FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Optimized Sorting for Out-of-Core Surface Reconstruction

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Sebastian Mazza
Matrikelnummer 00825828

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Dr.techn. Claus Scheiblauer

Wien, 5. April 2018

Sebastian Mazza	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Optimized Sorting for Out-of-Core Surface Reconstruction

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Sebastian Mazza
Registration Number 00825828

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr.techn. Claus Scheiblauer

Vienna, 5th April, 2018

Sebastian Mazza

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Sebastian Mazza
Ritzlingbachstraße 308, 3610 Weißenkirchen in der Wachau

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, 5. April 2018

Sebastian Mazza

Acknowledgements

I would like to thank Claus Scheiblauer for his support during my Bachelor project especially for his advice and feedback during the development of the Scanopy plugin.

Further I would like to thank Michael Wimmer for his help in order to achieve my goals.

Thanks to the Stanford Computer Graphics Laboratory for distributing their 3D models in the Stanford 3D Scanning Repository [Sta].

Last but not least I would like to thank my family, especially Franzi, for their endless support and motivation during the whole time. Special thanks to Carl and Edith who supported me in various ways.

Kurzfassung

In den letzten Jahren hat die Anzahl der Akquisitionsmethoden für Punktwolken kontinuierlich zugenommen und gewinnt zunehmend an Bedeutung für die Gesellschaft. Auch wenn es bereits möglich ist, Punktwolken direkt zu rendern, gibt es viel mehr Algorithmen, welche mit Dreiecks-Meshes arbeiten, als mit Punktwolken. Zum Beispiel benötigt die Software für 3D Drucker wasserdichte Meshes als Input. Diese Tatsachen machen das automatische Konvertieren von Punktmengen in Dreiecks-Meshes zu einem sehr relevanten Forschungsthema. Das Ziel dieser Bachelorarbeit war die Implementierung eines Plugins für Scanopy (ein Programm zum Rendern und Bearbeiten von Punktwolken), welches Punktwolken mit hunderten von Millionen von Samples in einem so hoch detaillierten Grad konvertieren kann, dass die Datenmenge übliche Hauptspeichergrößen übersteigt. Daher wurde ein Out-of-Core Algorithmus benötigt. Der verwendete Out-of-Core Poisson Surface Reconstruction Ansatz benötigt als Input sortierte Point Samples. Diese werden daher in einem Vorverarbeitungsschritt sortiert. In dieser Bachelorarbeit wird aufgezeigt, dass durch das Sortieren der Daten mit einem optimierten Multithreaded Mergesort Algorithmus die für die gesamte Rekonstruktion benötigte Zeit signifikant reduziert werden kann. Weiters wird in dieser Arbeit ein Problem dargelegt, welches bei der Rekonstruktion von Scans von offenen Terrains mittels Poisson basierter Rekonstruktion auftritt und zu großen und unnötigen Dreiecken führt, die die rekonstuierte Oberfläche verdecken. Ein grundlegender Lösungsansatz für dieses Problem wird ebenfalls beschrieben.

Abstract

In recent years the amount of acquisition methods for point clouds has been increasing consequently and it is getting more and more interesting for society. Even if it is possible to render point clouds directly, nowadays there exist many more algorithms which deal with triangle meshes than point clouds. For example 3D printer software requires watertight meshes as input. This makes automatic conversion of point sets to triangle meshes an important research topic. The aim of this Bachelor Thesis was to implement a plugin for Scanopy (a point cloud editing and rendering program) which can convert point clouds with hundreds of millions of samples in such a detailed degree that the data exceeds common main memory sizes. Therefore, an out-of-core algorithm was needed. The used out-of-core Poisson surface reconstruction approach requires the sorting of the input point samples in a preprocessing step. In this Bachelor Thesis it is shown that the sorting of the data with an optimized multithreaded merge sort algorithm can improve the total required time for the reconstruction process significantly. Further, this work indicates a problem which occurs while reconstructing meshes with a Poisson based reconstruction approach from scans of an open terrain. The problem leads to large unnecessary triangles which hide the reconstructed surface. A very basic solution approach for this problem is also stated.

Contents

CHAPTER 1

Introduction

The goal of this work was to develop a plugin for Scanopy (see Section 1.2) that can convert huge point clouds to triangle meshes. The implemented code is based on the out-of-core surface reconstruction from Bolitho et al. [BKBH07]. This work presents a sorting method that can significantly improve the total required time for the surface reconstruction approach from Bolitho et al. [BKBH07]. Details of the used multithreaded merge sort can be found in Chapter 3. Furthermore, a discussion about postprocessing algorithms that try to remove unnecessary triangles from the reconstructed surface can be found in Chapter 4.

This chapter contains a short introduction to point clouds (Section 1.1). Furthermore, Section 1.3 explains why an out-of-core surface reconstruction is required. Finally in this chapter Section 1.4 describes the motivation for the optimizations to out-of-core surface reconstruction that are covered by this work and gives a brief overview about it.

1.1 Point Cloud

A point cloud is a set of points. In a mathematical sense a point defines only a location in a space. It does not occupy any space because it is infinitely small. In the context of this thesis, a point defines a rendering primitive with at least a 3 dimensional vector that defines the location of the point in space. When a point gets rendered, its position gets visualized at least by one pixel on the screen. A point can also contain further information like a color, a normal vector or information about measurement precision.

Point clouds can for example be obtained from a range scanner. A range scanner is mounted on a fixed position during a scan operation. It calculates the distance of a point on a surface from a real world object by sending out a laser light pulse and measures the time it takes for the light to be reflected from the surface and reach a sensor on the scanner. Because the speed of light is a known constant, the distance can be calculated

from this measured time. The scanner rotates in discrete angles around its vertical and horizontal axis during the scanning. Therefore, the scanner measures the distances to the next surface along discrete angles. Because the angles are known for each distance sample, a 3-dimensional point relative to the position of the range scanner can be calculated for each sample. Furthermore, images can be taken during the scan. These images can be used to colorize a rendering of the point cloud. To scan larger areas it may be necessary to take multiple scans from different positions and merge the resulting point clouds together. This is also required if someone needs surface information from different sides of an object.

Laser scanners can also be mounted on a plane. Then the scanner only needs to rotate around the pitch axis of the plane, because the movement of the plane generates the second scan direction. For smaller objects scanners exist that move only up and down along the vertical axis and rotate the object that should be scanned with a turntable in front of the scanner.

Point clouds can also be obtained by photogrammetry. This is a technique that extracts 3D data from images. Explanations about fully automated 3D model reconstruction methods and algorithms can be found in [Sze11].

In the last years cheap and broadly available structured light scanners like the Microsoft Kinect have enabled nearly everyone to generate point clouds of a real world environment. Therefore, simple, fast and reliable algorithms for surface reconstruction have become more important than ever before.

All these acquisition methods have one problem in common: the produced data is not perfect. The property of the imperfections vary between the different acquisition methods, which leads to different challenges for the reconstruction algorithms. Imagine for example a range scanner. Because the scanner rotates along discreet angles, surfaces that are further away from the scanner are represented by less denser points than objects that are near the scanner. Therefore, point clouds that are obtained by a range scanner typically have a non-regular density.

1.1.1 Rendering of point clouds

In the last years many algorithms were developed that can render point clouds without a previous conversion from point clouds to polygonal meshes. Many of them are based on the pioneering Surfel approach [PZvBG00]. The basic idea behind the Surfel approach is to define a point as a small ellipsoidal disk with normal vector. This approach even allows to render point clouds at interactive frame rates.

However, many methods require a lot of preprocessing. Also, rendering of large point clouds that do not fit into the main memory was a big issue for many years. The first approach that was able to render large datasets from range-scanners and at least partially solved this problem was QSplat [RL00]. The QSplat approach requires a preprocessing step, where the a hierarchy of bounding spheres is created. This hierarchy is then used for

visibility culling, level-of-detail control and rendering. However, the creation of hierarchy of bounding spheres and the selection of the nodes that needs to be rendered is done on the CPU. The Instant Points approach from Michael Wimmer et al. [WS06] requires much less preprocessing because it does not make any assumptions about sampling density or availability of normal vectors for the points. Because Instant Points is an out-of-core algorithm, it can render a huge amount of points in realtime. A nested octree is used as data representation.

Today, even point-cloud rendering systems for web-browsers exist. "Potree: Rendering Large Point Clouds in Web Browsers" from Markus Schütz [Sch15] is based on the Instant Points approach [WS06]. The nested octree he used is optimized for fast streaming of point data through the network and, therefore, uses small chunks. For rendering of points in web browsers, Potree makes use of graphics hardware through WebGL.

1.2 Scanopy

Scanopy is a point cloud rendering and editing application written in C++, developed by the Institute of Computer Graphics and Algorithms at the Vienna University of Technology. It is designed for a large amount of point data. Scanopy can handle point clouds that do not fit in the main memory. Furthermore, Scanopy can read and write from and to different Point Cloud Data Formats. This allows the user to quickly preview the scan results in the field. Furthermore, it supports the registration of different range scans. Registration means to bring different range images from the same object to one coordinate system. This allows the user to see the whole picture and not only the result of one single scan.

1.3 Converting point clouds to triangle meshes

Although nowadays many methods working directly on point clouds exist, polygonal meshes still have a lot advantages. An innumerable number of methods, algorithms, tools and programs already exist that can handle polygonal meshes. If it is possible, to convert even point clouds with hundreds of millions of points to polygonal meshes, all of this already existing tools can be used to work with the data obtained by range scanners or other point cloud acquisition methods. Furthermore, a polygonal surface representation can be much more compact for many surfaces than a point-cloud representation, which leads to a considerably smaller file size. Imagine a surface with many flat areas like a cube, which can be defined by only twelve triangles, but a point cloud of the same cube would probably require at least some hundred points in order to describe the shape of the cube in a recognizable form.

1.3.1 Out-of-core surface reconstruction

Because of the advantages that polygonal meshes can provide, a Qt-plugin for Scanopy that can export triangle meshes should be developed. Converting point clouds to polygonal

meshes is a well studied problem in computer graphics. There are even ready to use applications that can perform surface reconstruction. Meshlab [CCC+08] is one example, it comes with three different algorithms for surface reconstruction. The goal was to develop a plugin that is able to convert point clouds with hundreds of millions of points. Therefore, an out-of-core algorithm is required because such big point clouds do not fit into present typical main memories. The representation of the implicit function required by approximating surface reconstruction approaches that support globally smooth results (see Section 2.1.2) can take even more space than the point cloud itself.

Most parts of the plugin for Scanopy that was developed by the author is based on the work "Multilevel Streaming for Out-of-Core Surface Reconstruction" from Bolitho et al. [BKBH07]. They also provide a working implementation [BKBH] of their approach, which greatly helped to understand the details of their out-of-core surface reconstruction approach and to develop the point cloud to mesh conversion plugin for Scanopy.

1.4 Developed optimizations for out-of-core surface reconstruction

The first step of development was to port the C++ source code from Bolitho et al. [BKBH] into a plugin for Scanopy. After fixing lots of incompatibilities with Microsoft's Visual Studio 2010, smart pointer implementations, string representations and many other problems, a Qt-based GUI for the plugin was created. During the tests of the newly written Scanopy plugin with our own data sets, some more challenging insufficiencies of the approach [BKBH07] were discovered.

1.4.1 Sorting runtime

The first improvement that was made by the author did not change anything on the reconstructed polygonal mesh but did greatly improve the overall runtime of the conversion. The implementation of the out-of-core surface reconstruction from Bolitho et al. [BKBH] uses a quicksort implementation for sorting points during the preprocessing. This sorting operation often takes more time than the actual reconstruction process (see Table 5.1). Therefore, a faster sorting method can greatly improve the overall runtime of the surface reconstruction process.

In Chapter 3 the author will present one possible implementation of a multithreaded sorting algorithm based on John von Neumann's [Knu73] merge sort. The idea to implement multithreaded sorting algorithms is not new but the way this was implemented is a little different. The advantage is that the implementation makes use of the balanced tree structure which a merge sort algorithm creates in memory for the distribution of the sorting problem across different threads. This has the advantage of much less overhead and, as the results in Section 5 show, this approach provides a very fast general-purpose sorting algorithm. Furthermore, Section 3.4.1 explains some changes that make it possible to obtain an algorithm that has optimal speedup with very little initialization overhead.

The presented merge sort implementation outperforms quicksort at sorting large arrays even if just one thread is used.

Even if this does not directly affect the resulting polygonal mesh, it greatly improves the user experience because it speeds up the conversion process massively.

1.4.2 Postprocessing

The approach from Bolitho et al. [BKBH07] is based on the work "Poisson Surface Reconstruction" from Kazhdan et al. [KBH06]. Poisson surface reconstruction tries to create watertight triangulated meshes. This is a big advantage over many other surface reconstruction algorithms because the resulting mesh has no holes in regions where the point cloud is undersampled. But as the authors of Poisson surface reconstruction [KBH06] already noted, this behavior can lead to wrongly connected regions for undersampled areas. Point clouds generated by a scan of a terrain represent only a height field and not a closed surface. For such terrain scans, Poisson surface reconstruction [KBH06] creates meshes with a rough approximation of a hemisphere (half sphere) over the reconstructed terrain surface. This rough hemisphere consists of some very large triangles that hide the actual important surface. With a mesh editing software an experienced user can easily remove those wrongly created triangles, but this forces the user to one more inconvenient step during the conversion process. Therefore, a postprocessing step that can automatically remove such disturbing triangles would be beneficial.

Chapter 4 provides a discussion about two different approaches that were developed to solve this issue. Tests showed that none of them are able to provide a perfect result but one delivers acceptable results.

CHAPTER 2

Previous Work

2.1 Surface reconstruction

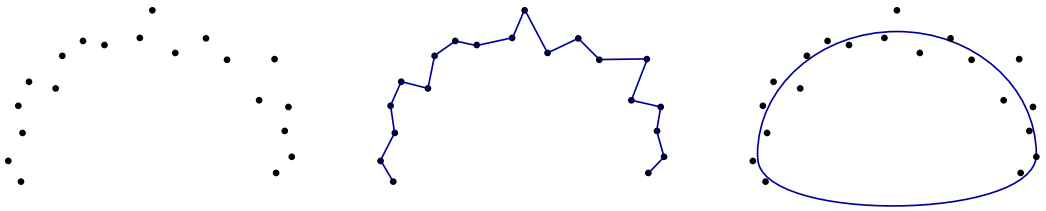
Surface reconstruction methods try to reconstruct the unknown surface $\mathcal{S}$ from a point cloud $\mathcal{P}$. The points $p_i \in \mathcal{P}$ represent discreet samples of the surface $\mathcal{S}$. Most such methods output a triangle mesh that represents the surface $\mathcal{S}$. There are also methods like the *Moving least squares* that resample the point cloud and output a so called *point set surface* [BTS+14]. A survey on the MLS method can be found in [CWL+08]. Because the Scanopy plugin should convert point clouds to triangle meshes only approaches that output the surface S as a polygonal mesh are of interest in the context of this work.

A lot of surface reconstruction approaches were developed in the past two decades. The approaches often address different imperfections of the input point cloud or make assumptions on the shape and topology of the scanned data. Some approaches are developed for a special acquisition method because they use additional scanner information such as RGB, confidence measures or oriented normals.

In general, surface reconstruction approaches can be classified by the form in which the reconstructed surface is represented: Explicit methods that interpolate the Surface by connecting sample points to polygons and implicit methods that approximate the Surface based on the input point set. Figure 2.1 illustrates how differently explicit methods (Subfigure 2.1b) and implicit methods (Subfigure 2.1c) behave for the same input data set (Subfigure 2.1a). The next two sections contain a bit more details about these two methods and Table 2.1 contains a comparison with the most important differences.

2.1.1 Explicit surface reconstruction methods

The most common explicit surface reconstruction methods are Delaunay-based approaches. They first compute a Delaunay triangulation or a dual Voronoi diagram. The resulting cells are then used to define the connectivity between the input point samples. A short



(a) Input (b) Interpolate (Explicit) (c) Approximate (Implicit)

Figure 2.1: Intuitive 2D illustration of the difference between an interpolated and approximated surface reconstruction

survey on Delaunay triangulation based surface reconstruction can be found in [CGY04]. Explicit reconstruction methods typically create triangle meshes containing all or at least most of the points from the input point cloud. The biggest advantage of explicit methods is that they only require the positional information from scanned data. They do not need normal vectors or any other additional information. Another advantage of these methods is the precision of the reconstructed mesh, they can even make provable guarantees on the geometric quality of the reconstructed surface [CGY04]. The computation of interpolating algorithms is often done only on a local subset of the entire Point cloud. Therefore, an easy and efficient parallelization is possible. However, explicit methods require high quality point clouds as input in order to deliver good results [BTS+14]. Therefore, the point cloud should be uniform and densely sampled everywhere, because otherwise the resulting mesh will contain holes. Furthermore, the point cloud should not contain noise or outliers because interpolation strategies are not able to remove them. Therefore, the output surface will have lots of creases where noise was present in the input point cloud and outliers can result in wrongly added and connected polygons. In general, explicit methods do not generate manifold meshes[1], the rather generate topological complex meshes. Scans of the real world can in general not fulfill the requirements explicit methods have to the input data [BTS+14]. The creases in the output mesh that is created by the noise would also make the resulting polygonal mesh unnecessary large, which makes the rendering or any further processing of meshes created from point clouds with hundreds of millions of points more sophisticated. Therefore, explicit methods do not fulfill the requirements of the Scanopy plugin that should be developed.

2.1.2 Implicit surface reconstruction methods

Implicit surface reconstruction methods fit much better to the requirements, because they can generally deal much better with challenging imperfections of point clouds. The characteristic property of implicit methods is the representation of the intermediate result as an implicit function. The computation of this implicit function is the key challenge

[1] A mesh is called manifold if each edge belongs to only one or two faces.

for implicit methods. The implicit function can be a *signed distance field*[2] [HDD+92] or an *indicator function*[3] [Kaz05] and is usually sampled on a regular 3-dimensional grid [HDD+92, Kaz05] or an octree [KBH06, BKBH07]. The reconstructed surface is extracted from the voxels via iso-contouring (the zero-set for signed distance fields or at the value change for indicator functions).

