UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

PARALLEL MESH GENERATION ALGORITHMS

Diploma Thesis

GALANIS KONSTANTINOS ORESTIS VASILEIOS

Supervisor: Michael Vassilakopoulos

June 2024

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

PARALLEL MESH GENERATION ALGORITHMS

Diploma Thesis

GALANIS KONSTANTINOS ORESTIS VASILEIOS

Supervisor: Michael Vassilakopoulos

June 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΠΑΡΑΛΛΗΛΟΙ ΑΛΓΟΡΙΘΜΟΙ ΔΗΜΙΟΥΡΓΙΑΣ ΠΛΕΓΜΑΤΟΣ

Διπλωματική Εργασία

ΓΑΛΑΝΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΟΡΕΣΤΗΣ ΒΑΣΙΛΕΙΟΣ

Επιβλέπων: ΒΑΣΙΛΑΚΟΠΟΥΛΟΣ ΜΙΧΑΗΛ

Ιούνιος 2024

Approved by the Examination Committee:

Supervisor **Michael Vassilakopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Athanasios Fevgas**
Laboratory Teaching Staff, Department of Electrical and Computer Engineering, University of Thessaly

Member **George Thanos**
Laboratory Teaching Staff, Department of Electrical and Computer Engineering, University of Thessaly

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

GALANIS KONSTANTINOS ORESTIS VASILEIOS

Diploma Thesis

PARALLEL MESH GENERATION ALGORITHMS

GALANIS KONSTANTINOS ORESTIS VASILEIOS

Abstract

This thesis explores the domain of Computational Geometry with a focus on developing efficient parallel algorithms for mesh generation, specifically through the application of the Bowyer-Watson algorithm for Delaunay Triangulation. The primary objective of this research is to enhance the performance and scalability of mesh generation processes in computational simulations and graphics by leveraging parallel computing architectures.

The methodology adopted involves the modification and optimization of the traditional Bowyer-Watson algorithm to exploit the parallel processing capabilities of modern GPUs and multi-core CPUs. This involved redesigning the algorithm's core components to reduce computational dependencies and increase the concurrency of operations. The CUDA programming model in python was utilized to implement these modifications, allowing for substantial data processing in parallel.

The main steps of the research included algorithmic redesign, implementation of the modified algorithm in CUDA, and a series of benchmarks comparing the performance of the parallelized algorithm against traditional sequential approaches. Various scenarios were tested to assess scalability and efficiency in generating large-scale meshes.

The results obtained demonstrate a significant improvement in processing times, with the parallel algorithm outperforming the sequential version by a factor of up to ten times, depending on the complexity and size of the input data. These findings highlight the potential of parallel processing techniques in the field of mesh generation and provide a foundation for further research into more complex geometries and higher-dimensional triangulation.

This thesis contributes to the field of computational geometry by demonstrating how parallel computing techniques can be effectively applied to complex algorithmic problems, significantly improving performance and opening new avenues for future research in graphics and simulation technologies.

Keywords:

Computational Geometry, Parallel Algorithms, Mesh Generation, Delaunay Triangulation, Bowyer-Watson Algorithm, GPU Computing, CUDA Programming, High Performance Computing, Scalability, Numerical Simulation

Διπλωματική Εργασία

ΠΑΡΑΛΛΗΛΟΙ ΑΛΓΟΡΙΘΜΟΙ ΔΗΜΙΟΥΡΓΙΑΣ ΠΛΕΓΜΑΤΟΣ

ΓΑΛΑΝΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΟΡΕΣΤΗΣ ΒΑΣΙΛΕΙΟΣ

Περίληψη

Αυτή η διπλωματική διερευνά τον τομέα της Υπολογιστικής Γεωμετρίας με επίκεντρο την ανάπτυξη αποδοτικών παράλληλων αλγορίθμων για τη δημιουργία πλεγμάτων, ειδικότερα μέσω της εφαρμογής του αλγορίθμου Bowyer-Watson για την Delauny Triangulation. Το κύριο αντικείμενο αυτής της έρευνας είναι η ενίσχυση της απόδοσης και κλιμάκωσης των διαδικασιών δημιουργίας πλεγμάτων σε υπολογιστικές προσομοιώσεις και γραφικά, εκμεταλλευόμενη τις αρχιτεκτονικές παράλληλης επεξεργασίας.

Η μεθοδολογία που υιοθετήθηκε περιλαμβάνει την τροποποίηση και βελτιστοποίηση του παραδοσιακού αλγορίθμου Bowyer-Watson για να αξιοποιηθούν οι δυνατότητες παράλληλης επεξεργασίας των σύγχρονων GPUs. Αυτό περιελάμβανε την ανασχεδίαση των βασικών στοιχείων του αλγορίθμου για τη μείωση των υπολογιστικών εξαρτήσεων και την αύξηση της ταυτόχρονης λειτουργίας των λειτουργιών. Το μοντέλο προγραμματισμού CUDA σε python χρησιμοποιήθηκε για την υλοποίηση αυτών των τροποποιήσεων, επιτρέποντας την ταυτόχρονη επεξεργασία σημαντικών όγκων δεδομένων.

