file system, we will be using a single node with a single NVIDIA Pascal GeForce GTX 1080 GPU. For processing the tile files and outputting the reconstructed model, we will be using 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 will be run multiple times while varying the number of nodes used in order to analyze the scaling behavior of DistributedBallFilter.

5.1.2 Datasets

As input point clouds for testing, two data sets have been selected, both obtained via photogrammetry from pictures taken by drones. Both were provided from *Pix4D* [Pix22]. They were chosen based on their large size and real world relevance. The point clouds are 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.

5.1.3 Parameters

Besides the data sets, there are a number of relevant parameters we need to consider in order execute DistributedBallFilter on the cluster and obtain meaningful running times. Apart from the input point cloud the following parameters are necessary.

1. The parameters for BallFilter itself are the threshold for the intersection ratio δ and the threshold for the filtering based on the bounding box diagonal $tlen$. These parameters only influence the quality of the resulting 3D model and generally do not impact performance in a significant way. For this reason, we keep them fixed at the values generally recommended for real-world scans for both, the runs of BallFilter and DistributedBallFilter [Ohr22].

2. Splitting the input requires three parameters, x, y and z, which denote the number of tiles along each axis respectively. This determines the total number of tiles $s = x \times y \times z$. Correctly choosing these parameters is vital for work distribution and in turn has a great impact on the overall running time. The number of tiles will also influence the number of duplicate points because each tile contains some points that are contained in neighboring tiles as well. Given a specific point cloud, an inappropriate choice of x, y and z may result in unbalanced tile sizes, leading to longer running times. 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 generally be picked based on the number of nodes to employ.

   Furthermore, for point clouds with an uneven distribution of points there is no regular grid subdivision that yields a balanced set of tiles. For these cases, a better suited approach would be to use a k-d tree or an octree for splitting the input rather than a regular grid. Implementing, testing and comparing these approaches may be done as separate work in the future.

3. Executing DistributedBallFilter on a cluster requires the number of nodes p to be specified. This parameter's influence on performance 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. Having more tiles allows for better work distribution because smaller chunks can be distributed more evenly.

However using much more tiles than nodes leads to considerable overhead by point duplication. In our tests, we will vary this parameter to observe its impact on the running times. Generally, using more processes should result in lower running times.

Tiles are the unit of work that can be distributed to nodes and calculated independently from each other. Consequently, having less tiles than there are processes would leave some processes without work. In such a case, the same set of tiles can be processed in the same time by less nodes. Therefore, the number of tiles s must not be smaller than the number of processes p to avoid unnecessary idle times, thus $s \geq p$. Nevertheless, more tiles lead to more duplicate points, increasing the overall work that has to be performed, which may in turn lead to overall higher running times.

The fastest way to process a set of (a fixed number of) s tiles is to process each tile on a dedicated node, thus setting $p = s$. For sets of tiles where the number of vertices per tile is roughly equal across all tiles i.e. *balanced tile sets*, this results in a good distribution of work across all nodes. However, if the tile set is unbalanced, the nodes processing (very) small tiles remain idle until the nodes with (very) large tiles are finished. Because nodes remain allocated for the entire running time of the algorithm, CPU time is wasted.

In cases of imbalance, the same running time can be achieved using less nodes. Such imbalances may be a result of poor choice of parameters (x, y, z) or simply be the consequence of the distribution of points within the input point cloud. This case is illustrated in 5.1b and 5.1c. In this example, a set of four tiles need to be processed. Allocating four nodes would mean each node processes exactly one tile. As the number of vertices within each tile vary greatly, nodes 1, 2 and 3 receive much less work compared to node 0. After finishing the small tiles, the nodes stay allocated and have to wait for node 0 to finish the larger tile. Scheduling the same four tiles on only two nodes leads to the same running time, because all the small tiles can be processed by a single node while the other node processes the large tile. Formalizing the exact conditions under which a set of independent tasks can be processed by fewer nodes in the same time is not within the scope of this paper.