Implicit methods can build smoothed surfaces from noisy point clouds and therefore can create much smaller triangle meshes. Furthermore, implicit methods create topological simpler meshes (manifold). However, most of such methods require a surface normal vector for every sampled point [BTS+14]. For acquisition methods that can provide normals this is not an issue but calculating surface normals from point clouds that contain only positional information this is a challenging problem, especially if the data contains noise. A simple method that uses a Principal Comonent Analysis (PCA) for calculating an unoriented normal was described in "Surface Reconstruction from Unorganized Points" from Hoppe H. et al. [HDD+92]. They defined a local neighborhood N_p around the point p and compute the covariance matrix of N_p by:

$$C_p = \sum_{q \in N_p} (p - q)(p - q)^\intercal$$

($(p - q)(p - q)^\intercal$ is the outer product of two vectors.) The unoriented normal is then defined by the eigenvector of C_p with the smallest eigenvalue. For reconstruction methods that require an oriented normal, the orientation of the normal vector can be estimated by propagating a normal vector that is initially defined for a start point to neighboring points whose unoriented normal has a similar direction [HDD+92]. This method is not very reliable in the presence of noise. – see [BTS+14]

Reconstruction methods that are able to provide a globally smooth, watertight surface even if the input data is noisy and nonuniform sampled, require a global processing/optimization. This leads to two problems: The size of the point cloud that should be processed is limited by the computers main memory size and it is difficult to parallelize such algorithms.

A comprehensive comparison of different approximating surface reconstruction algorithms can be found in "State of the Art in surface reconstruction from Point Clouds" from Berge M. et al. [BTS+14]. They provide an overview of different point cloud imperfections and discuss which algorithms can deal with the different types of imperfections. They also provide a more detailed description about surface normals and shortly discuss more complex methods for calculating unoriented and oriented normals than the already mentioned very simple methods.

[2] A signed distance function or a oriented distance function is a function that determines the distance of a given point x from the surface. The values are > 0 for points outside the model and < 0 inside. Therefore, it is 0 on the surface.

[3] An *indicator function* or a *characteristic function* is a function that is equal to 1 inside the model and 0 outside of it.

	Advantages	Disadvantages
Interpolating Explicit methods	• more precise / guarantees • easy parallelization for local algorithms • do not need normals at points	• do not remove noise • topological complex meshes
Approximating Implicit methods	• can deal with noisy data and handles outliers • simpler meshes (2-manifold, generic) • watertight meshes	• oversmooth (reconstructed surface is far away from scanned points) • requires a global solution • requires normals at points

Table 2.1: Comparison overview of explicit- and implicit-surface reconstruction methods

2.2 Out-of-core surface reconstruction

As already mentioned the Scanopy plugin for Automatic Conversion of Point Clouds to Triangle Meshes is based on "Multilevel Streaming for Out-of-Core Surface Reconstruction" from Bolitho at al. [BKBH07]. Their approach is an implicit method that uses an octree to sample the indicator function. The reconstructed meshes are watertight and globally smooth. The method can handle non-uniform sampled point clouds and is resilient against imperfections of input data such as noise and outliers. Furthermore, the method uses the same reconstruction scheme as the Poisson surface reconstruction [KBH06] which is known for its stability and reliability [BTS+14].

2.2.1 FFT-based reconstruction

The "Poisson Surface Reconstruction" [KBH06] is based on the idea of "Reconstruction of Solid Models from Oriented Point Sets" from Kathdan [Kaz05]. The method requires as input *oriented points*. An oriented point is a point with information about its position ($p_i \in \mathbb{R}^3$) and an oriented surface normal for that point ($n_i \in \mathbb{R}^3$).

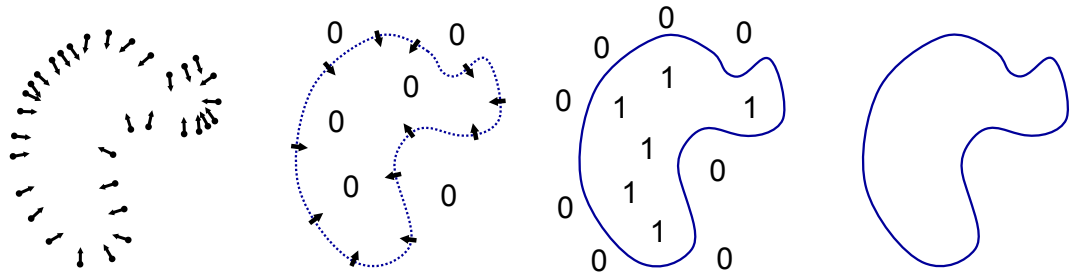
Their idea is to compute the indicator function by computing its Fourier coefficients. *Stokes' Theorem* provides a method for expressing the integral of a function over the interior of a region as an integral over the region's boundary. They used the *Divergence Theorem* or *Gauss's Theorem* which is a specific instance of *Stokes' Theorem* to express the volume integral as a surface integral. This makes it possible to approximate the volume integral of a three-dimensional solid model by integrating a uniformly sampled point set sampled from that model using a Monte-Carlo integration. Since the Fourier coefficients of the characteristic function can be expressed as volume integrals the indicator function can be calculated by applying the inverse Fourier transform to these Fourier coefficients. [Kaz05]

In practice, they implement this reconstruction of the indicator function by "splatting" the sample normals into a voxel grid (where each voxel stores a vector-3), this grid can be seen as a vector field which points in the direction of the surface normals at points on the surface and is zero everywhere else. Convolving the voxel grid with the integration filter results in a function that is like the indicator function but only defined up to an additive constant. Therefore, the function is constant almost everywhere, with a sharp change in value at the sample points so that all points inside the model have the same constant value c_i and all points outside have the same constant value c_o, with $c_i \neq c_o$. If the normals are facing inward the values inside the model are larger than those outside the model ($c_i > c_o$). Finally, the reconstructed surface is extracted as an iso-surface of the voxel grid. [Kaz05] In order to allow non uniformly sampled point clouds they used a simple heuristic method that scales the normal vectors of a sample by a weight which represents the reciprocal sampling density in the region of the sample.

This method is able to reconstruct watertight meshes from noisy, non-uniform sampled point clouds. The biggest limitations of this method is the maximum resolution r of the reconstructed mesh because it is limited to the used voxel grid size. Hence, the grid has a voxel count of r^3 and each voxel has to store multiple floats, the grid will fill up the memory of common computer hardware even for relative low resolutions.

2.2.2 Poisson surface reconstruction

The "Poisson Surface Reconstruction" [KBH06] is like the *FFT-based reconstruction* [Kaz05] a global optimization approach that considers all points from the point cloud at once. The authors of "Poisson Surface Reconstruction" [KBH06] also used oriented points as input and stated that the oriented point samples (Subfigure 2.2a) can be viewed as samples of the gradient (Subfigure 2.2b) of the model's indicator function (Subfigure 2.2c). Therefore, the indicator function can be computed by inverting the gradient operator. They showed that the operation that calculates the indicator function from its gradient can be expressed as a Poisson problem. The advantage of the Poisson problem is that it can be solved from a hierarchy of *locally supported* function. Hence, an accurate solution



(a) Oriented points $\vec{V}$ (b) Indicator gradient $\nabla\chi_M$ (c) Indicator function χ_M (d) Surface ∂M

Figure 2.2: Intuitive illustration of Poisson reconstruction in 2D. (Taken from [KBH06])

is only required close to the reconstructed surface adaptive Poisson solvers can be used to solve this sparse linear system. This allows the authors to replace the voxel grid that was used by [Kaz05] with an adaptive octree that represents the implicit function. Furthermore, the octree can be used to solve the Poisson system.

The resolution r of the reconstructed mesh is controlled by the maximal depth of the octree. Some examples from Kazhdan et at. [KBH06] demonstrate that the memory consumption of the adaptive octree behave approximately quadratic to the reconstruction resolution. The memory consumption of the voxel grid used by the *FFT-based reconstruction* approach [Kaz05] instead is proportional to r^3. Therefore, the "Poisson Surface Reconstruction" [KBH06] scales much better to higher resolutions and can therefore provide much more detailed triangle meshes.

In order to allow non-uniformly sampled point clouds they improved the method that scales normal vectors of a sample used by [Kaz05] in a way that allows sharper features in regions with high sampling density.

To obtain the final mesh from the reconstructed indicator function the iso-surface of the indicator function must be extracted. For this purpose an adapted Marching Cubes [LC87] method that can handle the nonconforming properties of the octree is used. Figure 2.2 illustrates all processing steps performed by the "Poisson Surface Reconstruction" [KBH06] for a simple 2D point cloud. It starts with the input data set (Subfigure 2.2a) and the interpretation of the point normals as gradient of the indicator function (Subfigure 2.2b). Next, the indicator function (Subfigure 2.2c) is calculated from the gradient and finally the surface 2.2d is created by an iso-surface extraction from the indicator function.

2.2.3 Multilevel streaming for out-of-core surface reconstruction

"Multilevel Streaming for Out-of-Core Surface Reconstruction" from Bolitho et al. [BKBH07] is able to reconstruct meshes from point clouds with billions of points. Instead of splitting the point cloud, solving local problems and merging the result by methods like blending the approach from Bolitho et al. [BKBH07] solves the problem globally which makes it more resilient to imperfect input data and allows globally smooth watertight results. Furthermore, the method is able to preserve fine details of such huge point clouds which requires resolutions that are not feasible with the method of [KBH06] because the resulting complexity would exceeds the memory available in standard computer hardware.

Bolitho et al. [BKBH07] also compute the indicator function by using an adaptive octree to represent the data and solve the Poisson equation like described in the work of [KBH06]. However, they enhance the approach of [KBH06] and allow the octree to be out-of-core. This makes it possible to use higher octree depth and therefore obtain resulting surfaces with higher resolutions. In order to allow the octree to be out-of-core and still ensure fast data access without disk IO at random addresses which would make the reconstruction incredibly slow the octree is represented by multiple data streams. For each level of the octree one stream is created. Therefore, an octree with the height h is stored in h different streams on disk. The stream for the level d contains all nodes that

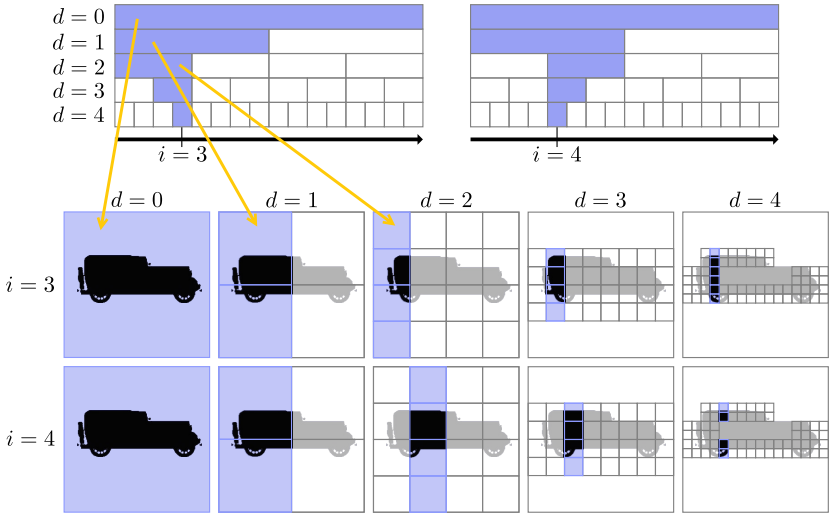


Figure 2.3: Illustration of the multilevel stream structure (top row) and the corresponding quadtree nodes (bottom rows) at two moments in time ($i = 3, 4$). In-core blocks and nodes are highlighted in blue. (Adapted from [BKBH07])

the octree has in level d. The nodes within one stream are grouped into blocks. Each of these blocks contain all nodes with the same x coordinate. Therefore, the stream for the level d is partitioned into 2^d blocks. In order to allow the files to be streamed and not accessed randomly the blocks inside a stream are sorted by the x coordinate. Figure 2.3 illustrates this multilevel streaming approach for a 2D quadtree.

Bolitho et al. [BKBH07] showed that the Poisson equation can be reduced to the sparse symmetric system $Lx = b$ and that a single pass of an Gauss-Seidel solver is able to produce a solution with adequate accuracy for surface reconstruction. This observation allows them [BKBH07] to decompose the "Poisson Surface Reconstruction" [KBH06] into only three streaming passes over the multilevel stream data representation. Each pass traverses the octree by scrubbing all streams in parallel along the x-axis. Because only local data of each stream is required for computation only a small set of nodes near the current x position need to be in-core. During a streaming pass nodes in lower level streams stay longer in memory than nodes in streams for higher levels. Therefore, streams for higher level need to be read and written faster than streams for lower levels of the octree. This behavior is also observable from the changes of the in-core nodes (blue) between the left and right stream states in the top row of Figure 2.3.

This approach not only allows the matrix L to be larger than the main memory but also the vectors b and x can exceed the main memory. The size of the in-core nodes of the octree scales approximately linear to the resolution of the reconstructed surface. [BKBH07]

2.3 Sorting

General purpose sorting algorithms are used to bring elements of an array in a certain order [CGD11]. The main difference between sorting algorithms is their runtime. Another important characteristic of a sorting algorithm is if it is stable or not. A sorting algorithm is stable if elements with equal keys do not change the relative order during the sorting [CGD11].

2.3.1 Insertion sort

Insertion sort is a very simple sorting algorithm with a runtime of $O(n^2)$. It is not efficient on large arrays but for short sequences or already sorted arrays ($O(n)$ for already sorted arrays) it is often faster than more complex algorithms such as quicksort and merge sort. Furthermore, it is simple to implement and provides stable results. [CGD11]

2.3.2 Merge sort

Merge sort was presented by John von Neumann in 1945 [Knu73]. It is a stable, divide and conquer algorithm with a runtime of $(O(n \cdot log_2(n)))$. Important about merge sort is its worst-case runtime because it is only $O(n \cdot log_2(n))$ and the fact that it can efficiently sort data that can only be accessed sequentially [CGD11]. (more details follow in Section 3.3.1)

2.3.3 Quicksort

Quicksort was invented by Hoare [Hoa62]. It has an average runtime of $O(n \cdot log_2(n))$ but can have a worst case runtime of $O(n^2)$. However, if it is implemented correctly this worst case runtime should occur only in very rare cases [CGD11]. Canaan et al. [CGD11] state that Quicksort is typically significantly faster in practice than other $O(n \cdot log_2(n))$ algorithms. In general and especially for performance optimized implementations quicksort is not stable. The main steps executed by quicksort during sorting:

1. Pick an element from the array (often randomly). This element is called the pivot.
2. Move the elements of the array in a way that all elements with a smaller value than the pivot come before the pivot and all elements with a bigger value comes after the pivot. Note: the pivot is now at its final position.
3. Recursively sort the subsequences left and right from the pivot.

A more comprehensive list and more detailed discussion about popular sorting algorithms can be found in [CGD11] from Canaan et al..

CHAPTER 3

Merge Sort

3.1 Why is fast sorting important?

In order to be able to calculate the surfaces of arbitrary point clouds, the algorithm described by Bolitho et al. [BKBH07] requires a preprocessing. During the preprocessing the points of the point cloud are uniformly translated and scaled so that the whole point cloud fits into a unit cube. Further, the points will be rotated with the intention to align the dominant axis of the covariance matrix with the x-axis. This is done in order to reduce the size of the in-core octree to a minimum during the reconstruction. The transformation matrix that defines the required rotation and scaling are calculated within three streaming passes over the point cloud file. In the first pass the centroid is calculated, in the second pass the covariance matrix and in the third pass the bounding box of the point cloud is computed. The point cloud does not have to fit into the main memory for these calculations, because the computation is implemented in three streaming operation.

Subsequently, in one more streaming pass all points of the input point cloud are transformed by the transformation matrix. The transformed points are segmented into equally long partitions along the x-axis, in order to produce subsets $\mathcal{P}_i \subset \mathcal{P}$. Then, for every subset $\mathcal{P}_i$ a file is created and opened for writing before the transformation starts. This makes it possible to perform the transformation and partitioning of the point cloud in one singe input-multiple output streaming pass.

Every $\mathcal{P}_i$ needs to be small enough to fit into the main memory, because the points within the individual subsets $\mathcal{P}_i$ need to be sorted. The $\mathcal{P}_i$ are created by splitting the x-axis into equally long partitions, therefore, the size of $\mathcal{P}_i$ can vary strongly between different $\mathcal{P}_i$ because it depends on the count of points that have a x-value within the x-range of a specific $\mathcal{P}_i$. Therefore, the partition length must be chosen short enough that even the largest $\mathcal{P}_i$ fits into the main memory.

During the sorting all files containing the individual $\mathcal{P}_i$ are read into the main memory, then the points of the $\mathcal{P}_i$ are sorted and finally the points are written back to the disk. This sorting occupies the most time of the preprocessing. Therefore, the overall processing time highly depends on the efficiency of the used sorting algorithm.

3.2 Existing quicksort implementation

The sorting algorithm used by the implementation of Bolitho et al. [BKBH] is an optimized version of quicksort. An explanation of the fundamental function of quicksort is stated in Section 2.3.3.

One of the optimizations used in the algorithm was first suggested already back in 1962 by Hoare, who invented the quicksort algorithm. His idea was to use another algorithm suitable for sorting small numbers of items when the partition contains only a few numbers of items [Hoa62]. Sedgewick et al. picked up the idea and used insertion sort for sorting small partitions in order to speed up quicksort [SA78]. This can also reduce the size of required stack memory [MJG16]. But this is not really important in this implementation, because of the next optimization.

The second optimization addresses the problem of a high amount of stack space and overhead, by removing the recursion and using an explicit stack [SA78].
Additionally, there is an optimized pivot selection that tries to produce two equally long subsequences left and right of the pivot element, no matter how the input data is constituted (i.e. if the input data is already sorted). This works by estimating the median of the subsequence based on 3 samples of the subsequence. This procedure is called “Median-of-Three Modification” and was already mentioned by Hoare in his original paper about quicksort. The implementation of “Median-of-Three Modification” used by Bolito for the StreamingReconsructor [BKBH] takes the first, middle and last element of the subsequence as samples based on which the median is estimated. These specific indices were already used by Singelton in 1969 [Sin69]. A detailed discussion about “Median-of-Three Modification” can also be found in “Implementing Quicksort Program” by Sedgewick et al. [SA78].