Τα κύρια βήματα της έρευνας περιελάμβαναν την ανασχεδίαση του αλγορίθμου, την υλοποίηση του τροποποιημένου αλγορίθμου σε CUDA και μια σειρά από benchmarks που σύγκριναν την απόδοση του παραλληλοποιημένου αλγορίθμου έναντι των παραδοσιακών σειριακών προσεγγίσεων. Διάφορα σενάρια δοκιμάστηκαν για να αξιολογηθούν η κλιμακωσιμότητα και η αποδοτικότητα στη δημιουργία πλεγμάτων μεγάλης κλίμακας.

Τα αποτελέσματα που λήφθηκαν αποδεικνύουν σημαντική βελτίωση στους χρόνους επεξεργασίας, με τον παράλληλο αλγόριθμο να υπερέχει της σειριακής έκδοσης κατά έναν παράγοντα έως και δέκα φορές, ανάλογα με την πολυπλοκότητα και το μέγεθος των εισερχόμενων δεδομένων. Αυτά τα ευρήματα υπογραμμίζουν τη δυναμική των τεχνικών παράλληλης επεξεργασίας στον τομέα της δημιουργίας πλεγμάτων και παρέχουν μια βάση για περαιτέρω έρευνα σε πιο περίπλοκες γεωμετρίες και υψηλότερης διάστασης τριγωνοποίηση.

Αυτή η διπλωματική συμβάλλει στον τομέα της υπολογιστικής γεωμετρίας δείχνοντας πώς οι τεχνικές της παράλληλης υπολογιστικής μπορούν να εφαρμοστούν αποτελεσματικά

σε σύνθετα αλγοριθμικά προβλήματα, βελτιώνοντας σημαντικά την απόδοση και ανοίγοντας νέους δρόμους για μελλοντικές έρευνες σε τεχνολογίες γραφικών και προσομοιώσεων.

Λέξεις-κλειδιά:

Υπολογιστική Γεωμετρία, Παράλληλοι Αλγόριθμοι, Δημιουργία Πλεγμάτων, Delauny Triangulation, Αλγόριθμος Bowyer-Watson, Υπολογισμός GPU, Προγραμματισμός CUDA, Υπολογιστική Υψηλών Επιδόσεων, Κλιμακωσιμότητα, Αριθμητική Προσομοίωση

Table of contents

List of figures

Chapter 1

INTRODUCTION

Chapter 1 introduces the field of computational geometry, emphasizing its critical role in solving spatial problems through algorithmic approaches. It highlights mesh generation as a fundamental application essential for numerical simulations in areas like fluid dynamics and aerospace engineering. The chapter discusses the main challenges in mesh generation, such as maintaining mesh quality, achieving scalability, enhancing computational performance, and ensuring interoperability across different platforms.

The thesis focuses on enhancing the Bowyer-Watson algorithm for Delaunay triangulation using parallel computing techniques. It aims to improve mesh quality, scalability, and performance by leveraging modern GPUs and multi-core CPUs. These advancements are rigorously evaluated and integrated into standard simulation software. The subsequent sections will delve deeper into Delaunay triangulation, its computational methods, applications in computer vision, and the detailed evaluation of the proposed algorithms, concluding with overall findings and acknowledgments.

1.1 Computational Geometry

Computational geometry is a specialized area within computer science focused on developing algorithms that address geometric problems. This field merges theoretical foundations with practical applications across diverse scientific and engineering disciplines. Essentially, computational geometry involves creating and analyzing algorithms to solve problems related to the spatial arrangement and interaction of shapes and objects. These problems typically involve entities like points, lines, polygons, and polyhedra in two and three dimensions, and

can extend to higher dimensions in more complex scenarios.

A key feature of computational geometry is its systematic approach to solving spatial issues. These issues may include tasks such as identifying the nearest pair of points in a dataset, calculating the convex hull of a point set, determining intersections between geometric shapes, and optimizing spatial configurations for various uses. The techniques developed in this field are both theoretically sound and practically efficient, making them highly relevant for addressing real-world challenges.

Key Areas of Computational Geometry

Geometric Data Structures: These are fundamental for efficiently storing and querying spatial information. Examples include k-d trees, range trees, and segment trees, which support operations like point location, range searching, and nearest neighbor search.

Geometric Algorithms: These algorithms solve problems such as convex hull computation, Voronoi diagrams, Delaunay triangulations, and polygon triangulations. Each algorithm is designed to handle specific types of geometric data and operations.

Computational Topology: This area extends computational geometry to study the properties of geometric shapes that are preserved under continuous deformations. It includes the study of topological spaces, simplicial complexes, and Morse theory.

Intersection Problems: These involve determining if and where geometric shapes intersect. Applications include collision detection in computer graphics, geographical information systems (GIS), and robotics.

Proximity Problems: These focus on finding closest pairs or nearest neighbors among a set of points. Such problems are crucial in areas like clustering analysis, network design, and spatial indexing.

Geometric Optimization: This involves optimizing geometric configurations under certain constraints. Examples include the traveling salesman problem in a geometric context and optimizing network layouts.