Following from the above argument, when allocating a number of p processes, it might be faster to split the input in more than p tiles. Having more tiles than there are processes allows the scheduling algorithm to compensate for unbalanced tile sets, resulting in better load balancing and (much) less idle time. By this approach, unequal point distributions in the tiles may not necessarily lead to unbalanced work. While this improves hardware utilization, in cases with balanced tiles it may actually increase overall running times compared to using exactly one node per tile. In cases of unbalanced tiles however, it might actually decrease the running time, because it allows large chunks (that otherwise all processes would have to wait upon) to be subdivided and scheduled on multiple processes. This case is illustrated in 5.1a and 5.1c. In this example two processes are used. Here it is actually faster to split the input into four tiles rather than two, because one of the two tiles would contain almost all of the points and serve as a bottleneck to

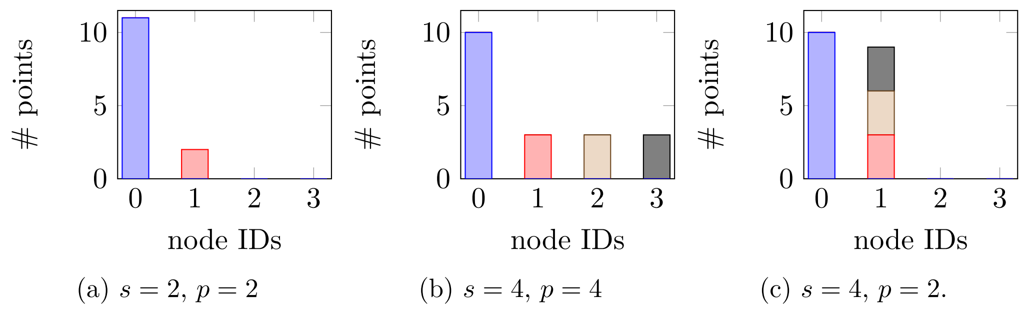


Figure 5.1: Comparison of using $s = p$ vs. $s = 2p$ for unbalanced tiles: four tiles might be processed as fast with two processes as with four. Additionally, two processes might run faster when processing four tiles rather than two, depending on balance.

overall completion. Note that with four tiles, the total number of points to process is larger then with two. The reason for this is the need to add duplicate points to each tile, in order for the results to be a correct reconstruction.

Our input data sets are rather unbalanced, suggesting that in real scan data, the occurrence of unbalanced tile sets is commonplace. This makes sense considering that some tiles might entirely lie inside the interior of a scanned object and thus contain no points at all. For those point clouds, using another subdivision strategy, such as a k-d tree or octree, would be more suitable. However, the splitting step would become more complex and harder to execute in (both, shared- and distributed- memory) parallel. When a larger number of small tiles is used, the scheduling algorithm can still compensate for a certain degree of imbalance within the tile sizes. It is assumed, that this strategy still leads to good results with tile sets that are moderately unbalanced.

Generally, we could choose the number of processes (manually) depending on the concrete set of tiles that was output by the split step. In practice, because of its low running times sometimes it would also make sense to repeat the split step with varying parameters, until a balanced tile set is found. For analysis however, it is important not to adapt the parameters to specific cases or point distributions in the input data. This means we can not choose the number of processes depending on the actual vertex counts within the tiles.

For the balanced cases, $p = s$ would be optimal. For the imbalanced cases we cannot know beforehand how many processes we should choose. In figure 5.1b, only half of the processes ($p = \frac{s}{2} \iff s = 2p$) is actually needed. A more extreme case could a set of $s = 8$ tiles, one very large one and seven very small ones. If the seven small ones combined are still smaller than the large one, we could process the tile set on only two nodes without increasing the overall running time. In this case, $p = 2$ would be optimal and thus $p = \frac{s}{4} \iff s = 4p$ would hold. However, using more tiles without using more processes is generally less desirable since it adds more duplicate points without utilizing parallel resources.

BallFilter		**Splitting**				**Cluster**
δ	$tlen$	**cells along x-axis**	**cells along y-axis**	**cells along z-axis**	**tiles s**	**nodes p**
1.35	200	2	1	1	2	1
		2	2	1	4	2
		2	2	2	8	4
		4	2	2	16	8
		4	4	2	32	16
		4	4	4	64	32

Table 5.1: Parameter combinations used with each data set.

As a general rule we set $s = 2p$ which means we use double the amount of tiles than we have processes. As already discussed, it is not optimal for extremely balanced ($s = p$ would be) or imbalanced cases ($s = cp$ with $c > 1$, c dependent on exact point distribution). Nevertheless, in the average case with decent but not extreme imbalance, it leads to good work distribution and in consequence, a high degree of hardware utilization as well as fast running times. Depending on the point distribution, it might allow for better running times compared to using less tiles when it leads to the subdivision of large tiles. It allows for good speed up, while keeping the total number of duplicate points that need to be processed relatively low.