The algorithms 3.4 and 3.5 show the above mentioned quicksort implementation in the code of the StreamingReconstructors by Bolitho et al. [BKBH].

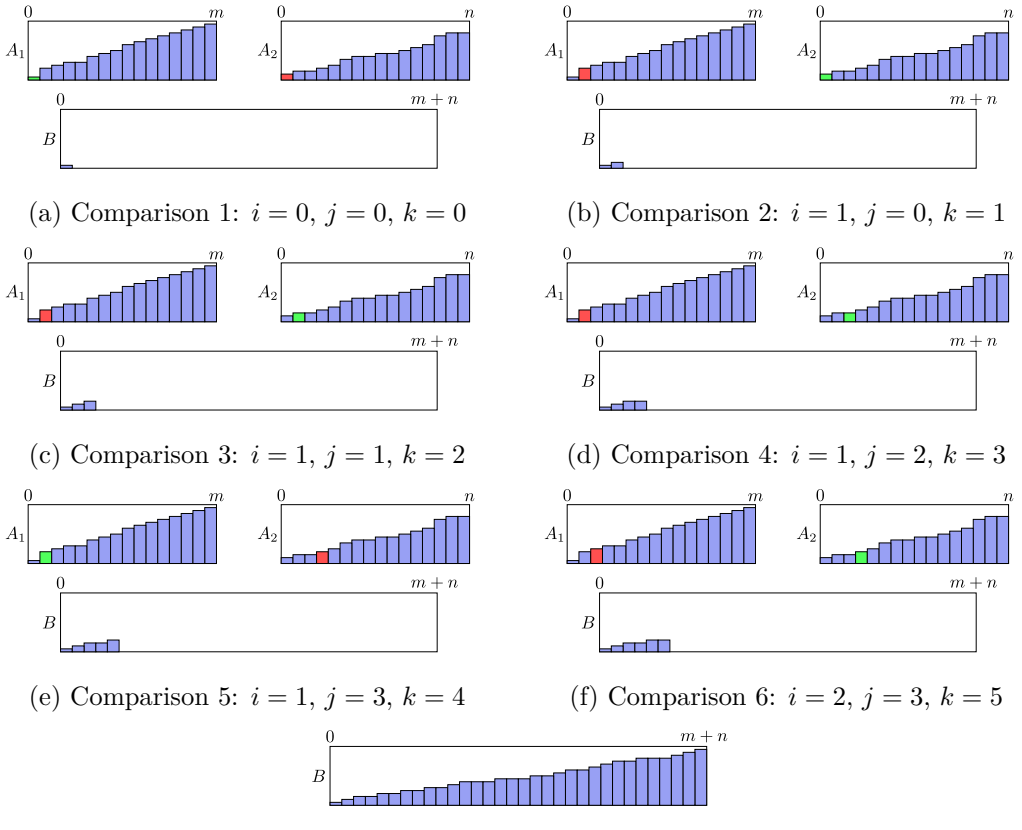
3.3 Multithreaded sorting

This quicksort algorithm uses only one thread and is therefore not able to take advantage of modern multicore systems. So it was natural to implement a multithreaded sorting algorithm that fits exactly the requirements that occur during sorting the point data. The decision which sorting algorithm should be used as a base was much in favor for merge sort, which was first presented by John von Neumann in 1945 [Knu73], because like quicksort it is a general-purpose and comparison-based sorting algorithm with a

runtime of $n \cdot log_2(n)$ [CGD11]. Most importantly, the parallelization of different merge operations is relatively straightforward.

3.3.1 Explanation of merge sort

In order to understand why parallelization of the different merge operations was so obvious from the author's point of view, it is first necessary to understand how a top-down implementation of merge sort works. Merge sort is a so-called divide and conquer algorithm. This means that the algorithm divides the data into two equal subsequences recursively until only subsequences containing one element are left. Each sequence with only one element is always a correct ordered sequence. Afterwards, in each case two ordered subsequences are merged. The merge operation takes two already sorted subsequences as input. It compares the elements located farthest to the left in both sequences which are not already used. The smaller element of those two is written to the next free index in the temporary output array B. This process continues until one of the



(a) Comparison 1: $i = 0$, $j = 0$, $k = 0$

(b) Comparison 2: $i = 1$, $j = 0$, $k = 1$

(c) Comparison 3: $i = 1$, $j = 1$, $k = 2$

(d) Comparison 4: $i = 1$, $j = 2$, $k = 3$

(e) Comparison 5: $i = 1$, $j = 3$, $k = 4$

(f) Comparison 6: $i = 2$, $j = 3$, $k = 5$

(g) Final result

Figure 3.1: Illustration of merge; first 6 comparison for $n = m = 16$ and the final merged sequence;

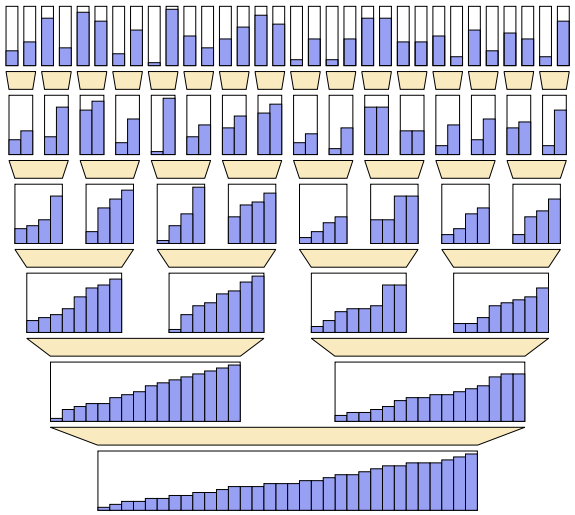


Figure 3.2: Illustration of merge sort

input subsequences is exhausted. As soon as one of the input subsequences is exhausted the remaining elements of the other input subsequence are copied into the temporary array B past the last already written element. Afterwards, all ordered elements of the temporary array B need to be copied back to the array A at the indices where the two input sequences have been located. Following to the completion of the two merge operations of one recursive divide, both subsequences from those two merge operations are merged as well. All subsequences are merged until the original sequence which has been previously divided is merged and is therefore ordered correctly.

This merge operation is very efficient due to the fact that only the first unused elements of each subsequence are compared. Since both subsequences are already in the correct order, one of the first elements needs to be the smallest element. This described merge algorithm is only one of many possibilities to implement a merge. Figure 3.1 illustrates the first six iterations of a merge operation. Algorithm 3.1 shows an implementation of merge which gets along with fewer checks for index boundaries and therefore is more efficient than the above described merge algorithm. However, both algorithms need a temporary array B with the same size as A.

Algorithm 3.2 shows a recursive implementation of merge sort and Figure 3.2 provides a visualization of that algorithm for an input array with 32 elements.

3.3.2 Comparison between quicksort and merge sort

Merge sort divides all sequences into subsequences with equal size. Quicksort on the other hand has a very high probability to divide the sequence in subsequences with unequal size due to the fact that it divides the sequence into subsequences based on the final position of the so called pivot element. The selection of the pivot element depends

Algorithm 3.1: Merge (Taken from [Rai09])

Input: sorted subsets Array A[l],...,A[m], A[m + 1],...,A[r] and temporary array B
Parameters: sorting begin middle and end indices l, m, r
Output: merged, sorted subset A[l],...,A[r]

```
1  procedure merge(integer l, integer m, integer r)
2      B[l,...,m] ← A[l,...,m]
3      B[m + 1,...,r] ← A[r,...,m + 1] // ordered upside down!
4      p ← l
5      q ← r
6      foreach i = l, ..., r do
7          if B[p] ≤ A[q] then
8              A[i] ← B[p]
9              p ← p + 1
10         else
11             A[i] ← B[q]
12             q ← q − 1
13         end
14     end
```

Algorithm 3.2: Mergesort (Taken from [Rai09])

Input: unsorted Array A and temporary array B
Parameters: sorting begin and end indices l, r
Output: sorted subset A[l],...,A[r]

```
1  procedure mergesort(integer l, integer r)
2      if l < r then
3          m ← ⌊(l+r)/2⌋
4          mergesort (l, m)
5          mergesort (m + 1, r)
6          merge (l, m, r)
7      end
```

on the chosen implementation of the particular quicksort. However, the fact that the final position of the pivot element is not predictable is common to all implementation possibilities (at least for an unsorted input). Therefore, the size of the subsequences on the left and the right site of the pivot element is not predictable either and therefore the two subsequences are in general not of equal size. This behavior makes quicksort less suitable for parallel execution (i.e. multithreaded implementation).

As mentioned above, merge sort always divides all sequences in subsequences with equal size. This leads to the fact that in case of sorting each of these array subsequences with

another thread each thread has to do nearly the same amount of work. Therefore, it is highly likely that both threads finish nearly at the same time, which ensures an optimal resource utilization of the CPU cores. Of course quicksort could be parallelized by sorting the partition on the left and on the right side of the pivot element with two different threads. However, since the partitions have a very high probability to have different sizes, the threads have a high likelihood to finish the sorting of the subsequences in a different amount of time. This means that a lot more threads than available CPU cores would need to be created and synchronized in order to provide the utilization of all CPU cores. The creation and synchronization of more threads than available CPU cores would lead to an unnecessary overhead. The use of thread pools would surely help, although a lot more synchronizations than available CPU cores would be needed and this leads to an unnecessary increase in computing effort and waiting time.

A detailed discussion of the above mentioned approach of parallelizing quicksort is presented by "Parallel quicksort using Thread Pool Pattern" by Somshubra Majumdar, Ishaan Jain und Aruna Gawade [MJG16].

Merge sort accesses the main memory in a more linear way than quicksort does, because the main memory access pattern of quicksort tends to be random. Through the linear memory access of merge sort, all caches of the CPU should be used in an optimal way, which leads to a decrease of waiting time for the data to be loaded into the CPU registers. This has not a big influence on the amount of data which does not exceed the fast CPU memory caches, but it has a huge impact on the total amount of time needed for an amount of data which exceeds the CPU memory caches by hundreds or thousands of times. Since the sorting of points within a point cloud includes a huge amount of data which massively exceeds the size of the CPU caches and the rather optimal use of the CPU caches is yet another argument in favor for the use of merge sort.

Another advantage of merge sort is its ability to produce stable sorts, an ability quicksort does not provide [CGD11]. However, that is of no real interest to our goal, which is sorting point clouds for surface reconstruction. An additional disadvantage of quicksort is its runtime behavior that is dependent on the input data [MJG16].

3.4 The implementation of multithreaded merge sort

The approach chosen by the author to parallelize the merge sort is based on the fact that (also view Algorithm 3.2) the second recursive invoke of merge sort is made within a new thread. This means that the second half of the data to be sorted is sorted within a new thread and, parallel to that, the first half is sorted by the thread which already invoked the actual merge sort method. Figure 3.3 illustrates this approach for four threads.

Of course it does not make any sense to apply this method to every invoke of merge sort because the use of another thread for subsequences with only a small amount of data to be sorted would rather lead to more overhead than it would help to decrease the run

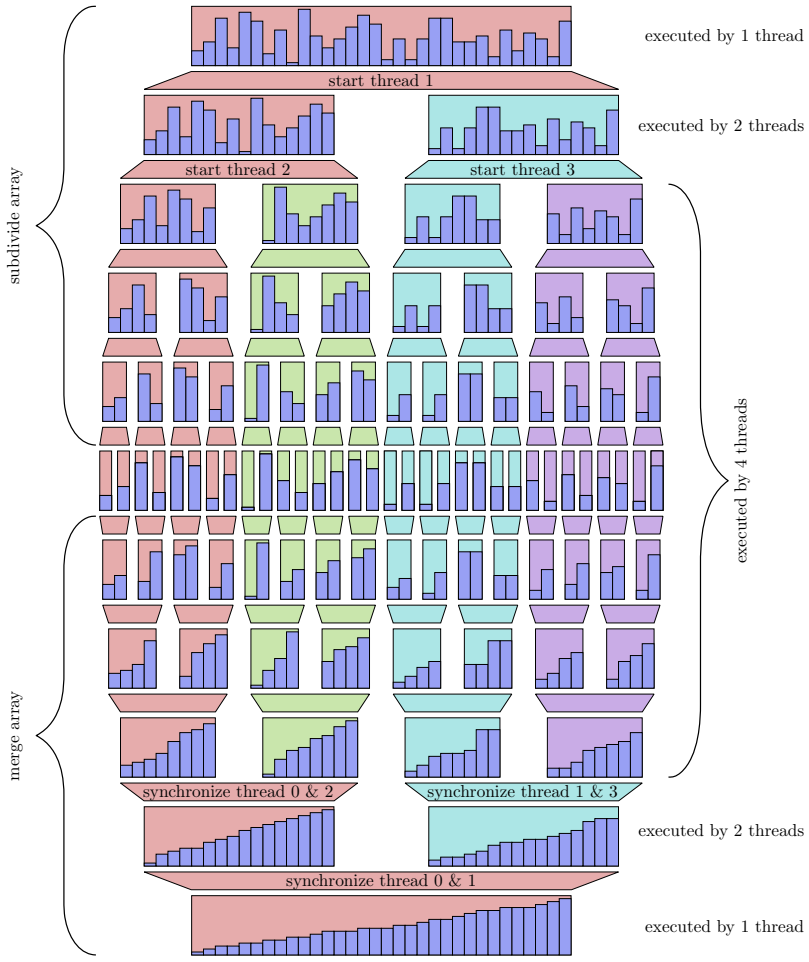


Figure 3.3: Illustration of multithreaded merge sort (4 threads); the color indicates the thread that is executing the visualized operations (thread 0: red, thread 1: turquoise, thread 2: green, thread 3: purple);

time. The most efficient way to use the parallelized version of merge sort is to create only as many threads as CPU cores are available.

Since merge sort recursively divides all data that has to be sorted into two equal halves, an equal and therefore optimal distribution of the required computation to an amount of t threads can be ensured if t is a power of two ($t = 2^x$). In case the amount of available CPU cores p is 2^x as well, all necessary computation can be distributed equally to all available computing resources with a minimum of overhead. To ensure the optimal resource utilization for any amount (unequal 2^x) of p CPU cores, more threads t than available CPU cores p need to be used. The minimum number of threads that should ensure an utilization of all CPU cores during most of the time in that case is $t = 2^{\lceil \log_2(p) \rceil}$. In other words, the amount of CPU cores needs to be rounded up to the next bigger

power of two. After the requested amount of threads is available as a power of two, the merge sort method can check if a new thread should be started which sorts the right subsequence or if both subsequences can be computed by the actual thread. This check can be easily performed against the recursion depth d. In order to do that, it needs to be evaluated if $2^{d+1} \leq t$ is fulfilled. As long as 2^{d+1} is smaller or equal to the amount of requested threads t a new thread will be used to sort the second half. Algorithm 3.9 shows the concept described above whereby the parameters `maxThreads` and `threadStartDepth` meet the variables t and d.

3.4.1 Runtime analysis

The concept of parallelizing only the different merge operations at the top of the call tree has the advantage of relatively little overhead. Also, for the whole sorting process, just t synchronizations are needed. However, it has the disadvantage of not being able to parallelize the last $t-1$ merge operations over all available CPU cores p and the last merge operation is done only by one thread. This is due to the fact that the level d with $0 \leq d \leq \lceil \log_2(n) \rceil < h$ of the call tree can only be processed by 2^d different threads, because there are no more merge operations in that level.