Applications of Computational Geometry

Computer Graphics:

- Hidden surface removal
- Determining the geometric primitive pointed to with the mouse in a collection of objects being displayed
- Assessing the collision between geometric objects
- Rendering objects in programs such as Blender
- Mesh generation for rendering based on Delaunay triangulation
- Techniques such as rasterization, ray casting, and ray tracing
- Physical simulations like finite element analysis or computational fluid dynamics

Robotics:

- Path planning, relying on visibility
- Positioning and fixing parts (like arms) of a robot so it can work on a part being manufactured
- Motion planning, collision detection, and path optimization
- Robot grasping and manipulation
- Collision-free pathways using algorithms like the Visibility Graph Algorithm and the Rapidly-exploring Random Tree (RRT) algorithm
- Determining optimal grasping locations and orientations using algorithms like the Convex Hull algorithm and the Grasp Quality Metrics algorithm

Artificial Intelligence:

- A* pathfinding to determine the shortest route between two points
- Voronoi diagrams for path clearance
- Computer vision applications like facial recognition and self-driving cars using convex hulls

Networking:

- Shortest-path algorithms like Dijkstra's for directing router traffic
- Voronoi diagrams for mapping the maximum capacity of wireless networks

Linear Programming:

- Solving problems with the simplex algorithm by constructing feasible solutions
- Optimizing public transport routes for cost and time efficiency

Geographic Information Systems (GIS):

- Storing and efficiently extracting geographic data
- Digitizing paper maps to determine boundaries
- Locating the nearest public phone
- Map overlay
- Analyzing geographic data using convex hulls and Voronoi diagrams
- Visualization of topography data using Delaunay triangulation

Computer-Aided Design:

- Geometric modeling to represent real-world objects
- Intersection and union of objects
- Simulating heat emission of a printed circuit board via meshing
- Simulating stress calculation at every point of a structure via meshing

Computer-Aided Manufacturing:

- Machining along a tour that touches a set of points
- Designing a mold so that the manufactured product can be removed by a single translation

Electronic Design Automation:

- Packing a set of chips onto a motherboard while accommodating pairwise connections between chips

- Partitioning an orthogonal polygon into approximately the minimum number of rectangles
- Forming a simple polygon of smallest perimeter with a given set of points as its vertices

Wireless Network Design:

- Placing a minimum number of sufficiently powered antennas to cover every point of every region of interest

Pattern Recognition:

- Matching the image of a symbol against a collection of stored symbols for identification

Machine Learning:

- Partitioning data points into clusters
- Determining outliers

Data Analysis:

- Storing data points to efficiently answer range queries

Data Science:

- Finding approximate nearest neighbors
- Dimension reduction

Computational Learning Theory:

- Concepts like ϵ-nets and VC-dimension

Guarding:

- Surveying a museum using a minimum number of guards

Molecular Modeling:

- Computing the outer surface of a homogeneous mixture formed by molecules, wherein molecules are represented by balls with a few pairwise intersections

Efficient Communication:

- Level-of-detail based information sharing

Computational Statistics:

- Determining how central a given point is inside a cloud of points

Rigidity Analysis:

- Given a discrete configuration of a set of points, positioning them on a plane while satisfying a set of pairwise distances
- Determining whether the set of solutions is unique

Crystallography:

- Measuring the similarity between two arbitrary crystal structures

Mesh Generation

A key application of computational geometry is mesh generation, essential for numerical simulations in fields like fluid dynamics, structural analysis, and aerospace engineering. Mesh generation involves subdividing a geometric space into simple, non-overlapping shapes (such as triangles in 2D or tetrahedrons in 3D) to facilitate the resolution of physical or mathematical problems. The quality of these meshes significantly impacts the accuracy, efficiency, and stability of simulation results.

Mesh generation is the process of creating a mesh, which subdivides a continuous geometric space into discrete geometric and topological cells. These cells often form a simplicial complex and partition the geometric input domain. They serve as discrete local approximations of the larger domain. Computer algorithms, sometimes aided by human input through a GUI, generate these meshes. The objective is to create a mesh that accurately represents the input domain geometry, with high-quality cells that are well-shaped, and not so numerous as to render subsequent calculations impractical. The mesh should also have finer elements in regions critical for further computations.

Meshes are used for rendering on computer screens and for physical simulations like finite element analysis or computational fluid dynamics. Simple cells, such as triangles, are used because operations like finite element calculations or ray tracing can be performed on these

shapes, whereas complex spaces and shapes like a roadway bridge cannot be directly computed. By performing calculations on each triangle and determining the interactions between them, we can simulate the strength of the bridge or visualize it on a computer screen.

There is a distinction between structured and unstructured meshing. Structured meshes form a regular lattice with implied connectivity between elements, whereas unstructured meshes have irregular connections, allowing more complex domains to be represented. This discussion primarily focuses on unstructured meshes. Unlike point set triangulation, meshing allows the addition of vertices not present in the input, aiming to represent the shape accurately with as few triangles as possible, where the shape of individual triangles is not a primary concern. Meshes are also used in computer graphics for rendering textures and realistic lighting conditions.

Many mesh generation software programs are integrated with CAD systems for input and simulation software for output. The input can vary greatly, commonly including solid modeling, geometric modeling, NURBS, B-rep, STL, or point clouds.

Types of Geometric Domains:

- **Simple Polygons:** Includes both boundary and interior.
- **Polygons with Holes:** A simple polygon minus the interiors of some other simple polygons.
- **Multiple Domains:** Allows internal boundaries and can be any planar straight-line graph.
- **Curved Domains:** Allow sides that are algebraic curves such as splines.
- **Simple Polyhedra:** Topologically equivalent to a ball.
- **General Polyhedra:** May be multiply connected or have cavities.
- **Multiple Polyhedral Domains:** General polyhedra with internal boundaries.

Types of Meshes:

- **Structured Mesh:** All interior vertices are topologically alike, forming an induced subgraph of an infinite periodic graph.
- **Unstructured Mesh:** Vertices may have arbitrarily varying local neighborhoods.