The concrete combinations of parameters are listed in table 5.1.

5.2 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 5.2 as well as the absolute and relative speed up. T_{split} encompasses the time for reading the input point cloud, splitting its vertices into tiles, duplicating vertices as necessary and writing them to disk as separate files. This is done on a single node with a GPU. These times are the same for all runs with the same x, y and z, because the tile files output by a single run of the split step can be used for multiple runs of the reconstruction step. $T_{reconstruct}$ is the combined time for starting p processes, calculating the tile assignment, reading the files, executing BallFilter on each tile, merging the results and writing it to disk. This is done on p nodes of the VSC cluster.

The running time of the split step are consistently very low. There are no significant changes in the times for different numbers of tiles. In comparison the running times of the reconstruction step are dominating the total running time of DistributedBallFilter. Combined with the fact that 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.

	p	s	x	y	z	T_{split}	$T_{reconstruct}$	T_{total}	S_{abs}	S_{rel}
eclepens 16 million points	1	1	1	1	1	1.51443	90.126	91.64043	0.87	1.00
	2	2	2	1	1	1.35175	48.1534	49.50515	1.60	1.85
	4	4	2	2	1	1.59808	34.4655	36.06358	2.20	2.54
	8	8	2	2	2	1.3445	26.3647	27.7092	2.87	3.31
	16	16	4	2	2	1.2544	20.1795	21.4339	3.71	4.28
	32	32	4	4	2	1.5654	16.4233	17.9887	4.41	5.09
	1	2	2	1	1	1.51443	88.8745	90.22625	0.88	1.00
	2	4	2	2	1	1.35175	53.7387	55.33678	1.43	1.63
	4	8	2	2	2	1.59808	31.4786	32.8231	2.42	2.75
	8	16	4	2	2	1.3445	22.4666	23.721	3.35	3.80
	16	32	4	4	2	1.2544	14.8993	16.4647	4.82	5.48
	32	64	4	4	4	1.5654	11.8065	13.47371	5.89	6.70
eclepens 32 million points	1	2	2	1	1	1.85363	170.587	172.44063	0.85	1.00
	2	4	2	2	1	1.56998	90.379	91.94898	1.59	1.88
	4	8	2	2	2	1.72504	62.5874	64.31244	2.28	2.68
	8	16	4	2	2	1.62856	48.895	50.52356	2.90	3.41
	16	32	4	4	2	1.75069	29.9302	31.68089	4.62	5.44
	32	64	4	4	4	1.84338	26.1637	28.00708	5.22	6.16
	1	2	2	1	1	1.85363	168.339	169.90898	0.86	1.00
	2	4	2	2	1	1.56998	96.3045	98.02954	1.49	1.73
	4	8	2	2	2	1.72504	61.7569	63.38546	2.31	2.68
	8	16	4	2	2	1.62856	36.3197	38.07039	3.84	4.46
	16	32	4	4	2	1.75069	25.6505	27.49388	5.32	6.18
	32	64	4	4	4	1.84338	23.7843	26.03393	5.62	6.53

Table 5.2: Parameter combinations and running times for `eclepens`

Data set	Num. points n	shared	distributed $p = 16$, $s = 32$	S_{abs}
eclepens	1 000 000	4.75855	2.628479	1.81
	2 000 000	9.72407	5.268556	1.85
	4 000 000	18.2101	7.887388	2.31
	8 000 000	47.0571	11.432506	4.17
	16 000 000	79.414	16.4647	4.82
	32 000 000	146.329	27.49388	5.32
	68 717 620	310.002	54.72936	5.66
matterhorn	274 890 274	1613.62	317.40777	5.08

Table 5.3: Comparison between BallFilter and DistributedBallFilter on inputs of varying sizes