The lowest level d that contains t merge operations can be calculated as follows:

$$\begin{aligned} 2^d &= t \\ d &= \log_2(t) \end{aligned}$$

In level $\log_2(t) - 1$ and all levels above there are therefore not enough merge operations to use t threads. Hence, for all merge sort calls where $2^d < t$ is fulfilled not all available CPU cores are used. That occurs exactly at $t-1$ merge operations, because in level $\log_2(t) - 1$ the number of $2^{\log_2(t)-1} = t/2$ merge operations have to be executed. In the level above there are just $(t/2)/2 = t/4$ merge operations left. With every decrease of d the count of merge operations gets divided by 2 until there is only one merge operation left at level 0. One can imagine the recursive merge operation calls like the nodes of binary tree. Therefore, the count of merge operations in and above the levels $\log_2(t) - 1$ can be calculated in the same manner as the count of nodes of a balanced binary tree with height h. (It is important to point out that the levels start with 0. Therefore, the highest possible d in a binary tree with the height h is $d = h - 1$.)

$$\begin{aligned} 2^h - 1 &= 2^{d+1} - 1 \\ &= 2^{(\log_2(t)-1)+1} - 1 \\ &= 2^{\log_2(t)} - 1 \\ &= t - 1 \end{aligned}$$

In every level the threads are doubled, starting with level $d = 0$ having one thread, level $d = 1$ having two threads, and so on, until the level $d = \log_2(t)$, where all t threads are

used for the first time. This leads to the following runtime for the first $d < \log_2(t)$ levels:

$$\sum_{i=0}^{\log_2(t)-1} \left(\frac{n}{2^i}\right) = \frac{2n(t-1)}{t} = 2n - \frac{2n}{t}$$
$$= O(2n)$$

For all further levels the runtime is calculated as follows:

$$\sum_{i=\log_2(t)}^{\log_2(n)} \left(\frac{n}{t}\right) = (\log_2(n) - \log_2(t)) \cdot \frac{n}{t}$$

And therefore the total runtime of the algorithm is:

$$\frac{2n(t-1)}{t} + (\log_2(n) - \log_2(t)) \cdot \frac{n}{t} = \frac{n}{t}(2t - 2 + \log_2(n) - \log_2(t))$$

The speedup of a parallel algorithm is calculated by $\text{Speedup}(t) = \Theta_{\text{seq}}(n)/\Theta_{\text{par}}(t,n)$ where $\Theta_{\text{seq}}(n)$ is the runtime of the best sequential algorithm for solving a problem and $\Theta_{\text{par}}(t,n)$ is the time taken by the parallel algorithm to solve the same problem using t threads. The best possible, absolute speedup is linear in t. An optimal parallelization of the merge sort would lead to a runtime of $n \cdot \log_2(n)/t$, because a sequential implementation of the merge sort algorithm has a runtime of $n \cdot \log_2(n)$ [CGD11]. The speedup of the described algorithm at the merge operations where $d \geq \log_2(t)$ is calculated by:

$$\frac{(\log_2(n) - \log_2(t)) \cdot n}{(\log_2(n) - \log_2(t)) \cdot \frac{n}{t}} = \frac{1}{\frac{1}{t}} = t$$

That means, for merge operations where $d \geq \log_2(t)$ is true the speedup is t. It is therefore linear to the count of the used threads and, hence, the best possible speedup. The algorithm is less optimal for merge operations where $d < \log_2(t)$ is true:

$$\frac{(\log_2(t) - 1) \cdot n}{\frac{2n(t-1)}{t}} = \frac{t \cdot (\log_2(t) - 1)}{2 \cdot (t-1)}$$
$$= \Theta\left(\frac{t \cdot \log_2(t)}{t}\right) = \Theta(\log_2(t))$$

Therefore, the described parallel merge sort implementation has a total speedup of:

$$\frac{n \cdot \log_2(n)}{\frac{n}{t}(2t - 2 + \log_2(n) - \log_2(t))} = \frac{t \cdot \log_2(n)}{2t - 2 + \log_2(n) - \log_2(t)}$$

For very large n, the whole algorithm has a speedup of approximately t, because:

$$\lim_{n\to\infty} \frac{t \cdot \log_2(n)}{2t - 2 + \log_2(n) - \log_2(t)} = t$$

However, the smaller n gets the less optimal the speedup gets. It is particularly adverse, if additionally a large amount of threads t comes together with a small n. Here is an example where 10^5 elements are to be sorted by 32 threads. For modern workstations 32 CPU cores could be a realistic number. Now, if $n = 10^5$ and $t = 32$ are put into the equation for the total speedup of the algorithm, the result is:

$$\frac{t \cdot \log_2(n)}{2t - 2 + \log_2(n) - \log_2(t)} = \frac{32 \cdot \log_2(10^5)}{2 \cdot 10^5 - 2 + \log_2(10^5) - \log_2(32)} \approx 7.22064$$

However, a speedup of ≈ 7.2 is far from optimal, because the optimal speedup for this example would be 32. The larger t gets in relation to n the more influence merge operations where $d < \log_2(t)$ have on the total runtime.
The speedup of the theoretical worst case (t tends to ∞):

$$\lim_{t\to\infty} \frac{t \cdot \log_2(n)}{2t - 2 + \log_2(n) - \log_2(t)} = \frac{\log_2(n)}{2}$$

Use co-rank for an optimal speedup

In the following, a method is described that makes it possible to use all available CPU cores for the last $t - 1$ merge operation. In order to do that, the merge operation itself has to be parallelized, if the condition $2^d \leq t$ is fulfilled at a call of merge sort. While this method was out of scope for this bachelor thesis and was therefore not implemented by the author, it is still discussed here for completeness.

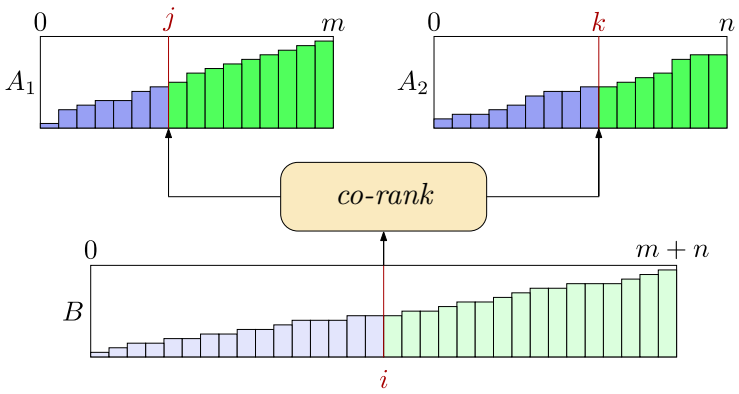


Figure 3.4: Illustration of *co-rank*; the *co-rank* defines the indices j and k in A_1 and A_2 for any given index i (rank) in B before merging A_1 and A_2 into B. (Adapted from [ST13])

During one merge operation two already sorted subsequences will be merged to one newly sorted subsequence which contains all elements of the two already sorted input subsequences. The author uses the "[x]" notation within the following text to indicate indices of one array as it is common in the programming language C. Further, all subsequences of the arrays start with index 0 for an improved understanding. At a concrete implementation of merge sort the indices would not start with 0 since the starting index of the particular subsequence would need to be added to the indices shown in this section.

For an improved understanding the two sorted input subsequences are named subsequence A_1 and A_2 with an amount of m or n elements. Since A_1 and A_2 are sorted ascendingly the following applies: $A_1[j-1] \leq A_1[j]$ for $1 \leq j < m$, and $A_2[k-1] \leq A_1[k]$ for $1 \leq k < n$. The output of the merge operation will be named B, analogous to the previous described temporary array which is always required by merge sort. The output subsequence B is also required to be sorted, therefore: $B[i-1] \leq B[i]$ for $1 \leq i < m+n$ must hold.

In order to guarantee an optimal utilization of all threads t which are involved at the merge operation, each thread should write $\frac{m+n}{t}$ element in B. The calculation for the necessary beginning and ending indices for each thread within the not yet calculated output array B is simple. If r is a unique ID for each thread and $0 \leq r < t$ applies, the start index i_{start} at which the thread with the ID r starts to write within B can be calculated as follows: $i_{\text{start}} = \lfloor \frac{m+n}{t} \cdot r \rfloor$. Analogous to that the last index i_{end} at which the thread r writes, can be calculated as follows: $i_{\text{end}} = \lfloor \frac{m+n}{t} \cdot (r+1) \rfloor - 1$.

Of course the beginning and ending indices of the input subsequences B_1 and B_2 will be also needed. The calculation of those two is not as easy as the calculation of the beginning and ending indices described above. A very time-consuming additional calculation effort seems to be required, at least at first sight. For each index i within B, also called *rank*, must be available either an index j within A_1 or an index k within A_2, at which the same element is given. The indices j and k, at which the same element as in $B[i]$ must be stored, are called *co-rank*. Figure 3.4 provides a visual definition for the *co-rank*.

With [ST13], Christian Siebert and Jesper Larsson Träff have introduced an algorithm in 2013 which resolves the mentioned problem very efficiently and elegantly. Since the algorithm calculates the so called *co-rank* they just named it *co-rank*.

The basic idea for co-rank was first introduced in [AS87], but it was only used for solving one special case where the rank i is the median of B. The authors of [ST13] found a way to generalize that idea. Furthermore they developed a simple algorithm that solves any merging problem in a stable way and provides an intuitive proof for it.

The base for the *co-rank* algorithm is the Lemma 1 developed and written down by Christian Siebert and Jesper Larsson Träff in [ST13].

Lemma 1 *For any $i, 0 \leq i < m + n$, there exists a unique $j, 0 \leq j \leq m$, and a unique $k, 0 \leq k \leq n$, with $j + k = i$ such that*

1. $j = 0 \vee A_1[j-1] \leq A_2[k]$ *and*
2. $k = 0 \vee A_2[k-1] < A_1[j]$

A proof for Lemma 1 can be found in [ST13].

The *co-rank* algorithm starts with the extreme assumption that all i elements that have to be written into B are coming from the Array A_1, at least as far as possible: $j = \min(i, m)$ (because A_1 has only m elements, the algorithm can only copy m elements from A_1 to B). In order to fulfill the equation $j + k = i$ the index k must therefore be initiated through $k = i - j$. Furthermore a lower bound for j is defined through j_{low}. In the case that $i > n$, the index j_{low} can be defined larger than 0, because if the amount of elements in A_2 is smaller than i at least $i - n$ have to come from A_1. Otherwise there are not enough elements to fill up B until the index i. Therefore, j_{low} is initialized through: $j_{\text{low}} = \max(0, i - n)$. The requested index for A_1 is therefore located between j and j_{low}. Furthermore the initialization of j_{low} through $\max(0, i - n)$ is important because it ensures that k is not larger or equal to n after the first iteration.

After the initiation, the algorithm starts a binary search. Therefore, a loop will be repeated over and over again until both conditions from Lemma 1 are fulfilled. During the entire search, the algorithm ensures that the equation $y + k = i$ is satisfied. In order to perform the binary search, j_{low} and k_{low} will be updated after each iteration in addition to j and k. At any time during this process, $j_{low} \leq j$ and $k_{low} \leq k$ must be satisfied. Further, it is necessary that either the searched index of A_1 is located within the interval j_{low} and j or the searched index of A_2 is located within the interval k_{low} and k.

Within the first loop iteration either the first Lemma condition is violated, this means $A_1[j-1] > A_2[k]$ or the extreme assumption $j = min(i, m)$ is correct and the *co-rank* was already found throughout the initialization. If the second Lemma condition would be violated, it would mean that $A_2[k-1] \geq A_1[j]$ and therefore that j would be too small. But, this dilemma is not possible since j was initiated with the highest possible value.

If $A_1[j-1] > A_2$ is fulfilled, and therefore the first Lemma condition is violated, the index j is too large. Therefore, j will now be decreased by $\delta = \lceil \frac{j - j_{low}}{2} \rceil$. Further, the lower bound k_{low} can be raised to k since no smaller index than k can fulfill the first Lemma condition. (If there would be an index smaller than k for A_2 that can fulfill the first Lemma condition, this would mean that j is too small, which is in conflict with the violation of the first Lemma condition.) Since j must be reduced by δ, the index k must be increased by δ so that the equation $j + k = i$ still applies.

If now the second Lemma condition is violated, which is the case when $A_2[k-1] \geq A_1[j]$ and therefore k is too large, k must be decreased by $\delta = \lceil \frac{k - k_{low}}{2} \rceil$. Since no index

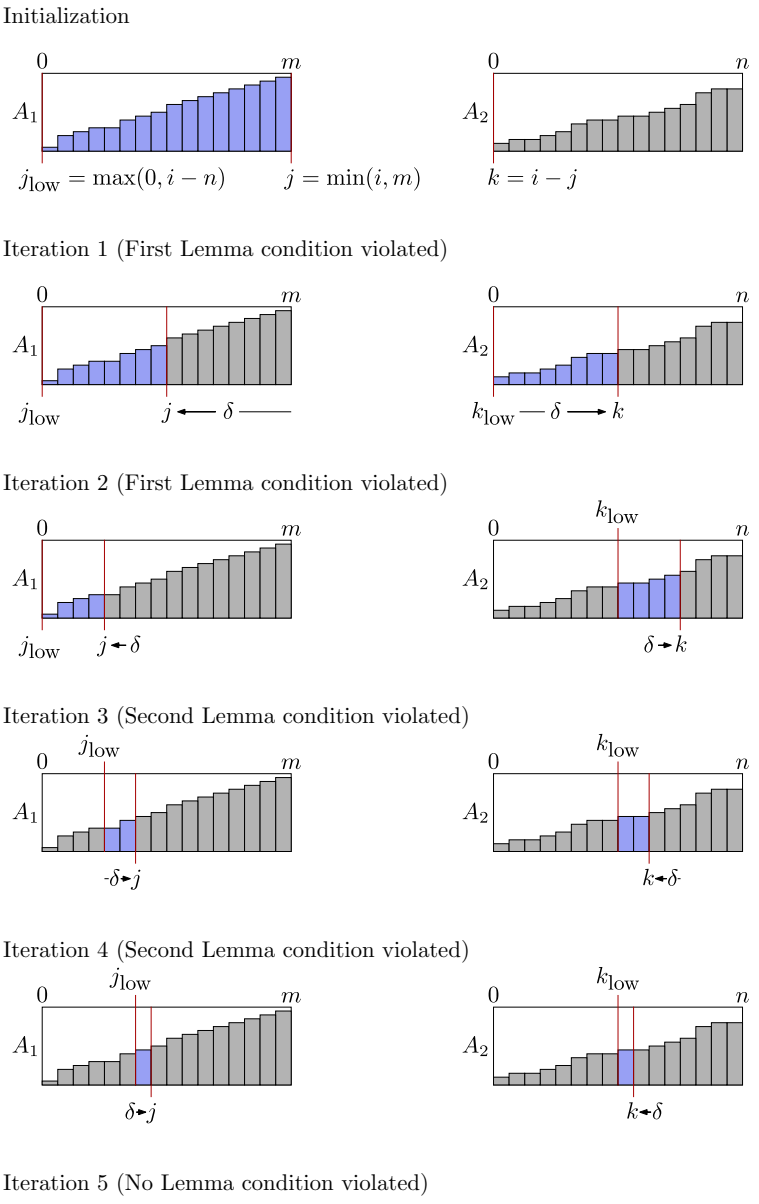


Figure 3.5: Illustration of a *co-rank* algorithm run for $i = 16$ (Adapted from [ST13])

smaller than j can fulfill the first Lemma condition, the lower bound j_{low} can therefore be increased to j. Also, in this case, it must be ensured that $j + k = i$ is fulfilled and therefore j must be increased by δ.

Before the particular Lemma conditions can be verified, the index boarders need to be checked. $j > 0 \wedge k < n$ must be fulfilled before for $A_1[j-1] > A_2$ can be evaluated and $k > 0 \wedge j < m$ must be fulfilled before for $A_2[k-1] \geq A_1[j]$ can be evaluated.

The loop with the above-described checks and calculations will be repeated until none of the two Lemma conditions is violated any more. After the loop execution, both variables k and j contain the searched co-rank and, therefore, the indices of the two input arrays A_1 and A_2 at which the values start that have to be written into the output array B from the index i on. A run of the co-rank algorithm is illustrated in Figure 3.5.

The *co-rank* algorithm has a worst case runtime of $O(\log_2(\min(m, n)))$. A proof for this formula can be found in [ST13].

For each i_{start} and i_{end} the respective indices j_{start}, j_{end} for A_1 and k_{start}, k_{end} for A_2 can now be calculated very efficiently through the *co-rank* algorithm. If the start and end indices for each thread will be calculated separately, and therefore the start index of the thread $r + 1$ is not used as the base for the end index of the thread r, the *co-rank* calculation can be parallelized as well. That means, the whole merge operation, including the calculations for the *co-rank*, can be parallelized without the need for any synchronization. This leads to a total runtime of $O(\frac{n+m}{t} + \log_2(\min(m, n)))$ for the merge operation that is parallelized by t threads [ST13].

Furthermore this parallel merge algorithm has an optimal speedup if the algorithm is executed by t threads on a computing system with $p = t$ processing cores and $t = p \leq \log_2(\min(m, n))$. [ST13]

If one now uses that parallel merge algorithm to solve the problem with the last $t - 1$ merge operations and, therefore, to further decrease the total runtime of merge sort, one should pay attention to the following: Since merge sort calls with a recursion depth of d are already running in parallel with a total number of d^2 merge sort calls, the merge operation itself should not be parallelized by t threads. Instead $\frac{t}{d^2}$ should be used. This is because $\frac{t}{d^2}$ threads within one merge sort call times d^2 parallel executed merge sort methods results in $\frac{t}{d^2} \cdot d^2 = t$ total threads. This should ensure an optimum utilization of all available CPU cores with a minimum overhead for the creation and synchronisation of threads.

This means, further more, that the length of input subsequences m and n for the *co-rank* can never be smaller than $\lfloor \frac{|A\text{ges}|}{2 \cdot t} \rfloor$. Where A_{ges} is the entire array to be sorted. Out of $m = n \geq \lfloor \frac{|A\text{ges}|}{2 \cdot t} \rfloor$ follows again that the total merge sort algorithm would achieve an optimal speedup if t does not exceed $\log_2(\lfloor \frac{|A\text{ges}|}{2 \cdot t} \rfloor)$.

In theory the usage of the described co-rank should lead to an optimal speedup for the whole multithreaded merge sort, but the author did not validate this in an empirical test. All the runtime measurements shown in Chapter 5 are done without the co-rank optimization. That means the code that was used for the test was not able to use all available CPU cores during the last $t - 1$ merge operations.

3.5 Reduce the memory consumption of merge sort

A naive implementation of the merge sort algorithm would allocate the temporary array B with the same size as the array A that should be sorted. However, the merge algorithm can be written in a way so that only the first half of elements need to be written into the temporary array B. The second half of elements can stay in A. The merge algorithm then compares elements in B with elements in the second half of A. This means that B only needs to have half the size of A.

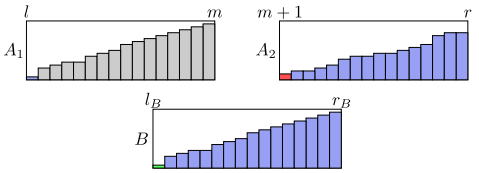
3.5.1 Relevance of the topic

Nowadays, where we have Gigabytes of main memory available, it does not really matter if the algorithm needs 0.5 Mbyte or 1Mbyte of additional temporary space to sort an array with the size of 1 Mbyte. Though if one needs to sort an array with a size of 2 Gigabytes it can make a difference if the algorithm needs 1 or 2 Gigabytes of additional temporary space in the main memory. Especially if the used computer has just 4 Gigabytes of main memory. To give an example let's say the operating system and the program code need 1 Gigabyte and an array with 2 Gigabytes of data that should be sorted is loaded into the memory, then there is only 1 Gigabyte left. In such a scenario, using n or just $\frac{n}{2}$ can make the difference if sorting is possible or not. Furthermore, the smaller an array is the less likely it gets fragmented in main memory and reading continuous lines from main memory is much faster than if the data is fragmented. Therefore, the target was to keep the size of B as small as possible. In order to achieve that goal the author has implemented a merge algorithm as follows.