- **Block-Structured or Hybrid Mesh:** Formed by combining small structured meshes in an overall unstructured pattern.

In structured meshing, typically quadrilaterals are used in 2D, and hexahedra in 3D, while unstructured meshes use triangles in 2D and tetrahedra in 3D. Subdividing elements can convert between these shapes.

Applications of Mesh Generation:

- **Rendering:** Creating polygonal mesh representations for objects in computer graphics.
- **Finite Element Analysis:** Simulating physical phenomena such as stress, heat transfer, and fluid flow.
- **Computational Fluid Dynamics:** Simulating the behavior of fluids and their interactions with surfaces.
- **Geographic Information Systems:** Compact representations of terrain data.
- **Computer-Aided Design and Manufacturing:** Geometric modeling, simulating heat emission, stress calculations, and machining paths.
- **Crystallography:** Measuring the similarity between arbitrary crystal structures.

Mesh generation is an essential aspect of computational geometry, enabling precise and efficient numerical simulations and graphical renderings in diverse fields. By continuously evolving, mesh generation techniques are paving the way for more complex and detailed analyses and visualizations.

However, the field faces several challenges:

- **Quality of Mesh:** Creating high-quality meshes that conform to intricate geometric configurations while ensuring numerical stability is a continual challenge. Poor mesh quality can undermine the accuracy and computational efficiency of simulations.
- **Scalability:** The increasing computational demands, especially in large-scale simulations involving millions of elements, require scalable mesh generation algorithms. Traditional serial algorithms often fail to efficiently handle such extensive datasets.
- **Performance:** The high computational demands of mesh generation and refinement necessitate substantial processing power, which can lead to performance bottlenecks, particularly in real-time applications.

- **Interoperability:** Meshes must be designed to be compatible with a wide range of simulation software and hardware platforms. This requires flexible mesh generation techniques that can accommodate different requirements and limitations.

1.2 Thesis subject

The subject of my diploma thesis is Parallel Mesh Generation Algorithms, with a specific focus on enhancing the Bowyer-Watson algorithm for Delaunay Triangulation using parallel computing techniques. This involves optimizing the traditional algorithm to leverage the computational power of modern GPUs and multi-core CPUs, aiming to improve the efficiency, scalability, and quality of mesh generation processes in computational simulations and graphics. This adaptation is pivotal for managing the computational complexity and size demands of modern mesh generation tasks, which are critical for high-fidelity numerical simulations in engineering and science. By implementing this algorithm o the thesis aims to:

- **Improve Mesh Quality:** Develop techniques to enhance the geometric fidelity and adaptability of meshes to complex surfaces, thereby ensuring higher accuracy in simulations.
- **Increase Scalability:** Enable the processing of large-scale simulations by reducing the time complexity of the triangulation process from quadratic to near-linear, as supported by parallel processing.
- **Enhance Performance:** Achieve significant reductions in computation time, demonstrating the algorithm's efficiency through comprehensive benchmarks against existing serial and parallel triangulation methods.

These advancements contribute to the broader field of computational geometry by providing robust, scalable solutions for real-time and high-resolution applications. Furthermore, the integration of these parallel algorithms into existing simulation frameworks can potentially transform practices in industries relying heavily on computational simulations.

1.2.1 Contribution

The contributions of this thesis are summarized as follows:

1. **Systems Studied:** Investigated the current limitations and bottlenecks in traditional mesh generation systems, especially focusing on computational geometry applications within fluid dynamics, aerospace, and structural engineering. The study emphasized the need for scalable and efficient algorithms to handle large-scale simulations.

2. **Three Calculation Algorithms Implemented:**

    - *Parallel Bowyer-Watson Algorithm:* Modified the traditional Bowyer-Watson algorithm for Delaunay triangulation to enable parallel computation, optimizing it for execution on modern GPUs and multi-core CPUs.
    - *Mesh Quality Optimization Technique:* Developed methods to automatically optimize mesh quality during the triangulation process, ensuring that the meshes are well-suited for numerical simulations.
    - *Real-time Mesh Generation Framework:* Implemented a dynamic mesh generation system that allows for real-time updates of the mesh based on simulation feedback, enhancing the adaptability of the computational models.

3. **Performance Evaluation:** The performance of the algorithms was rigorously evaluated through benchmark tests against traditional sequential methods. Results demonstrated that the parallel Bowyer-Watson algorithm improved computation speed by up to tenfold while maintaining or enhancing mesh quality and stability.

These actions and solutions directly address the critical issues previously outlined, providing substantial advancements in mesh generation technology that can significantly benefit computational simulations across various disciplines.

1.2.2 Related Work

Mesh generation has been extensively studied, with significant contributions enhancing its techniques and applications. One of the foundational methods is the Delaunay triangulation, introduced by Lawson [1], which ensures that no point is inside the circumcircle of any triangle, leading to well-shaped and stable meshes. Delaunay triangulation has been widely adopted due to its desirable properties such as maximizing the minimum angle of triangles, which helps in avoiding skinny triangles that can degrade the quality of numerical simulations.

Ruppert's algorithm [2] is another pivotal work that builds on Delaunay triangulation by providing a Delaunay refinement approach. This algorithm guarantees mesh quality by maintaining minimum angle constraints, ensuring that all angles are above a specified threshold. This method has been instrumental in generating high-quality meshes for finite element analysis and other applications requiring precision.