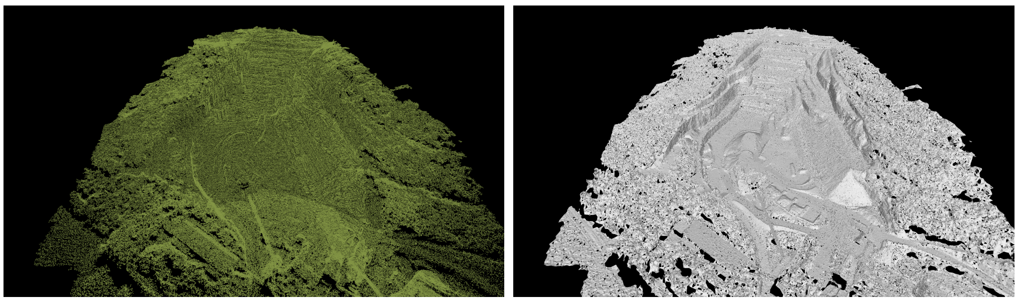


Figure 5.2: Input point cloud (left) and reconstruction (right) of the `eclepens` (8 million points) data set calculated using DistributedBallFilter with 16 processes and 32 tiles in 11.43s

5.3 Analysis of scaling behavior and speed-up

First we will discuss the behavior when scaling the input size n. Figure 5.3 and table 5.3 show the running times of the original (shared-memory) implementation of BallFilter and our implementation of DistributedBallFilter for different versions of the `eclepens` data set. The data set has been truncated to specific sizes (from $2^0 = 1$ to $2^5 = 32$ million points) to simulate growing input size. For each truncated data set version, the times for the original (shared) implementation as well as for the distributed implementation using 16 nodes and 32 tiles was run.

BallFilter has an asympotic 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. This means, the distributed version is in the worst case (asymptotically) as fast as the original algorithm and in the best case faster. 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 thus matches the one of BallFilter. Therefore the only difference between both running times lies in the coefficient (which is ignored by asymptotic complexities).

In our example, for all input sets the distributed version was faster than the shared version. Both curves look similar, as their asymptotic running times suggests. The absolute speed up increased with the size of the input and the running time up to a factor of 5.32. The best possible absolute 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. As already discussed, in order to utilize more nodes, more tiles are needed. For one series of runs, the number of tiles was set to the same as the number of nodes ($s = p$), so there

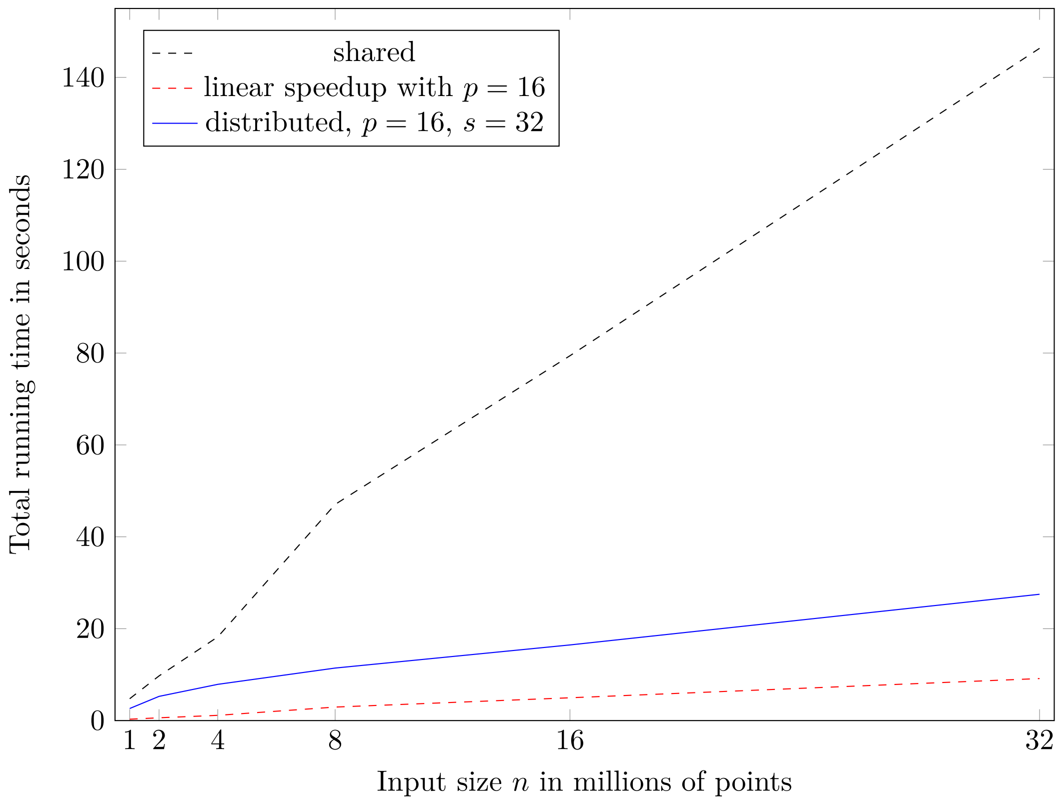


Figure 5.3: Running times of the original BallFilter vs DistributedBallFilter with increasing input size for `eclepens`