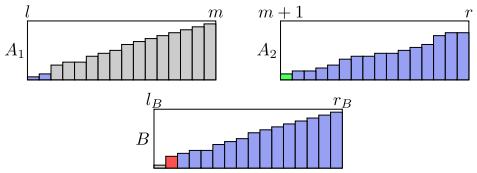
3.5.2 Memory reduced merge algorithm

To ensure a better readability we divide A into two halves and name the first half A_1 for all $A[x]$ with $l \leq x \leq m$ and the second half A_2 for all $A[x]$ with $m < x \leq r$. Thereby, the index m separates the two already sorted input sequences in A. Further, we define the index at which the next final sorted element in A is written as k. As well as the two indices at which the two elements that shall be compared next are located, as j for A_2 and i for B. As mentioned above, all elements from A_1 need to be copied to B at the beginning of the merge algorithm. For a multithreaded merge sort implementation it is crucial that not all merge operations are allowed to start writing at the index 0 in B. Therefore, the first index which is the starting point for writing in B is named l_B.

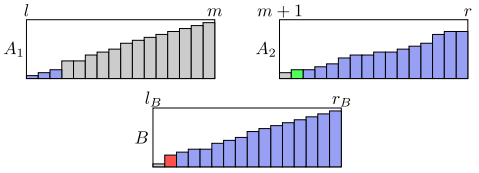
After the elements of A_1 are copied to B, the 3 indices i, j and k are set to the following values: $i = l_B$, which is the first usable index for the current thread in B; $j = m + 1$, which is the first index in the second sortet input subset A_2; $k = l$ which is equivalent to the beginning of array A. Now, a classic merge operation for the input arrays A_2 und B can be executed. A serves as destination for the merged output sequence. Within that operation, the next element which will be placed to $A[k]$ is determined by the comparison $B[i] \leq A[j]$. If $B[i] \leq A[j]$ is fulfilled, the value of $B[i]$ will be written to $A[k]$ and i will



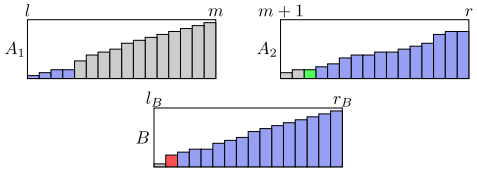
(a) C. 1: $i = l_B + 0$, $j = m + 1$, $k = l + 0$



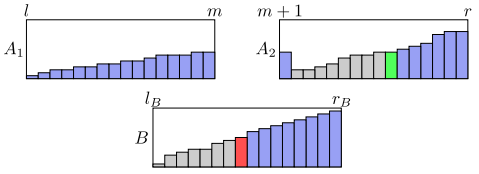
(b) C. 2: $i = l_B + 1$, $j = m + 1$, $k = l + 1$



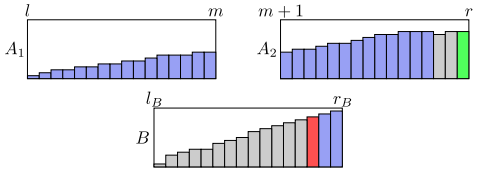
(c) C. 3: $i = l_B + 1$, $j = m + 2$, $k = l + 2$



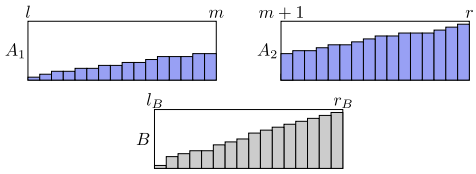
(d) C. 4: $i = l_B + 1$, $j = m + 3$, $k = l + 3$



(e) C. 17: $i = l_B + 7$, $j = m + 10$, $k = l + 16$



(f) C. 29: $i = l_B + 13$, $j = m + 16$, $k = l + 28$



(g) Final result

Figure 3.6: Illustration of memory reduced merge; Subfigures a, b, c and d show the first 4 comparisons. Subfigure e shows the action of the 17th comparison, which is the first one that writes the result to an index which is located in A_2 instead of A_1. Subfigure f shows the 29th comparison, which is the last comparison, because it uses the last Element in A_2. The remaining 3 elements in B are then copied back to A, which results in the final merged array which is shown in Subfigure g. The values drawn in gray are not required any more and can be overwritten.

be increased by 1. In case $B[i] \leq A[j]$ is not fulfilled, the value of $A[j]$ will be written to $A[k]$ and j will be increased by 1. In both cases the index k needs to be increased by 1.

This process will be repeated until either all elements of A_2 are used and therefore copied to a lower index in A ($j > r$ is fulfilled), or when all elements in B are copied back to A to their final position. In the first case, when A_2 is exhausted before B, the remaining elements in B need to be subsequently written to the end of A, starting of course at index k, like always when something is written in A. In the second case, if B is exhausted

before A_2, the merge process is finished without any further operations. Because if B is exhausted, the index k must have catched up to index j, and therefore $k = j$ is valid. This means further that all unchanged elements which are left in A_2 are placed on their correct position, because the distance between k and j is always equal to the amount of elements in B which are not used: $j - k = r_B - i$ with $r_B = l_B + (m - l)$. Furthermore, it is impossible that more elements from B are used than available in B. Therefore, the index k at which is always written into A can never overtake the index j at which is read in A. This ensures that no value in A_2, which is not yet copied to an other position, will be overwritten. Therefore, an error-free functionality of the merge operations which only needs half of the additional temporary space is ensured. The algorithm 3.8 shows a working implementation of merge which only requires a B that is only half as large as A. In the first condition ($k < j$) of the second loop of this algorithm it is also directly observable that it is impossible to overwrite a still needed element in A. Figure 3.6 illustrates the functionality of this memory reduced merge algorithm – also compare with Figure 3.1.

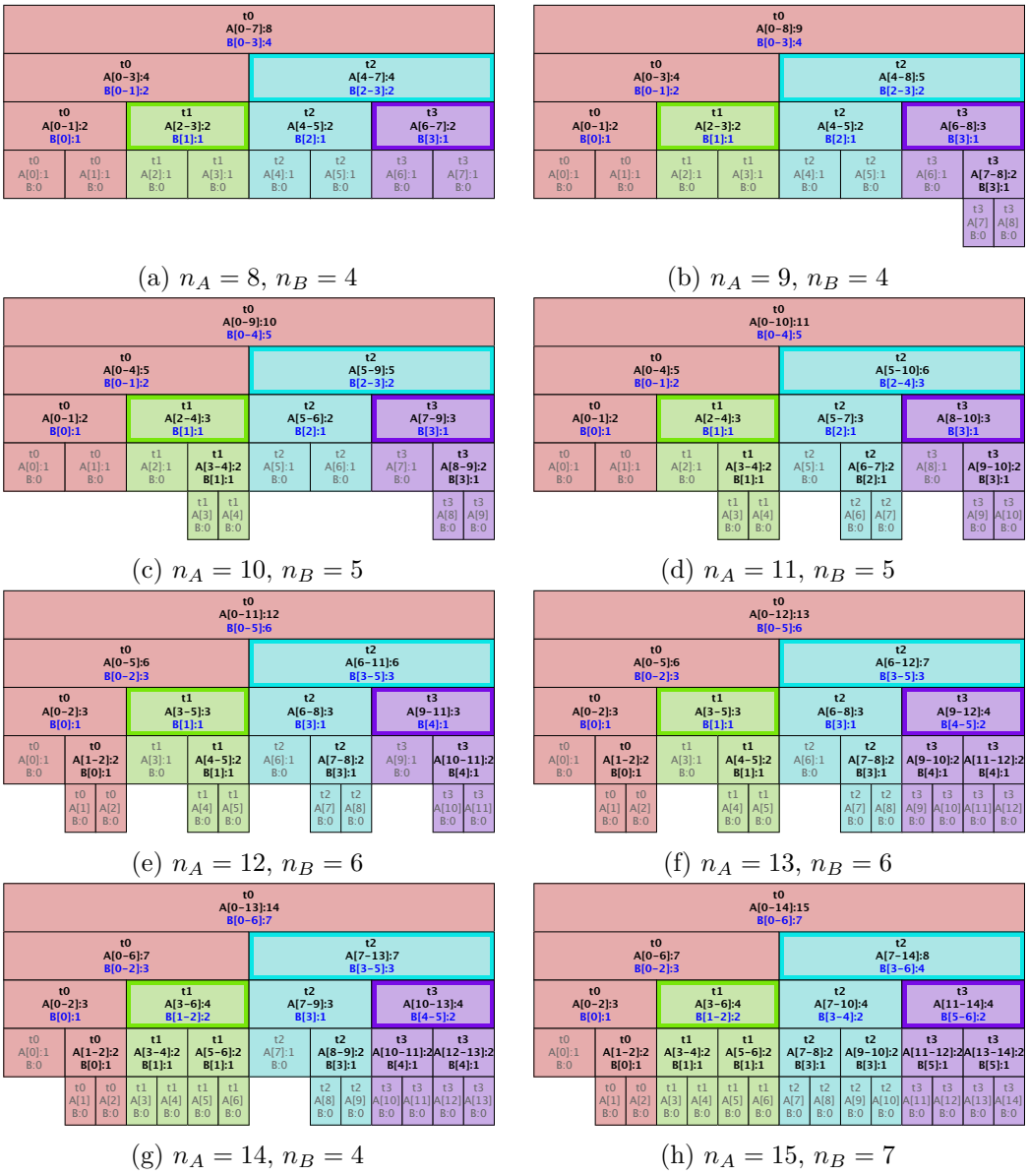
3.5.3 Memory reduced merge sort algorithm

The merge algorithm described above makes it possible to optimize the memory consumption of an individual merge operation, but the actual target is to keep the memory consumption of the entire merge sort algorithm as low as possible.

The execution of merge with the most elements to merge is the last one that is executed during a run of merge sort. This last execution of merge merges the first half and the second half of all elements in A. That implies that the size of B only needs to be $\frac{n}{2}$ for the total merge sort algorithm when A is of size n.

For a single threaded implementation of merge sort this is a relatively easy task, because every execution of merge can use the indices between 0 and $m - l$ to store data in B. But for a multithreaded implementation of merge sort this can lead to a race condition, because different threads will write and read to B in an undefined order. Synchronization between the threads is not really a solution because of its huge negative performance impact. Most of the time there would be only one thread running, because all merge operations would need an exclusive access for B and the merge routine consumes the most processing time of the merge sort algorithm. Therefore, this would not improve the performance compared to a single-threaded merge sort.

To solve this problem, each thread needs its own range in B where only this thread is allowed to read and write. After all child-threads which have been started by a thread are terminated again, the parent thread can of course reuse also the memory area that was reserved for the child-threads. For example: merge sort should be executed with two threads. Then, the first half of the input data is sorted by the parent thread and the second half by the child thread. Thereby, the parent thread is only allowed to use the first half of B as long as the child thread is running. The child thread may only use the second half of B. After the two halves have been sorted, the child thread terminates and



(a) $n_A = 8$, $n_B = 4$

(b) $n_A = 9$, $n_B = 4$

(c) $n_A = 10$, $n_B = 5$

(d) $n_A = 11$, $n_B = 5$

(e) $n_A = 12$, $n_B = 6$

(f) $n_A = 13$, $n_B = 6$

(g) $n_A = 14$, $n_B = 4$

(h) $n_A = 15$, $n_B = 7$

Figure 3.7: Illustration of the required indices in B for a multithreaded merge sort (4 threads) that ensures $n_l \leq n_r$ for every recursive call; the color indicates the thread that is executing the merge; all nodes represent the tree T_A, the nodes with "B: 0" does not belong to T_B

now the parent thread can use the entire memory space in B to merge the two already sorted halves to the final sorted output sequence. For a merge sort call with more than two threads, the same principle applies recursively.

In order to control from which index on each thread is allowed to write into B the

already mentioned parameter l_B (left for B) is needed in the merge algorithm. All merge sort calls within threads are allowed to use the same start index l_B since everything is executed serially within a thread and therefore no race condition can occur. Since for each recursive call of merge sort the length of the array to be sorted is cut in half, the required memory capacity in B is cut in half as well with every recursive call. Therefore, the biggest amount of the space that is required by a thread in B is needed in the recursion level in which the thread itself was generated. This means that a new l_B has to be calculated only for those merge sort calls for which a new thread is started. Furthermore, when calculating a l_B for a new thread only merge sort calls for which a new thread is started and that sorts subsequences further to the left than the thread for which the l_B is calculated must be taken into account.

Concretely, this means that the merge sort algorithm further has to calculate the first index $l_{B_{\text{sub-r}}}$ from which on the new thread can write into B, if the right half is sorted by a new thread. $l_{B_{\text{sub-r}}}$ is provided to the merge sort call that sorts the right half of the current subsequence as parameter l_B. $l_{B_{\text{sub-r}}}$ can be defined in dependence on the current start index l_B in B and the length of the left half of A.

If the center m and the right end r represent the last valid indices ($x \leq m$, $x \leq r$) as it is the case in the presented pseudocodes, the length of the left part can be calculated by $n_l = m + 1 - l$. However, if m and l are considered to be lengthwise and therefore valid indices must fulfill $x < m$ or $x < r$, the length of the left part must be calculated by $n_l = m - l$. Also, one should pay attention to calculate m in a way that for input array lengths n which are not dividable by 2 the larger half always gets processed by the merge sort call which sorts the right half. So the following equation is valid: $n_l \leq n_r$ with $n_l = |A_1| = |A[l, ..., m]|$ and $n_r = |A_2| = |A[m + 1, ..., r]|$. Therefore, if valid indices x defined by $x \leq m$, $x \leq r$ the middle m should be calculated by $m = \lfloor \frac{l+r-1}{2} \rfloor$. Because then the index range l to m (including) corresponds to the rounded-down half of n.

If the computation of m is chosen in a way that $n_l \leq n_r$ always holds, $l_{B_{\text{sub-r}}}$ can be calculated by $l_{B_{\text{sub-r}}} = l_B + \lfloor \frac{n_l}{2} \rfloor$. This works because the merge algorithm described above only copies the first half of A ($A_1 = A[l, ..., m]$) into B and the second half $A_2 = A[m+1, ..., r]$ always remains in A. Since the second half contains the remainder of the division $\frac{n}{2}$, the remainder does not need to be considered in the calculation of $l_{B_{\text{sub-r}}}$. Figure 3.7 illustrates the memory usage in B per thread in dependence of the size of A for a merge sort implementation where $n_l \leq n_r$ always holds.

Length of B for $n_l \geq n_r$

If on the other hand the calculation of m is chosen in a way that the remainder of the division $\frac{n}{2}$ is contained in the first half ($n_l \geq n_r$), the calculation of $l_{B_{\text{sub-r}}}$ is much more complex. This case for example would happen if m is calculated by $m = \lfloor \frac{l+r}{2} \rfloor$ in a system which defines the index boundaries as $x \leq m$, $x \leq r$. This case would probably occur very frequently in real implementations. Also in this case it is possible to calculate the size in B required by the merge sort call. Therefore, every l_B can be calculated.

In order to better understand the memory consumption of the merge calls one should imagine a binary tree T_A whose depth-first traversal corresponds to the recursive calls of the merge sort method. If a node of this tree is passed at the left side, A is divided into two halves and both subtrees below the node are sorting the left and right half of A. After these two halves are being sorted by the subtrees, the node is passed on the right side. Thereby, both sorted halves of A are merged together to the one sorted sequence.

In the leaves of this binary tree T_A, the length of the arrays that are to be sorted is only 1. Because one element is sorted correctly – which is trivial – the leaves of this tree do not need any memory in B. Therefore, the only relevant aspect for calculating the size of B is the subtree T_B which consists of all inner nodes of the tree T_A.

Let us define h_B as the height of the tree T_B and define the levels of the tree as d_B with $0 \leq d_B < h_B$. The height h_B can be calculated based on n by $h_B = h_A - 1 = \lceil \log_2(n) \rceil$.

For every node of the tree T_B in the last level that is entirely filled with nodes an element in B is required. If this level is indeed the last level then the calculation of $l_{B_{\text{sub-r}}}$ would be done, because in this case the result of $\log_2(n)$ is a nonnegative integer ($\log_2(n) \in \mathbb{N}$). Consequently, A can be divided into two equal halves with no remainder. Therefore, for every node $n_l = n_r$ is fulfilled.