The advancing front method, developed by Lo [3], introduces a different approach by building the mesh incrementally from the boundary towards the interior. This technique allows for better conformity to complex geometries and has been particularly useful in applications where the boundary shape is intricate. Advancing front methods have been further enhanced to handle three-dimensional meshing, making them versatile for various engineering problems.

More recent advancements include hybrid meshing techniques, such as those proposed by Löhner [4], which combine structured and unstructured elements. Hybrid meshes leverage the regularity and efficiency of structured meshes while incorporating the flexibility of unstructured meshes to handle complex geometries. This approach optimizes computational efficiency and geometric fidelity, making it suitable for simulations that require both accuracy and speed.

Adaptive mesh refinement (AMR) techniques, like those by Berger and Oliger [5], dynamically adjust the mesh resolution based on solution requirements. AMR methods refine the mesh in regions with high gradients or complex features and coarsen it in less critical areas. This adaptability improves both the accuracy and computational efficiency of simulations, making AMR a valuable tool in fields such as fluid dynamics and structural analysis.

Mesh generation has also seen significant contributions from the development of specialized algorithms for specific applications. For example, the work by Shewchuk [6] on generating unstructured tetrahedral meshes has been widely adopted for three-dimensional simulations. His algorithm, known for its robustness and ability to handle complex geometries, has become a standard in the field.

In addition to algorithmic advancements, software tools play a crucial role in mesh generation. Notable examples include Gmsh [7], an open-source 3D finite element mesh generator with built-in pre- and post-processing facilities, and TetGen [8], a program for generating tetrahedral meshes. These tools provide robust implementations of various mesh generation algorithms and are widely used in both academic research and industry.

The integration of machine learning techniques into mesh generation is an emerging trend. Researchers like Wu et al. [9] have explored using neural networks to automate and improve the mesh generation process. These methods aim to reduce the manual effort involved in mesh generation and enhance the quality of the meshes produced.

Overall, the field of mesh generation has evolved significantly, from foundational algorithms like Delaunay triangulation and Voronoi diagrams to sophisticated techniques for adaptive and hybrid mesh generation. Ongoing research continues to push the boundaries, exploring new methodologies and technologies to further enhance the quality and efficiency of mesh generation.

1.3 Organization of the Volume

This document is organized as follows: Regular Triangulation is described in Chapter 2. Delaunay Triangulation and Tetrahedralization are developed in Chapter 3, whereas Chapter 4 explains the Bowyer-Watson algorithm. In Chapter 5, the serial implementation of the algorithm is detailed, and in Chapter 6, the parallel implementation is discussed. Chapter 7 provides an overview of the environment. Chapter 8 focuses on the performance calculations, and Chapter 9 compares the developed algorithm with another thesis that served as inspiration. Finally, the conclusions are presented in the last chapter.

Chapter 2

REGULAR TRIANGULATION

Chapter 2 delves into the concept of regular triangulation, a generalization of the well-known Delaunay triangulation. The chapter begins with the fundamental definitions required to understand triangulations in d dimensions, including concepts such as convex hulls, affine independence, simplices, and simplicial complexes. It then introduces the idea of regular triangulation, where each point is associated with a weight, leading to potentially different triangulations compared to unweighted sets. Key definitions related to weighted points, power distances, and orthogonal points are provided to build a comprehensive understanding of regular triangulations.

The chapter also covers the construction and properties of regular triangulations, including the criteria for global and local regularity. Furthermore, the concept of Voronoi diagrams is introduced, highlighting its relationship with Delaunay triangulations and the unique properties of Voronoi regions. By the end of the chapter, readers will have a solid grasp of both the theoretical foundations and practical applications of regular and weighted triangulations, essential for advanced computational geometry tasks.

2.1 Triangulations of Point Sets

Most definitions will be provided in d dimensions, although the code will only handle two and three dimensions. This approach is taken because the Bowyer-Watson algorithm operates in higher dimensions, allowing the Voronoi diagram to be extracted as the dual in dimensions greater than three. Additionally, this method avoids the need to separately define terms for two and three dimensions.

Definition 1.1

The **convex hull** of a point set $A \subset \mathbb{R}^d$ is the smallest convex set that contains A, and is denoted $H(A)$. A convex set in $\mathbb{R}^d$ is a set of points such that given any pair of points in the set, the straight line segment joining the pair of points is fully contained in the set. See Figure 2.1.a.

Definition 1.2

Points $x_1, \ldots, x_n \in \mathbb{R}^d$ are **affinely independent** if any linear combination $\lambda_1 x_1 + \ldots + \lambda_n x_n = 0$ with $\lambda_1 + \ldots + \lambda_n = 0$ must have $\lambda_1 = \ldots = \lambda_n = 0$.

Definition 1.3

A k-**simplex** is the convex hull of $k+1$ affinely independent points in $\mathbb{R}^d$. These points are referred to as the **vertices** of the simplex.

From the definitions, $\mathbb{R}^d$ can contain at most a d-simplex. When we create programs, we shall be dealing with $\mathbb{R}^2$ and $\mathbb{R}^3$ so the biggest simplex we shall see is a 3-simplex, namely a tetrahedron.

Definition 1.4

Let σ and τ be simplices in $\mathbb{R}^d$, with vertices A and B respectively. Then we say that τ is a **face** of σ if $B \subset A$. If τ is a k-simplex we say it is a k-face of σ.