would be one dedicated node per tile. For the other series, the number of tiles was twice the number of nodes ($s = 2p$), so that in cases of imbalance, the scheduling algorithm could balance out the work between the processes. The concrete timings are listed in table 5.2 and visualized in figure 5.4.

For one node, both runs of DistributedBallFilter were slower than the original implementation. That is to be expected because of the overhead required by the distributed implementation (e.g. splitting the input in a single tile, initializing *MPI*). Using two or more nodes, however, significantly decreased the running times compared to the original shared-memory implementation of BallFilter.

As the absolute and relative speedup approach a value of around 6, both the running times stay the same even when increasing the number of nodes. The reason for this is that for larger numbers of tiles the additional work caused by duplicate points becomes noticeable. Eventually, it cancels out the performance gain that could be achieved by further splitting the input and processing it in parallel on more nodes.

Another observation is the fact that apart from $p = 2$, the runs with $s = p$ were consistently slower than those using $s = 2p$. The reason for this becomes apparent

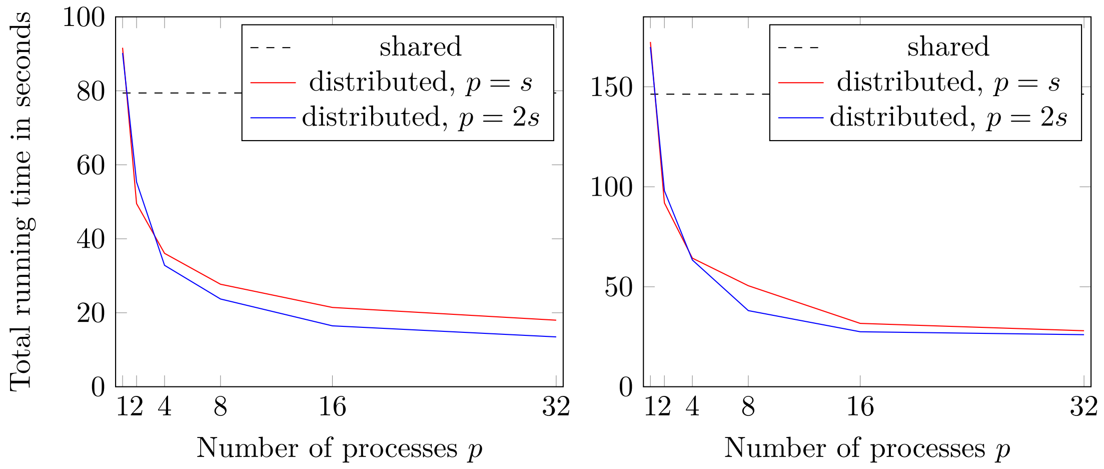


Figure 5.4: Absolute running times of the original BallFilter vs DistributedBallFilter with one tile per process and two tiles per process respectively, run on `eclepens` with 16 million (left) and 32 million (right) points.

when examining the distribution of the points within the point clouds as well as their assignment to the nodes, visualized in figure 5.5. The running time of one process can be estimated by its load. The process with the highest load has the longest running time and thus determines the overall running time. For two nodes, the $s = p$ version assigns one tile to each node, while the $s = 2p$ version assigns two to each node. Coincidentally, the loads for node 0 are similar in both tile assignments, slightly higher in the $s = 2p$, which also contains more duplicate points. Therefore, the $s = p$ version was slightly faster. The assignment for four processes reveals that with eight tiles, the highest process load is still considerably lower than when only four tiles are used, despite the higher number of duplicate points. Therefore, the work can be better distributed among the nodes, resulting in an overall lower running time compared to only using four tiles. This behaviour is also present in the runs with more processes, resulting in shorter running times for $s = 2p$.