Because we can assume that T_B is a balanced tree the count of the nodes in the last complete level of T_B is defined by:

$$N_B^c = \frac{N_A^c}{2} = \frac{2^{\lfloor \log_2(n) \rfloor}}{2} = 2^{\lfloor \log_2(n) \rfloor - 1}$$

(N_A^c is the count of the nodes in the last complete level of T_A.) If the last level is not completely filled up with nodes, the memory consumption that goes beyond N_B^c can be calculated by the count of the nodes in the last level, therefore, by the "remainder" of $\log_2$. The count of nodes in the last level of the tree T_B can be calculated as follows:

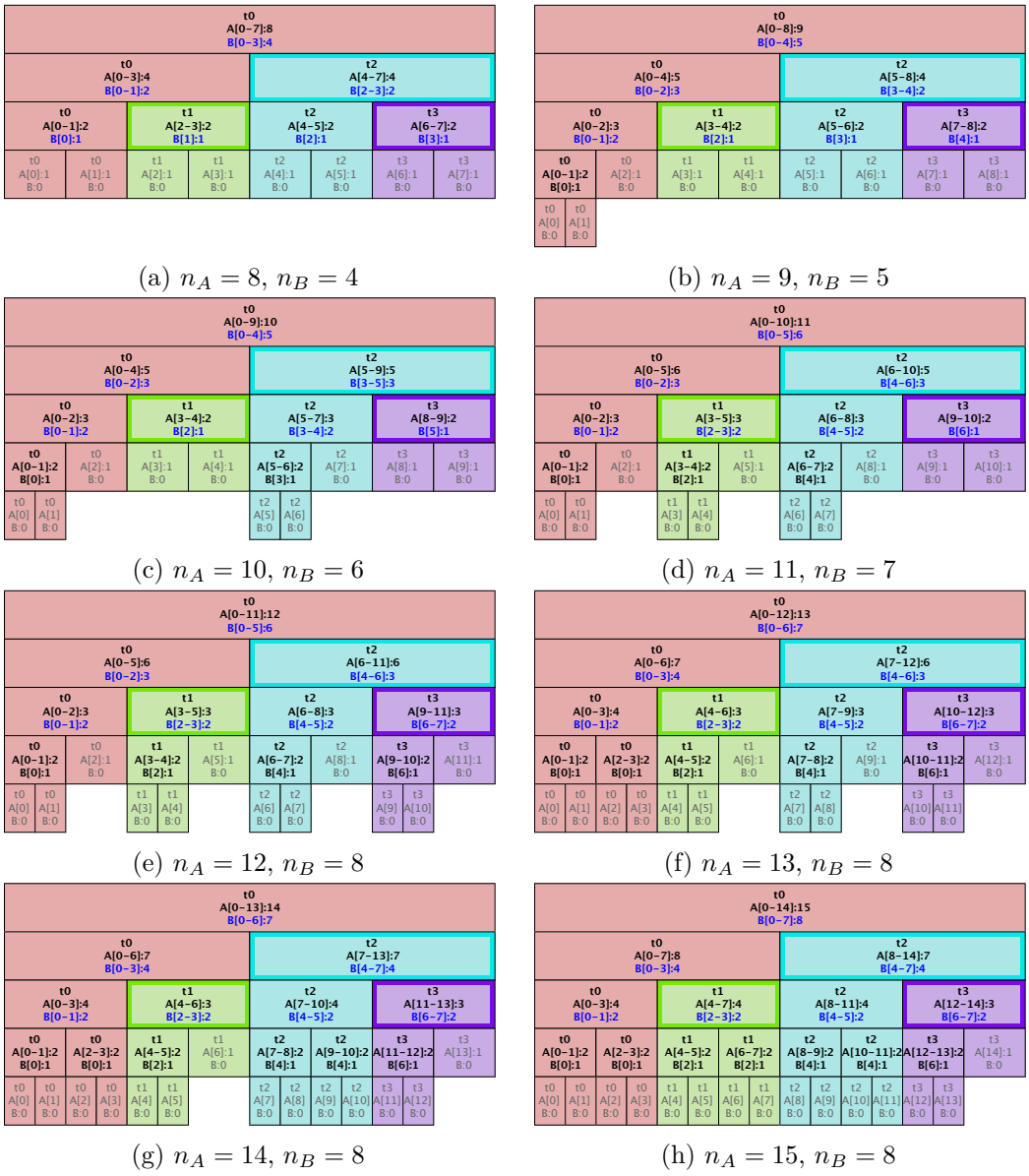
$$N_B^l = \left(\left\lfloor \frac{n}{2} \right\rfloor - N_B^c \right) \cdot 2 = \left(\left\lfloor \frac{n}{2} \right\rfloor - 2^{\lfloor \log_2(n) \rfloor - 1} \right) \cdot 2 = n - 2^{\lfloor \log_2(n) \rfloor}$$

In order to be able to calculate the required size of B, we also need the number of threads which computes the subset of A for which the current node is responsible. The number of threads that work in a subtree whose root node is in the level d can be calculated as follows:

$$t_d = 2^{\log_2(t) - d}$$

Theoretically, it is possible that n is in comparison to t so small that not all available threads t are ever used. Therefore, for an exact calculation of the length of B, the number of really running threads needs to be calculated:

$$t_w = \max\Big(1, \min\Big(t_d, N_B^c + \max(0, N_B^l - N_B^c)\Big)\Big)$$



(a) $n_A = 8$, $n_B = 4$

(b) $n_A = 9$, $n_B = 5$

(c) $n_A = 10$, $n_B = 6$

(d) $n_A = 11$, $n_B = 7$

(e) $n_A = 12$, $n_B = 8$

(f) $n_A = 13$, $n_B = 8$

(g) $n_A = 14$, $n_B = 8$

(h) $n_A = 15$, $n_B = 8$

Figure 3.8: Illustration of the required indices in B for a multithreaded merge sort (4 threads) that ensures $n_l \geq n_r$ for every recursive call; the color indicates the thread that is executing the merge; all nodes represent the tree T_A, the nodes with "B: 0" does not belong to T_B

The lowest level that contains nodes for which new threads are created is defined by: $d_t = \log_2(t)$. For better readability all nodes in the level d_t are label as: "`Threaded-Node`"

Let us assume an array A of the length $n_0 = 2^x$ that needs to be sorted. Therefore, the corresponding tree T_B has a last level completely filled with nodes (see Subfigure 3.8a).

If the length of array A is increased by one element, the subtree that is sorted by the thread with the ID 0 will get one new leaf node at the left end of the tree. The thread with ID 0 must therefore sort one element more than all other threads. Since $n_l \geq n_r$ holds, it also requires a by one element larger memory area in B (see Subfigure 3.8b). If A is increased by one additional element, the tree obtains a new leaf in the far left at the right half of the tree. The thread with the ID $t/2$ now has to sort one element more and therefore needs space for one more element in B (see Subfigure 3.8c). The next additional element must be sorted by the thread with the ID $t/4$. Of course also this thread therefore needs now an additional element in B (see Subfigure 3.8d).

The same happens with every additional element added to A until A has reached a length of $n_0 + t$ (see Subfigure 3.8e). When the length of A is then increased by one further element and is therefore $n_0 + t + 1$ elements long, this new element is added to the subtree that is sorted by the thread with the ID 0 (see Subfigure 3.8f). This time, however, the element is added to the right subtree below the `Threaded-Node`. Since all merge operations within a thread can use the same memory range in B, the right subtree can use the memory that was already added for the element count $n_0 + 1$. The merge operation of the `Threaded-Node` also includes one element more, but this has no effect on the size of B required by the `Threaded-Node` because the newly added element stays in A during the merge operation performed in the `Threaded-Node` and therefore, does not have to be copied to B. For this `Threaded-Node` $n_l = n_r$ does now apply. But the nodes that are located in the tree above the `Threaded-Node` have to copy again one element more to B, because for these nodes $n_l \geq n_r$ does now apply. However, for an A with a length of $n_0 + t$, two `Threaded-Nodes` with one common parent node together need more space in B than the parent node. Therefore, there is already enough space in B for all nodes above the `Threaded-Node`, and the length of B does not need to be increased.

If A is further increased, the n_l and n_r of all `Threaded-Nodes` are being rebalanced. Therefore, there is no need to change the size of B until the input array A reaches the length of $n_0 + 2 \cdot t$. Figure 3.8 illustrates this behaviour of the memory usage in B per thread in dependence of the size of A for a merge sort implementation where $n_l \geq n_r$ always holds (also compare with Figure 3.7).

From $n_0 + 2 \cdot t + 1$ on, the left subtrees below the `Threaded-Nodes` starts again to get larger than the right subtrees, which leads to $n_l \geq n_r$ inside the `Threaded-Nodes` and therefore B must be increased again for each additional element. After t elements have been added, all n_l and n_r values inside the `Threaded-Nodes` start to rebalance again and therefore the length of B is not required to change. This behavior alternates always after t added elements and repeats until the tree has a last level that is completely filled with nodes again. A function that illustrates this behavior when increasing the length of A is shown in the Figure 3.9.

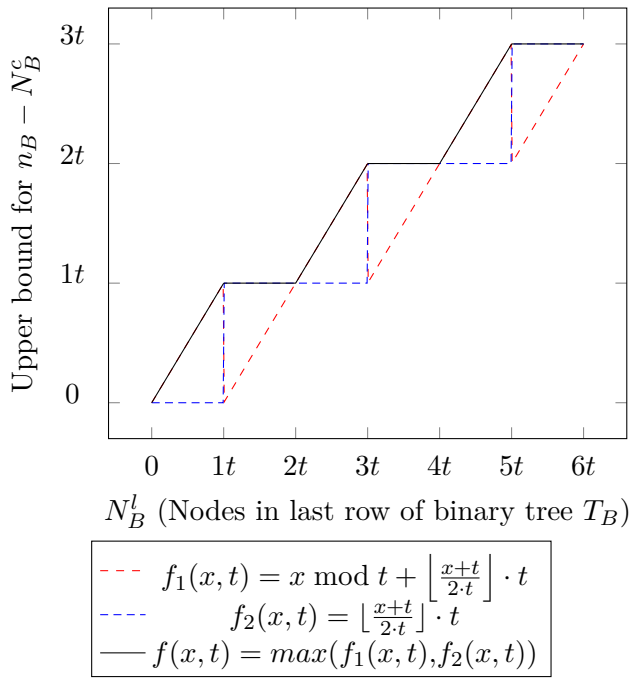


Figure 3.9: Upper bound for additional elements that are required for leaves N_B^l

A mathematical definition for this mapping can be constructed by the following functions:

- A function that looks like a sawtooth whose teeth rise at 45 degrees and have a length as well as a height of t. (Corresponds to the number of all those nodes in the last level of T_B which can not be evenly distributed across all threads.)

$$f_{1a}(x,t) = x - \lfloor \frac{x}{t} \cdot t \rfloor = x \bmod t$$

- Staircase function whose step height equals to t and the step depth to $2 \cdot t$. (Corresponds to the half of the nodes in the last level, which can be evenly distribute across all threads.)

$$f_{1b}(x,t) = \left\lfloor \frac{\frac{x}{t}}{2} \right\rfloor \cdot t = \left\lfloor \frac{x}{2 \cdot t} \right\rfloor \cdot t$$

- f_1 is generated by adding together f_{1a} and f_{1b}. Thereby, the staircase function f_{1b} always raises the sawtooth f_{1a} by t after $2 \cdot t$ elements. The result is a function which increases by 1 for every added element until t elements are added and afterwards falls to 0. Subsequently, however, for the next $2 \cdot t$ element, the mapping continuously increases by 1 each time an element is added and then it is decreased by t. This behavior is repeated for each additional element. As a matter of principle, the mapped values are always increasing but not continuously.

$$f_1(x,t) = f_{1a}(x,t) + f_{1b}(x,t)$$

- f_2 is a staircase function like f_{1b} but moved to the left by t. This function is required to trap the decreasing function values that occurs always when $\lfloor x/t \rfloor \bmod 2 = 1$ in f_1.

$$f_2(x,t) = \left\lfloor \frac{\frac{x+t}{t}}{2} \right\rfloor \cdot t = \left\lfloor \frac{x+t}{2 \cdot t} \right\rfloor \cdot t$$

- The final function f can then be defined as follows:

$$f(x,t) = \max(f_1(x,t), f_2(x,t))$$

For $n < 2 \cdot t$, the following must be considered: Once the last level of the tree T_B contains more than half of its maximum possible nodes, with every element added to A, not only the number of leaves increases but also the number of actually started threads t_w. This causes the function f to return a too large result. To solve this problem, one could use the number of threads already started above the last level of the tree T_B instead of t_w. This could be done by replacing t_w with $2^{\lfloor \log_2(t_w) \rfloor}$. Or it can be solved by simply limiting f by N_B^c which exactly corresponds to the half of the maximum possible nodes in the last level of T_B. Since the function f never returns values greater than N_B^c for $n \geq 2 \cdot t$, no distinction of cases is required. Therefore, N_B^c can always be used as an upper bound for f: $\min(N_B^c, f(N_B^l, t))$

The effects described above on the length of B when increasing the length of A apply not only to the root node of the whole tree, but also to all subsequences of A and B for which a node is responsible that uses a new thread for sorting. The formulas can therefore also be used for all recursive merge sort calls where a new thread is created. Thereby, n has to be replaced just with the length of the respective subsequence which is sorted by this particular merge sort call. Furthermore, of course, t_w also has to be recalculated in every merge sort call.

The required length n_B for B can therefore be calculated for each node (merge sort call) by the following function:

$$n_B = N_B^c + \min\Big(N_B^c, f(N_B^l, t_w)\Big)$$

This results in the following function which is able to calculate every required $l_{B_{\mathrm{sub-r}}}$ for a merge sort implementation that has r and m defined in a way that $n_l \geq n_r$ holds for every merge operation.

$$l_{B_{\mathrm{sub-r}}} = l_B + n_B = l_B + N_B^c + \min\Big(N_B^c, f(N_B^l, t_w)\Big)$$

3.6 Optimization of merge sort through insertionsort

In order to further speedup the algorithm the author picked up the idea Hoare [Hoa62] and Sedgewick et al. [SA78] had to speed up quicksort with the usage of insertion sort for

small subsequences and tested if this applies as well to merge sort. For sorting oriented points it does not improve the performace significantly, but for some test cases a slight runtime improvemen can be obtained – see Section 5.2 for details.

Algorithm 3.3: InsertionSort (Adapted from [BKBH])

Input: array A
Parameters: sorting begin and end indices l, r
Output: sorted subset A[l],...,A[r]

```
1  procedure insertionSort(integer l, integer r)
2      for j ← l + 1 to r do
3          key ← A[j]
4          i ← j − 1
5          while i ≥ l and i < r and key < A[i] do
6              A[i + 1] ← A[i]
7              i ← i − 1
8          end
9          A[i + 1] ← key
10     end
```

Algorithm 3.4: SortPartition (Adapted from [BKBH])

Input: array A
Parameters: sorting begin and end indices l, r
Output: sorted subset A[l],...,A[r]

```
1  procedure sortPartition(integer l, integer r)
       // choosing pivot item as median of l–m–r.
2      m ← l + (r−l)/2
3      r ← r − 1
4      if A[m] < A[l] then swap A[m] and A[l]
5      if A[r] < A[l] then swap A[r] and A[l]
6      if A[r] < A[m] then swap A[r] and A[m]
7      swap A[m] and A[l]
8      pivot ← l
9      while l < r do
           // move left while item ≤ pivot
10         while l ≠ r do
11             if A[l] < A[pivot] then l ← l + 1                    // less
12             else if A[pivot] < A[l] then break
13             else l ← l + 1                                       // equal
14         end
           // move right while item > pivot
15         i ← 0
16         while A[pivot] < A[r] do
17             i ← i + 1
18             r ← r − 1
19         end
20         if l < r then swap A[l] and A[r]
21     end
       // right is final position for the pivot
22     swap A[pivot] and A[r]
23     return r
```

Algorithm 3.5: Quicksort (Adapted from [BKBH])

Input: array A
Parameters: sorting begin and end indices l = 0, r = A.*size*()
Output: sorted subset A[l],...,A[r]

```
1  procedure quicksort (integer l, integer r)
2      stackP ← 1
3      repeat
4          if l < r then
5              if r − l < 10 and use insertion sort for small subsequences then
6                  insertionSort (l, r)
7                  stackP ← stackP − 1
8                  l ← lowStack[stackP]
9                  r ← highStack[stackP]
10                 continue
11             end
12             pivot ← sortPartition (l,r)
13             if pivot − 1 − l < r − pivot + 1 then
14                 lowStack[stackP] ← pivot + 1
15                 highStack[stackP] ← r
16                 stackP ← pivot + 1
17                 r ← pivot
18             else
19                 lowStack[stackP] ← l
20                 highStack[stackP] ← pivot
21                 stackP ← stackP + 1
22                 l ← pivot + 1
23             end
24         else
25             stackP ← stackP − 1
26             l ← lowStack[stackP]
27             r ← highStack[stackP]
28         end
29     until stackP > 0
```

Algorithm 3.6: Merge sort initialisation for multithreaded and memory optimized merge sort

Input: array A
Parameters: number of threads that should be used to sort the array
Output: sorted array A

```
1 procedure mergesortInit(integer maxThreads)
    // make shure maxThreads is in 2^x
2   threadedDepth ← ⌊log_2(maxThreads)⌋
3   threadCount ← 2^threadedDepth
4   allocate array B with calcSizeOfB(threadCount, A.size()) elements
5   mergesortThreaded (0, A.size() − 1, 0, threadCount, 0)
```

Algorithm 3.7: CalcSizeOfB for multithreaded and memory optimized merge sort

Parameters: elementCountA: number of elements to sort;
maxThreads: number of threads that should be used to sort the array
Result: required temporary array size (size of B)

```
// maxThreads have to be 2^x
1 procedure calcSizeOfB(integer maxThreads, integer elementCountA)
2   N_A^c ← 2^⌊ld(elementCountA)⌋ // The count of nodes in last complete
      level of binary tree of A
3   N_A^l ← (elementCountA − 2^⌊ld(elementCountA)⌋) · 2 // The count of nodes
      in last level in binary tree of A
4   N_B^c ← ⌊N_A^c / 2⌋ // The count of nodes in last complete level of
      binary tree of B
5   N_B^l ← ⌊N_A^l / 2⌋ // The count of nodes in last level of binary
      tree of B
6   T_w ← max(1, min(maxThreads, N_B^c + max(0, N_B^l − N_B^c))) // number of
      really working threads
7   s ← N_B^c + min(N_B^c, max(N_B^l − ⌊N_B^l / T_w⌋ · T_w + ⌊(N_B^l / T_w) / 2⌋ · T_w, ⌊(N_B^l + T_w) / T_w / 2⌋ · T_w))
8   return s;
```

Algorithm 3.8: Merge for multithreaded and memory optimized merge sort

Input: array A, array B
Parameters: sorting begin middle and end indices l, m, r
Integer l_B
Output: sorted subset A[l],...,A[r]

```
1  procedure mergeT(integer l, integer m, integer r, integer l_B)
2      i ← l_B
3      j ← l;
       // Copy the first half of elements from A to B.
4      while j ≤ m do
5          B[i] = A[j]
6          i ← i + 1
7          j ← j + 1
8      end
9      i ← l_B
10     k ← l
       // Copy the values back from A_2 and B to A, in correctly
          sorted order.
11     while k < j and j ≤ r do
12         if B[i] < A[j] then                  // use ≤ for a stable merge
13
14             A[k] ← B[i]
15             i ← i + 1
16         else
17             A[k] ← A[j]
18             j ← j + 1
19         end
20         k ← k + 1
21     end
       // Copy back the remaining values from B to A
22     while k < j do
23         A[k] ← B[i]
24         k ← k + 1
25         i ← i + 1
26     end
```

Algorithm 3.9: Multithreaded and memory optimized merge sort

Input: array A, array B
Parameters: sorting begin and end indices l, r
Output: sorted subset A[l],...,A[r]

```
1  procedure mergesortThreaded(integer l, integer r, integer l_B, integer
     maxThreads, integer threadStartDepth)
2      if r − l < 10 and use insertion sort for small subsequences then
3          insertionSort(l, r + 1)
4      else if l < r then
5          m ← ⌊(l+r)/2⌋
6          if 2^(threadStartDepth+1) ≤ maxThreads then
               // sort second half with a new thread
7              T_l ← 2^(ld(maxThreads)−(threadStartDepth+1))
8              l_B_sub−r ← l_B + calcSizeOfB(T_l, m + 1 − l)
9              threadStartDepth ← threadStartDepth + 1
10             Tnew ← execute new Thread begin
11                 mergesortThreaded (m + 1, r, l_B_sub−r, maxThreads,
                     threadStartDepth)
12             end
               // sort first half with current thread
13             mergesortThreaded (l, m, l_B, maxThreads, threadStartDepth)
14             wait until Tnew is finished
15         else
16             mergesortThreaded (l, m, l_B, maxThreads, threadStartDepth)
17             mergesortThreaded (m + 1, r, l_B, maxThreads, threadStartDepth)
18         end
19         mergeT(l, m, r, l_B)
20     end
```

CHAPTER 4

Postprocessing

As already mentioned in Section 2.2, the "Multilevel Streaming for Out-of-Core Surface Reconstruction" approach from Bolitho et al. [BKBH07] produces watertight surface reconstructions. While this is a nice behavior for many cases, it can sometimes lead to problems. For scanned point clouds of objects with a closed surface such as characters, this works very well. Figure 5.4 shows a reconstructed surface mesh of a point cloud sampled from a dragon sculpture [Sta] with such a closed surface. For scans of a terrain, which does not have a closed surface, the behavior of the Poisson Surface Reconstruction [KBH06] often leads to some big unwanted triangles. The reconstruction of a point cloud acquired from Mount St. Helens shown in Figure 4.1 shows such unwanted triangles in the two images on the left side. These unwanted triangles sometimes create something like a half sphere over the terrain, which makes it harder to take a quick look on the reconstructed surface than necessary. This is especially a problem for unexperienced users. To overcome this problem, two simple approaches with the aim to remove these big unwanted triangles from the reconstructed mesh were tested.