Definition 1.5

A finite collection K of simplices in $\mathbb{R}^d$ is said to be a **simplicial complex** if the following two conditions are satisfied:

1. If σ belongs to K then every face of σ belongs to K.

2. If σ and τ belong to K then either σ and τ are disjoint or they intersect along a common face of σ and τ.

If the biggest simplex in K is a k-simplex, then K is said to be a simplicial k-complex.

Definition 1.6. We shall use a slightly modified definition from De Loera et al. [10]. A *triangulation* of a point set $A \in \mathbb{R}^d$ is a simplicial d-complex K with vertices A such that the union of all d-simplices in K is $H(A)$. See Figure 2.1.b.

Figure 2.1: (a) Convex Hull of 8 points. (b) Triangulation of 8 points [11].

2.2 Regular Triangulation

The regular triangulation (RT) is a generalization of the well-known Delaunay triangulation (DT). In a regular triangulation, each point is associated with a real number, known as the weight of the point. If the weights of all points are equal, the regular triangulation is identical to the Delaunay triangulation of the same set of points; otherwise, the triangulations can differ. We will begin with a few basic definitions and then provide the exact definition of the regular triangulation.

Triangulation Given a set of points S in $\mathbb{E}^3$, the triangulation $T(S)$ of this set of points is a set of tetrahedra such that:

- A point $p \in \mathbb{E}^3$ is a vertex of a tetrahedron in $T(S)$ only if $p \in S$.
- The intersection of two tetrahedra of $T(S)$ is either empty or it is a shared face, a shared edge or a shared vertex.
- The union of all tetrahedra in $T(S)$ entirely fulfills the convex hull of S.

Note that in this definition of $T(S)$ we do not require each $p \in S$ to be a vertex of $T(S)$.

Weighted point A point $p \in \mathbb{E}^3$ with an associated weight $w_p \in \mathbb{R}$ is called a weighted point. If the weight w_p is non-negative, then p can be interpreted as a sphere centered at the point p with a radius $\sqrt{w_p}$.

Power distance A power distance of a weighted point p from a point $x \in \mathbb{E}^3$ (no matter whether x is weighted or unweighted) is defined as

$$\pi_p(x) = |\mathbf{p} - \mathbf{x}|^2 - w_p, \tag{2.1}$$

where $|\mathbf{px}|$ denotes the Euclidean distance between the points p and x. The power distance $\pi_p(x)$ can be interpreted as a square of length of a tangent from the point x to a sphere centered at p with the radius $\sqrt{w_p}$ (if x lies outside this sphere), see Fig. 2.2.a.

Orthogonal points Two weighted points p and q are said to be orthogonal if $|\mathbf{pq}|^2 = w_p + w_q$, see Fig. 2.2.b.

Orthogonal center Let a, b, c, d be non-coplanar weighted points. A weighted point z is an orthogonal center of a tetrahedron $abcd$ if z is orthogonal to the points a, b, c, d.

Global regularity Let z be the orthogonal center of a tetrahedron $abcd$. The tetrahedron $abcd$ is globally regular with respect to a set of weighted points S if $\pi_z(p) > w_p$ for each point $p \in S - \{a, b, c, d\}$.

Regular triangulation A triangulation $T(S)$ is a regular triangulation of S (*RT(S)*) if each tetrahedron in $T(S)$ is globally regular with respect to S.

Redundant point A point p, $p \in S$, is called a redundant point if no globally regular tetrahedron $pabc$ exists in $RT(S)$. Redundant points are not vertices of any tetrahedron in $RT(S)$, therefore, the vertex set of $RT(S)$ is generally only a subset of S.

Local regularity Let $abcd$ and $abce$ be two tetrahedra with a common face abc and z be the orthogonal center of the tetrahedron $abcd$. The common face abc is said to be locally regular if $\pi_z(e) > w_e$.

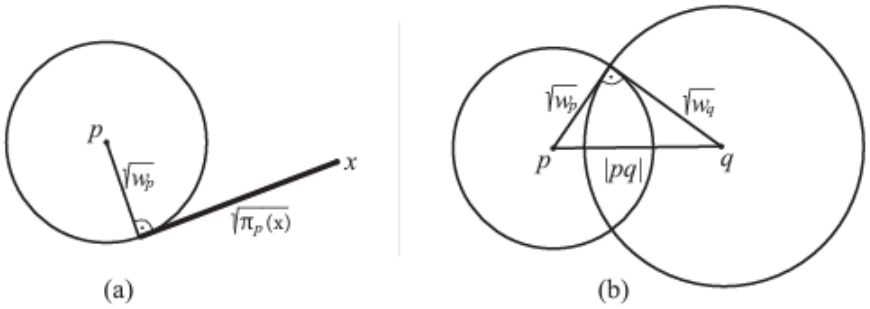

Figure 2.2: (a) The power distance $\pi_p(x)$. (b) Two orthogonal weighted points [11].

2.3 Voronoi Diagrams

We will use a slightly modified and generalized definition from Liebling and Pourmir [47].

Definition 1.13. Consider a point set $A \subset \mathbb{R}^d$. The *Voronoi region* R_x associated with the point x in A is a possibly unbounded convex d-polytope which consists of those points in $\mathbb{R}^d$ whose distance to x is not greater than their distance to any other point of A.