As already mentioned, using p nodes enables the fastest running times for processing s tiles. Nevertheless, using $s > p$ tiles on p processes leads to a lower running time compared to using exactly p tiles because its allows for a more balanced distribution of work. Choosing s much higher than p however leads to a larger number of duplicated points, eventually limiting running time improvements achievable by better work distribution.

To summarize, DistributedBallFilter performed good 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 was faster by a factor of 5.66 than BallFilter. We also observed that having a higher number of tiles, can and in many cases will decrease the overall running time because it enables better work distribution, despite requiring more duplicated points. Figure 5.3 shows that,

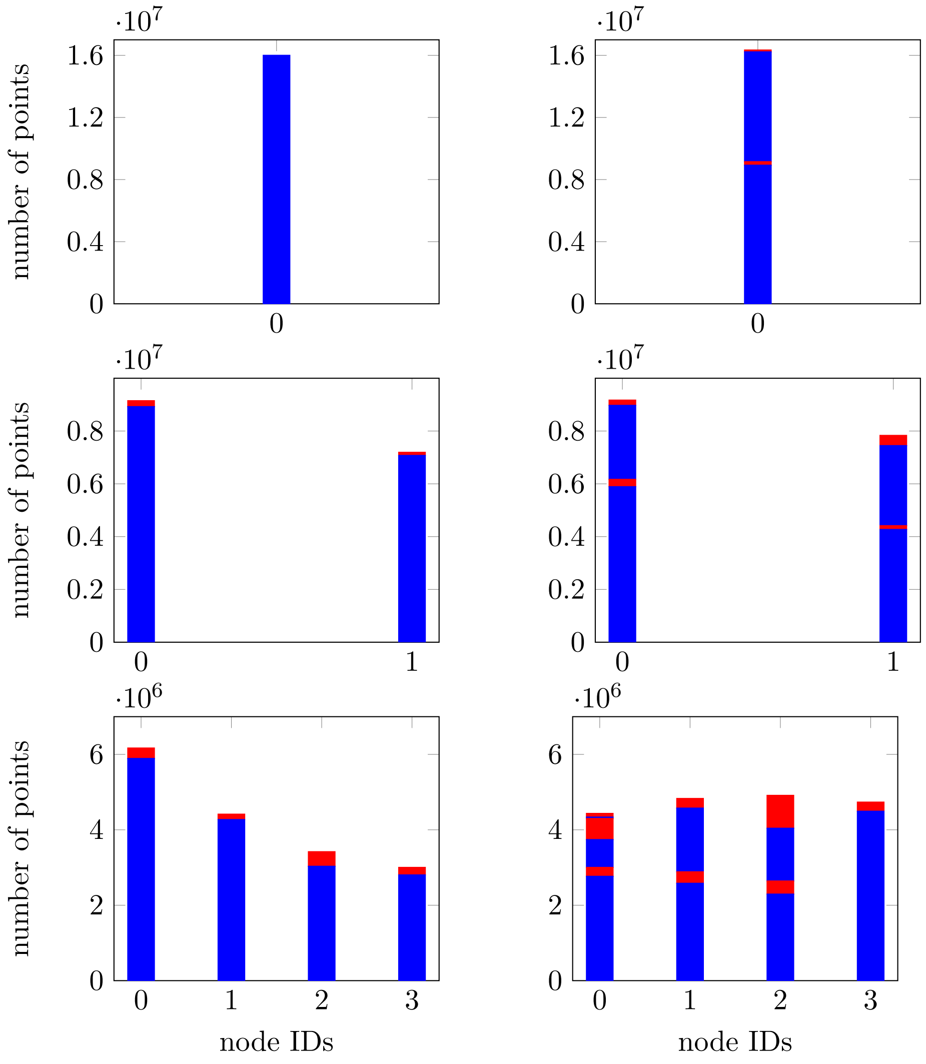


Figure 5.5: Assignment of tiles to nodes, for using $s = p$ (left) vs. $s = 2p$ (right) with one (first row), two (second row) and four (third row) nodes. Blue segments represent original points, red segments represent duplicated points.

DistributedBallFilter is a large improvement towards running times achievable by linear speedup.

CHAPTER 6

Conclusion and Future Work

Finally, in this chapter we will briefly summarize this paper's result, as well as give a short outlook on future work that could be done on DistributedBallFilter.