The idea of the **first approach** was to check how many samples of the point cloud are located near to the triangle in order to evaluate the relation of the size of the triangle to the count of nearby located samples. In order to do that, first an axis-aligned bounding box (AABB) is computed for each triangle. Then the number of samples of the point cloud that lie inside the AABB is calculated. To calculate the "density" of the triangle the number of points within the AABB is divided through the volume of the AABB. Then all triangles are sorted according to their "density" and then a user-controlled percentage of triangles with the least density will be removed. The problem with this approach is that – at least for reconstruction resolutions that are high enough to reconstruct as many details as possible – most AABB do not contain any samples. This happens because the Poisson Surface Reconstruction [KBH06] creates meshes that are so smooth that most of the triangles are located too far away from the samples of the input point cloud.

Since the first approach mostly does not reach the goals a **second approach** was tested. Within this approach the triangles are sorted by their area and a user-controlled percentage of the triangles with the largest area will be removed. This approach works acceptably for low-resolution reconstruction (2^8), though narrow but large triangles often will not be removed. For high resolution reconstructions holes often occur for surface parts with low sampling density or at very planar surface parts. If the user uses a very small percentage (0.1), it works generally, though the reconstructed mesh still contains some unnecessary triangles and ragged boarders. However, the optimal percentage depends on the reconstruction resolution. In order to avoid the removal of needed triangles, a further constraint was added to the algorithm: namely every edge of the triangle needs to exceed a certain length (0.005) to be considered for a removal. This check of the edge lengths makes the approach more independent from the reconstruction resolution and the sampling density of the input point cloud. With this approach still unnecessary triangles exist, though they do not disturb a quick view over the model.

Figure 4.1 shows a reconstructed mesh from a point cloud with 5.7 million points. The mesh was reconstructed with a maximum octree depth of 10. The left two images show the result without any postprocessing. The right two images show the results with postprocessing, where a percentage of max. 1% of the largest triangles with a minimum edge length of 0.005 has been removed.

Both approaches where implemented as a postproccessing step and designed as **out-of-core algorithms**. Therefore, they can also be used to postprocess meshes reconstructed with resolutions of 2^{14} and higher, which generally have a vertex count that does not fit in common main memory. In order to make the algorithm out-of-core, the file which contains the faces and their vertex indices are streamed. The vertex positions are read from another file and buffered in a least recently used (LRU) cache. Since the neighboring triangles are stored close to each other in the face index file that was created by the reconstruction process, the LRU cache in combination with a buffered reader efficiently prevents random readings from disk. The indices of the faces that should be removed from the mesh are just stored in an in-core hash-table. Because the faces that should be removed are only some 100 or $1,000$, this hash-table is so small that it does not need to be out-of-core. Finally, when writing out the mesh as `ply` or `obj` file, every face is checked against the hash-table that contains the indices of unwanted faces. If the actual face is in the hash-table of unwanted faces it is just not written to the final output.

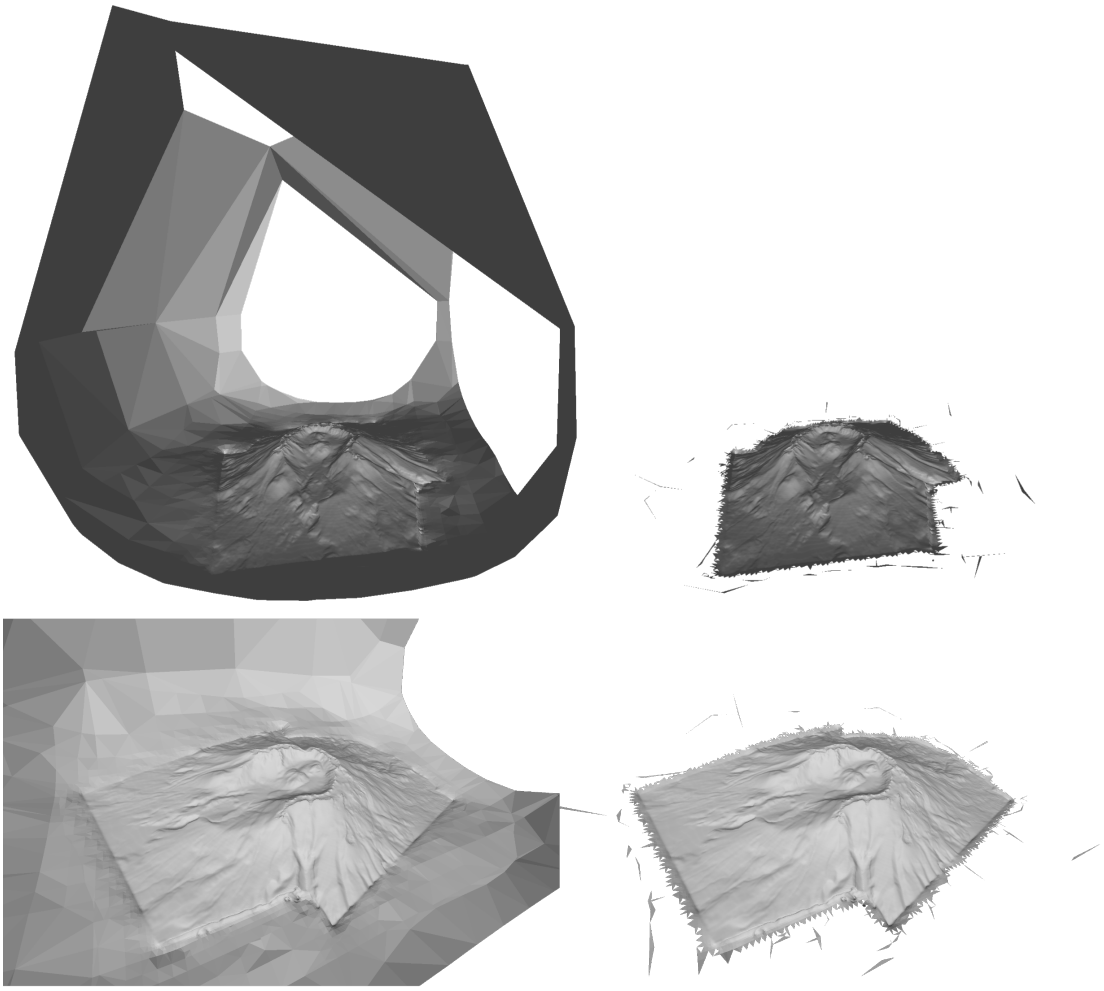


Figure 4.1: Reconstructed mesh with 43.0602 vertices and 86.0242 faces from a scan of Mount St. Helens. with 5.784.252 Points. For the reconstruction a maximal octree depth of 10 was used. Left images: without Postprocessing, right images: removed maximal 1% of the largest triangle with an minimum edge length of 0.005

CHAPTER 5

Results

5.1 Test environment

All runtime tests were done on an Apple MacMini with a 2.3GHz Intel® Core™i7-3615QM quad-core processor, 16GB RAM and a 500GB Crucial MX200 SSD connected via Thunderbolt. The used operating system was Windows 8.1 Pro 64Bit. All tested algorithms were implemented in C++ and compiled with Microsoft® Visual Studio® 2010 Premium as 64Bit application.

If you look at the runtime results, which were performed with 8 threads, be aware that the used CPU has only 4 cores. Therefore, the slight difference that can be seen between the test with 4 and 8 threads does not mean that the algorithm does not scale well for more then 4 threads. The little runtime improvement when using 8 instead of 4 threads comes from Intel's Hyper-Threading Technology only and not from using the double amount of real CPU cores.

5.2 Sorting runtime

The results of the runtime measurements are shown in Figure 5.2. The tests have shown that at least for data with simple comparison operations (integer & float) the total runtime can be improved by using insertion sort for small subsequences – exactly like it applies to quicksort. Unfortunately, it seems that the use of insertion sort does not have any influence on the total runtime of sorting-processes which require rather complex comparison operations like the comparator for the oriented points.

One of the possible explanations for that phenomenon could be that the runtime needed by the oriented-point comparator is so huge that it destroys the advantages of smaller overheads of insertion sort even if a small n ($= 9$) is used due to the higher amount of necessary comparisons required by insertion sort. Insertion sort has in principal the

advantage of a smaller overhead compared to merge sort or quicksort because no recursive method calls are needed. However, insertion sort requires n^2 comparisons, whereas merge sort or quicksort need only $n \cdot \log_2(n)$ comparisons. Since the overhead for the recursive method calls is independent from the runtime complexity of the comparison operator, the runtime advantages of insertion sort decrease with increasing runtime complexity of the comparison operator even with very small n.

During the test for run-time measurements, however, other interesting observations have been made. One of them occurred in the tests with integer and float arrays: Quicksort seems to benefit more from the use of insertion sort at $n < 10^4$ than merge sort. But at approximate 10^6 elements it does not seem to have any significant influence on the total runtime of quicksort any more. At least the optimization through insertion sort does not behave logarithmically to the amount of elements to be sorted. In contrast to that the total runtime of merge sort also benefits at $n > 10^6$ from the usage of insertion sort.

Another interesting observation is the fact that the total runtime of quicksort does not behave like $n \cdot log_2(n)$ in practice as one would assume theoretically [Hoa62, CGD11]. The total runtime of merge sort on the other hand behaves also with big amount of data ($n > 10^6$) like $n \cdot log_2(n)$ and therefore exactly like assumed.

This leads to the fact that merge sort is faster than quicksort for the purpose of sorting big point clouds even under the use of only one thread (clearly visible in Figure 5.1). Another advantage of merge sort is its predictable runtime. The number of required comparisons and merge operations is defined by just the count of elements that need to be sorted. The required copy operations of a single merge operation are not exactly known at the beginning but even without that information, a good runtime estimation can be calculated easily. This allows an application developer to present an accurate remaining time estimation to the user. As a result, merge sort is better suited to our needs than quicksort.

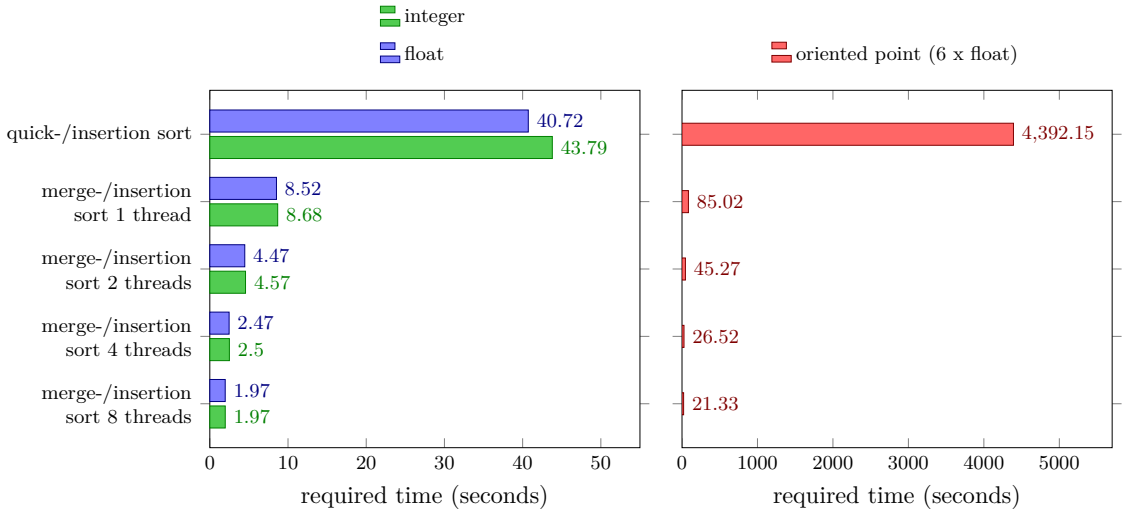


Figure 5.1: comparison of required time for sorting $50 \cdot 10^6$ elements

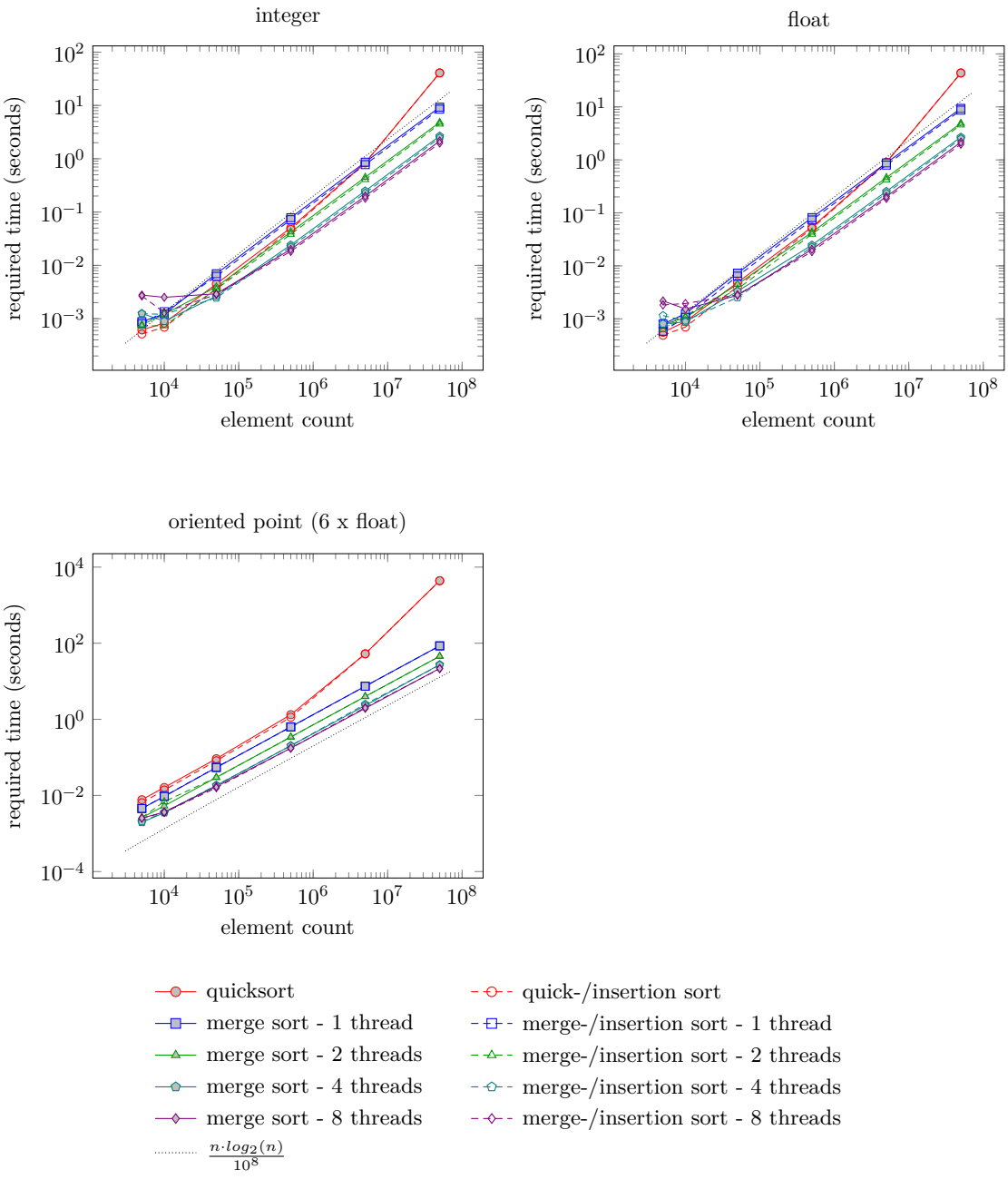


Figure 5.2: integer, float and oriented point sorting performance

5.3 Surface reconstruction

The author has evaluated the Surface Reconstruction Plugin for Scanopy with point clouds of different sizes. In order to show the most relevant results, the author indicated runtime measurements for two different point clouds, each reconstructed with 3 different maximum octree depths as visible in Table 5.1. The right bar chart in Figure 5.3 clearly shows that for reconstructions of point clouds with hundreds of millions of points, the total runtime of the reconstruction was significantly reduced by the multithreaded merge sort algorithm introduced in Chapter 3. For relatively small point clouds (less than 5 million points) and high octree depths (12) the time required for sorting is only a small percentage of the total runtime. This can be seen from the left bar chart in Figure 5.3. Since the multithreaded merge sort algorithm affects sorting only, the result on the total runtime is not significant. However, for point clouds with hundreds of millions of points, where the time required for sorting is up to 80% and more, the sorting algorithm has a high impact on the total runtime. Therefore, the proposed multithreaded merge sort algorithm can significantly reduce the total runtime as indicated in Table 5.1 from nearly 7 hours to less than 1 hour.