Definition 1.14. The *Voronoi diagram* $V(A)$ induced by A is a decomposition of $\mathbb{R}^d$ into the Voronoi regions associated with the points of A. $V(A)$ will often be referred to as the Voronoi diagram of A.

Remark. Notice that a Delaunay triangulation of a point set A is not always unique, while the Voronoi diagram of a point set A is unique.

The formal definition of a cell of EVD is very similar to the definition of a power cell in PD. A sphere is used as a generator instead of a weighted point, and the power distance between a point and a weighted point is replaced by a Euclidean distance between a point and a spherical surface.

EVD cell Given a set of spheres S in $\mathbb{R}^3$. For each sphere s in S with a radius r_s, its EVD cell is defined as:

$$EVD\ \text{cell}(s) = \{x \in \mathbb{R}^3 \setminus S \mid |x - s| - r_s \leq |x' - s| - r'_s\}$$

where $|x - s|$ denotes the Euclidean distance between the point x and the center of s.

The faces, edges, and vertices of Euclidean Voronoi Diagrams (EVD) can be described similarly to the intersections of cells in power diagrams. However, there is a significant distinction: in power diagrams, the bisector of two generators is a plane, whereas, in EVD, it is a hyperboloid (or a segment of a hyperboloid). As a result, the geometry of EVD cells is non-linear. A face is a connected portion of a hyperboloid, and an edge is a conic section. Additionally, faces can have ”holes,” and edges can be enclosed ellipses.

Chapter 3

DELAUNY TRIANGULATION

A Delaunay Triangulation is a mesh of triangles where each triangle adheres to the Delaunay condition. This condition specifies that the circumcircle of any triangle in the mesh should not contain any vertices from other triangles. Developed by Boris Nikolaevich Delone in 1934, Delaunay Triangulation is extensively utilized in computational geometry. Essentially, it forms a network of triangles such that no vertex of any other triangle lies within the circumcircle of each triangle. This principle extends to edges and tetrahedra in three dimensions. The most common methods for constructing a Delaunay Triangulation include the Lawson method [12], Bowyer method [13], and Watson method [14]. Any simplex (triangle, tetrahedron, edge) that satisfies the circumcircle property is referred to as a Delaunay simplex, and the collection of all such simplices forms the convex hull. Each Delaunay Triangulation has a dual graph known as a Voronoi Diagram, created using the circumcenters of the Delaunay triangles.

Chapter 3 explores Delaunay Triangulation in detail, starting with its basic definition and historical background. The chapter discusses various applications of Delaunay Triangulation, including its use in stereo data, video compression, terrain modeling, networking, reconstruction, protein folding, spatial clustering, and boundary detection. It provides a comprehensive overview of the properties and algorithms used for constructing Delaunay Triangulations, such as incremental algorithms, sweepline algorithms, and divide-and-conquer methods. Additionally, the chapter delves into the properties of Delaunay Triangulation, such as the local empty-circle property, the max-min angle property, and the uniqueness of the triangulation. By the end of the chapter, readers will have a thorough understanding of the principles, methods, and applications of Delaunay Triangulation in both two and three dimensions.

3.1 In depth

Delaunay Triangulation is a fundamental structure in computational geometry, with applications extending to various domains. Key applications include stereo data [15], video compression [16], terrain modeling [17, 18], networking [19, 20], reconstruction [21], protein folding [22, 23], spatial clustering [24], and boundary detection [25], among others. This structure connects the nearest neighbors within a given neighborhood, making it suitable for modeling collision detection problems. Delaunay Triangulation is efficient, as each point in the triangulation represents an object in the environment, and the connectivity between points is maintained to uphold the Delaunay structure. However, the movement of even a single point in the triangulation can alter its structure, necessitating updates to the Delaunay Triangulation to accurately model dynamic phenomena. Consequently, surveys such as those by [26, 27, 28] discuss methods for updating Delaunay Triangulations.

Here are some basic definitions:

Definition 1.10. A *Delaunay triangulation* $DT(A)$ of a point set $A \subset \mathbb{R}^2$ is a triangulation of A such that no point of A lies inside the circumcircle of a triangle in $DT(A)$. A triangle having no points in the interior of its circumcircle is often referred to as having the *empty circle property*.

We can generalise the above definition to a point set $A \subset \mathbb{R}^d$. But first we need the notion of a d-dimensional disc, or d-disc.

Definition 1.11. A d-dimensional open disc, or simply an *open d-disc*, of radius r and centre c is the set of points

$$\{x \in \mathbb{R}^d : d(x, c) < r\}$$

where $d(x, c)$ is the Euclidean distance between x and c.

Remark. Since d-simplices consist of $d + 1$ vertices, the vertices of a d-simplex σ define an open d-disc such that the boundary of the disc passes through all the vertices of σ. The boundary of this open d-disc is a $(d - 1)$-hypersphere which circumscribes σ.

Definition 1.12. A *Delaunay triangulation* $DT(A)$ of a point set $A \subset \mathbb{R}^d$ is a triangulation of A such that no point of A lies in an open d-disc whose boundary circumscribes a d-simplex in $DT(A)$.