6.1 Summary

In this thesis we have presented a distributed-memory parallel algorithm for 3D surface reconstruction called DistributedBallFilter that is based upon the BallFilter algorithm. The idea is to split the input point cloud along a 3D grid into overlapping chunks. For each chunk, the surface reconstruction using original BallFilter can be run independently and thus be executed in parallel on a number of nodes. After all chunks have been processed, the result is merged back into a single resulting 3D model.

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. For example, processing the `matterhorn` data set consisting of roughly 275 million points with 16 nodes took only one fifth of the time required by the original algorithm.

6.2 Future Work

While we looked at it 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.

Looking at the concrete running times of calculating the Delaunay complex and performing the check of the δ-merged property reveals that the latter actually ran slower than the former despite its better theoretical time complexity. We think the reason for this is the large number of processor cores available on the VSC3+ cluster nodes. In our runs, we utilized all 20 cores each node provides with the shared-memory parallel implementation of the Delaunay complex from the *CGAL* library. Our implementation, like the original BallFilter, uses only a single core for the δ-merged and *tlen* checks. Here we see the potential of further improving the performance of DistributedBallFilter by utilizing available cores in a shared-memory parallel implementation of this step.

List of Figures

List of Tables

Bibliography

[ABE98] 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.

[ACDL00] 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.

[ACK01] 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.

[Boi84] Jean-Daniel Boissonnat. Geometric structures for three- dimensional shape representation. *ACM Transactions on Graphics. October, 1984. vol. 3: pp. 266-286 : ill. includes bibliography*, 3, October 1984.

[Bru22] Lukas Brunner. Technical report - work in progress. TU Wien, 2022.

[Clu22] VSC Vienna Scientific Cluster. Vienna Scienfic Cluster (VSC) - website. `https://vsc.ac.at/systems/vsc-3/`, 2022. [Online; accessed 13-September-2022].

[Ede03] Herbert Edelsbrunner. *Surface Reconstruction by Wrapping Finite Sets in Space*, pages 379–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[EGO$^+$20] 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.

[EM94] Herbert Edelsbrunner and Ernst Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13, October 1994.

[GO08] Leonidas J. Guibas and Steve Y. Oudot. Reconstruction using witness complexes. *Discrete & Computational Geometry*, 40(3):325–356, October 2008.

[HK06] Alexander Hornung and Leif Kobbelt. Robust reconstruction of watertight 3d models from non-uniformly sampled point clouds without normal information. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP '06, pages 41—50, Goslar, DEU, 2006. Eurographics Association.

[KH13] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Trans. Graph.*, 32(3), July 2013.

[KT06] 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.

[Lea92] Geoff Leach. Improving worst-case optimal delaunay triangulation algorithms. In *In 4th Canadian Conference on Computational Geometry*, page 15, 1992.

[Mes21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[NVI22] NVIDIA Corporation. CUDA Toolkit - project website. `https://developer.nvidia.com/cuda-toolkit`, 2022. [Online; accessed 13-September-2022].

[Ohr22] Stefan Ohrhallinger. Personal communication. TU Wien, 2022.

[Pix22] Pix4D SA. Pix4D - website. `https://www.pix4d.com/`, 2022. [Online; accessed 13-September-2022].

[Pro22] The Open MPI Project. OpenMPI - project website. `https://www.open-mpi.org/`, 2022. [Online; accessed 13-September-2022].

[RR10] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*, chapter 4.2.1 Speedup and Efficiency. Springer Berlin Heidelberg, 2010.

[Sch21] SchedMD LLC. Slurm Workload Manager - project website. `https://slurm.schedmd.com/overview.html`, 2021. [Online; accessed 13-September-2022].

[SGMHS17] B. Schmidt, J. Gonzalez-Martinez, C. Hundt, and M. Schlarb. *Parallel Programming: Concepts and Practice*, chapter 9.1 Message Passing Interface. Elsevier Science, 2017.

[The22] The CGAL Project. CGAL - project website. `https://www.cgal.org/`, 2022. [Online; accessed 13-September-2022].

[Wie22] Stadt Wien. Wien Gibt Raum - project website. `https://digitales.wien.gv.at/projekt/wiengibtraum/`, 2022. [Online; accessed 13-September-2022].

[Xia04] 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.

[YLL$^+$20] 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.