The reconstruction results for the Asian Dragon and the Siebenschläferhöhle data set that was used for the runtime measuring visible in Table 5.1 and Figure 5.3 are shown in

Name	Asian Dragon			Siebenschläferhöhle		
Points	3,609,600			382,784,008		
Buckets	8			512		
Octree depth	8	10	12	8	10	12
Resolution	256	1,024	4,096	256	1,024	4,096
Vertices	69,745	1,207,014	3,584,152	68,374	1,101,233	17,880,635
Faces	139,280	2,413,718	7,167,844	136,563	2,202,242	35,760,188
Computing Centroid	1.1			167.5		
Computing Covariance	1.0			158.1		
Computing Bounding Box	1.1			170.4		
Creating Buckets	1.3			190.2		
Sort Algorithm	**quick**		**merge**	**quick**		**merge**
Sorting (in-core)	**7.6**		**1.6**	**21,346.0**		**145.7**
Read & writ Buckets for S.	0.3		0.4	54.8		73.8
Sorting total	**7.9**		**2.0**	**21,400.7**		**219.6**
Preprocessing total	12.5		6.5	22,086.9		905.8
Building Octree	5.5	17.6	71.8	443.4	621.0	1,070.3
Solving Laplacian	1.1	22.5	72.7	1.0	21.4	458.2
Extracting Iso-Surface	1.0	16.1	61.0	1.0	16.2	312.2
Reconstruction total	7.6	56.3	205.6	445.5	658.6	1,840.7
Save Obj File	0.6	10.1	30.4	0.6	9.5	162.2
Sort Algorithm	**quick merge**	**quick merge**	**quick merge**	**quick merge**	**quick merge**	**quick merge**
Total	**20.7 14.7**	**78.8 72.8**	**248.4 242.4**	**22,533.0 1,351.8**	**22,755.0 1,573.8**	**24,089.8 2,908.7**
Total without Sorting	12.7	70.9	240.4	1,132.2	1,354.2	2,689.1
Sorting %	**38.5% 13.4%**	**10.1% 2.7%**	**3.2% 0.8%**	**95.0% 16.2%**	**94.0% 14.0%**	**88.8% 7.5%**

Table 5.1: Required time (in seconds) for reconstruction with different models and different reconstruction resolutions. The rows labeled with *Vertices* and *Faces* belong to the reconstructed mesh. The row *Total without Sorting* contains the total time required for preprocessing, reconstruction and file saving except the time required for sorting (*Total - Sorting total*). The last row *Sorting %* expresses how much time of the total required time was spent on sorting (*Sorting total/ Total*). All tests where merge sort was used are executed with 8 threads (on a quad-core CPU with Hyper-Threading).

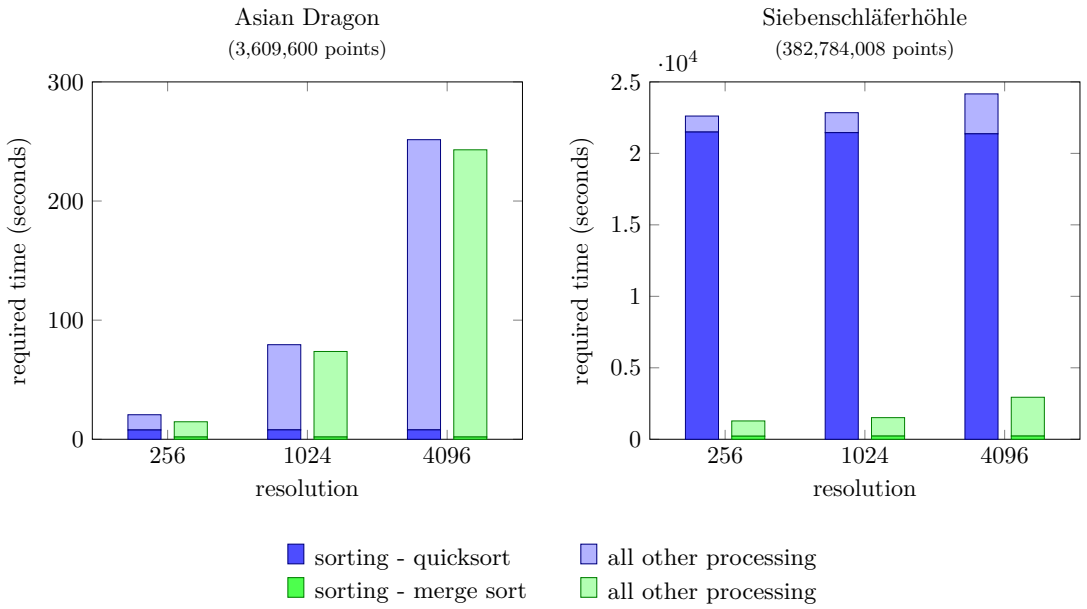


Figure 5.3: Comparison of the time (in seconds) required for sorting in comparison to the total time required for reconstruction with different models and different reconstruction resolutions. The blue bars visualize the time required when the original quicksort is used and the green bars visualize the time required when the multithreaded merge sort is used with 8 threads (on a quad-core CPU with Hyper-Threading).

Figures 5.4, 5.5 and 5.6. Figures 5.4 (Asian Dragon) and 5.5 (Siebenschläferhöhle) show the results of the reconstruction with a maximum octree depth of 12. Figures 5.5 provides a comparison of some details from the Siebenschläferhöhle data set reconstructed with different maximum octree depths (8, 10 and 12).

The reconstruction approach from Bolitho et al. [BKBH07] is able to perform the whole reconstruction process out-of-core. Therefore, even meshes which are so detailed, that they do not fit into the main memory can be reconstructed. For example: a reconstruction of the Siebenschläferhöhle data set with nearly 383 million points can be processed with a maximal octree depth of 14. This corresponds to a reconstruction resolution of 16.384 and an out-of-core octree size of 75.9 GByte. The result of this surface reconstruction is a mesh with approximately 143 million vertices and 285 million faces. When the sorting is done with the multithreaded merge sort algorithm, this reconstruction takes only 4.5 hours. Though realtime rendering of such a huge triangle mesh is another problem because only the vertex positions would require 1.6 Gbyte of graphics memory.

Figure 5.4: Reconstruction with a maximal octree depth of 12 from the Asian Dragon sculpture (point cloud data set from the Stanford 3D Scanning Repository [Sta]).

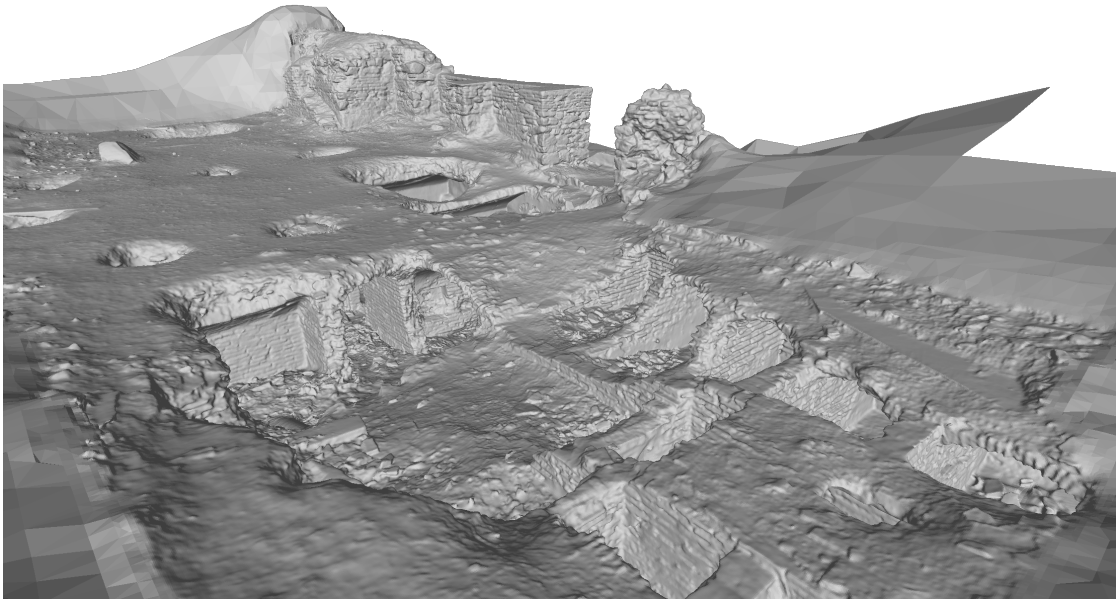


Figure 5.5: Mesh with 17M veteces and 35M faces reconstructed from the Siebenschläfer data set with a maximal octree depth of 12.

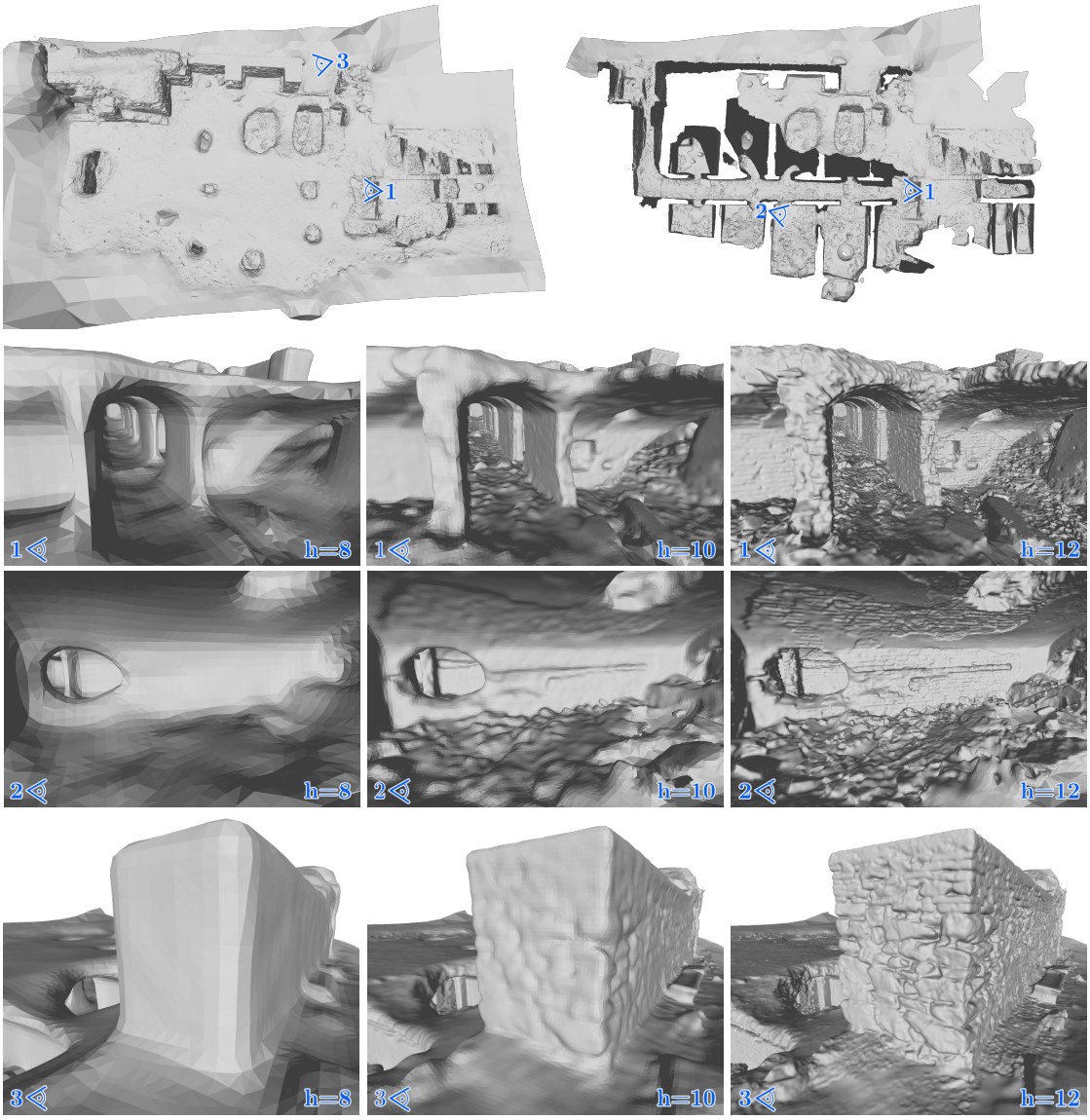


Figure 5.6: Comparison of reconstructions with different octree depth from the Siebenschläfer data set. The first image (upper left) shows an overview of the mesh; the second image (upper right) shows also an overview but with some faces cut away in order to allow a look inside the rooms under the earth. The nine images below show details of the same dataset. The images within one row show the same detail from meshes reconstructed with different resolutions. (The maximal octree depth is noted inside the image as $h =$.)

CHAPTER 6

Conclusion

The approach from Bolitho et al. [BKBH07] was implemented in order to convert point clouds of hundreds of millions of points in polygonal meshes. The author showed that the optimized multithreaded merge sort algorithm is able to significantly reduce the sorting time of very large point clouds. This is not so important for smaller point clouds but for huge point clouds this can improve the overall processing time for the out-of-core surface reconstructions significantly. Further, the author indicated that the reconstruction with the approach of Bolitho et al. [BKBH07] leads to problems if it reconstructs a surface from a non-closed object, e.g. the scan of a terrain like the Siebenschläferhöhle. The author has presented a simple approach how to remove the disturbing triangles of such reconstructions.

6.1 Future work

One possible option for future works could improve the removal of disturbing triangles. It would be interesting if it is possible to find a cluster which consists only of triangles with large area and remove the whole cluster. Further it would be interesting if the implementation of the mentioned co-rank algorithm for paralyzing the last $t - 1$ merging operations during a run of merge sort would lead to a further significant improvement of the overall sorting time.

As the tests showed, the used reconstruction approach is able to reconstruct meshes that are too big to be rendered on common graphics hardware. Therefore, it would be interesting to generate height maps that can be used to tessellate lower resolution meshes in order to obtain very detailed real-time renderings from meshes which are small enough to fit into common sized graphics memory.

The author got the impression that the used reconstruction algorithms led to an over-smoothing of the data. Further tests are required to investigate this observation and if

this is the case, further research would be required in order to improve this behavior.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AS87] Selim G. Akl and Nicola Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, 36(11):1367–1369, November 1987.

[BKBH] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. url-http://www.cs.jhu.edu/%7Ebolitho/Research/StreamingSurfaceReconstruction/. Multilevel Streaming for Out-of-Core Surface Reconstruction - Sourcecode, Accessed: 2016-05-05.

[BKBH07] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Multilevel streaming for out-of-core surface reconstruction. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, SGP '07, pages 69–78, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

[BTS+14] Matthew Berger, Andrea Tagliasacchi, Lee M. Seversky, Pierre Alliez, Joshua A. Levine, Andrei Sharf, and Claudio T. Silva. State of the Art in Surface Reconstruction from Point Clouds. In *Eurographics 2014 - State of the Art Reports*. The Eurographics Association, 2014.

[CCC+08] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.

[CGD11] C. Canaan, M. S. Garai, and M. Daya. Popular sorting algorithms. *World Applied Programming*, 1(1):62–71, April 2011.

[CGY04] Frédéric Cazals, Joachim Giesen, and Mariette Yvinec. Delaunay Triangulation Based Surface Reconstruction : a short survey. Technical Report RR-5394, INRIA, November 2004.

[CWL+08] Z.-Q. Cheng, Y.-Z. Wang, B. Li, K. Xu, G. Dang, and S.-Y. Jin. A Survey of Methods for Moving Least Squares Surfaces. In *IEEE/ EG Symposium on Volume and Point-Based Graphics*. The Eurographics Association, 2008.

[HDD+92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 71–78, New York, NY, USA, 1992. ACM.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[Kaz05] Michael Kazhdan. Reconstruction of solid models from oriented point sets. In *Proceedings of the Third Eurographics Symposium on Geometry Processing*, SGP '05, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association.

[KBH06] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP '06, pages 61–70, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[Knu73] Donald Ervin Knuth. *The art of computer programming - Sorting and searching*, volume 3 of *Addison-Wesley series in computer science and information processing*, chapter Sorting by Merging, pages 159–169. Addison-Wesley Professional, 1973.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[MJG16] Somshubra Majumdar, Ishaan Jain, and Aruna Gawade. Parallel quick sort using thread pool pattern. *International Journal of Computer Applications*, 136(7):36–41, February 2016.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Rai09] Günther Raidl. Algorithmen und datenstrukturen. Vorlesungsskript, Institut für Computergraphik und Algorithmen, Technische Univerität Wien, 2009.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[SA78] Robert Sedgewick and Robert Ashenhurst. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, Octobre 1978.

[Sch15] Markus Schuetz. Potree: rendering large point clouds in web browsers. Master's thesis, Fakultät für Informatik der Technischen Universität Wien, Wien, 2015.

[Sin69] Richard Singleton. Algorithm 347: an efficient algorithm for sorting with minimal storage. *Communications of the ACM*, 12(3):185–186, March 1969.

[ST13] Christian Siebert and Jesper Larsson Träff. Perfectly load-balanced, optimal, stable, parallel merge. *CoRR*, abs/1303.4312, November 2013.

[Sta] The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`.

[Sze11] Richard Szeliski. *Computer vision Algorithms and Applications*, chapter Structure from motion, pages 345–376. Texts in computer science. Springer, London, 2011.

[WS06] Michael Wimmer and Claus Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'06, pages 129–137, Aire-la-Ville, Switzerland, 2006. Eurographics Association.