3.2 Triangulation

The primary objective of triangulation is to create a mesh, which is a widely used method for representing continuous surfaces. The input for a triangulation process is a set of points, and the output is a set of triangles or linked edges that do not overlap [29]. Constructing a triangulation involves various challenges, but the initial point selection and point dispersion are generally not critical. Triangulation has several important applications, including Digital Terrain Modeling (DTM) Generation, Feature Surface Modeling, Computer Graphics, Scientific Visualization, Robotics, Computer Vision, Image Synthesis, Mathematics, and Natural Sciences. Numerous triangulation methods have been proposed, such as those by [30, 31, 32]; Greedy Triangulation [33], Triangulation by Garey [34], Radial Sweep [35, 36], and Delaunay Triangulation [37].

Greedy Triangulation [33] is employed to triangulate simple polygons by connecting the closest points, resulting in the shortest edges for the given set of points. While it is not a Delaunay Triangulation, it serves as an approximation for the Minimum Weight Triangulation Problem.

The Radial Sweep method consists of four steps [35, 36]: (i) selecting the centroid in a set of points P and linking the centroid to every point in P; (ii) sorting and ordering the points by orientation and distance (excluding the centroid) and linking them in a star shape without crossing edges; (iii) adding triangles to form a convex shape (convex hull), connecting points both inside and outside; (iv) finally arranging the points based on their angles to produce a correct triangulation. For the set of points P shown in Figure 1a, a Delaunay Triangulation is created in Figure 1b. A convex hull for P is depicted in Figure 1c. Figure 1d shows the circumcircles, Figure 1e illustrates the centers of each circumcircle, and Figure 1f presents the Voronoi Diagram.

3.2.1 Delaunay Triangulation Algorithms

In Figure 3a, three new triangles are formed when a new point p is inserted inside an existing triangle. Conversely, if p is inserted on an edge, two new triangles are created, as illustrated in Figure 3b. Figure 3c shows two adjacent triangles, and the addition of a new point results in the creation of four new triangles, as depicted in Figure 3d. Since Delaunay

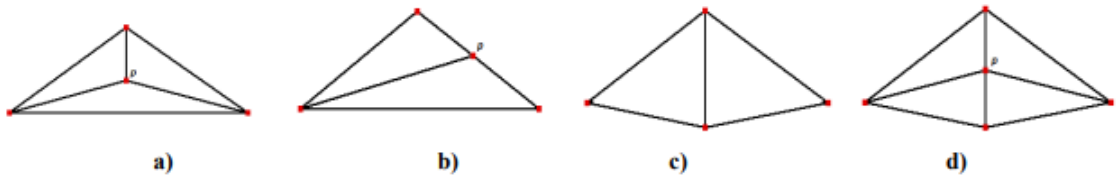

Figure 3.1: a) Inserting a Point P when it Fall into the Triangle. b) Inserting a Point P when it Fall into an Edge c) Two Triangles Sharing an Edge (Adjacent Edge) d) Inserting a Point P when it Fall into an Adjacent Edge [38]

triangulations do not have overlapping triangles, the new triangles replace the old ones.

- **Incremental Algorithms**
  - **Incremental Insertion:**
    1. **Initialization:** A triangle t is constructed to encompass all points in P, though the vertices of t are not included in P.
    2. **Triangulation:** The first point is inserted into the triangle t, forming three new triangles ($t1, t2, t3$), and t is replaced by these new triangles (as shown in Figure 3a). Inserting the second point can proceed in two ways:
       * (a) If the point is inside a triangle ($t1, t2, t3$), three new triangles are created, and the exterior triangle (ti; $1 \leq i \leq 3$) is removed.
       * (b) If the point lies on an edge, it is necessary to identify the triangle(s) involving that edge. If a single triangle is found, it is replaced by two new triangles (see Figure 3b). If two triangles are involved, they are replaced by four new triangles (each triangle is replaced by two new ones), as shown in Figures 3c and 3d. It is then necessary to check the circumcircle condition for each new triangle and repeat the process for inserting the second point [12].

       Alternatively, all triangles whose circumcircles contain the new point are found and deleted, creating a cavity. Finally, all vertices in the cavity are connected to the new point [13, 14].
    3. **Finalization:** Triangles that include the vertices of the initial triangle are removed.

- **Sweepline Algorithms**

  - **Plane Sweep Algorithm:** TA line sweeps the points, when a point is found by the line this is inserted in the triangulation. It only works in two-dimensions.

- **Incremental Construction Algorithms**

  - **Step-by-Step Algorithm:** This implementation leverages the unique properties of Delaunay Triangulation. In each step, a new Delaunay edge or triangle is added until all points in the set P are included. The process begins with identifying a Delaunay edge and then creating a new Delaunay triangle by searching for an appropriate point p [35].

  - **Grid Algorithm:** To identify a point near the center of the complete set of points, this point is designated as one. An initial edge is formed with the closest point, labeled as two. The process continues by creating the first triangle according to the right-hand rule, labeling it as three, and the subsequent point as four, and so on. Labels are removed once a point is fully connected or serves as the center of a star-shaped configuration [39]. An enhanced algorithm was developed utilizing a finer grid subdivision [40].

  - **Bowyer-Watson:** The Bowyer-Watson algorithm employs incrementalization, continually adding points and deleting any triangles whose circumcircles contain those points. For n points, this process may require between n^2 and $n\log(n)$ operations to complete the triangulation.

- **Local Improvement Algorithms**

  - **Flipping Algorithm:** Flipping is a straightforward method. Begin with any triangulation, and whenever non-Delaunay triangles are identified, flip one of their edges. Continue this process until all triangles satisfy the Delaunay condition.

- **Divide and Conquer Algorithms**

  - **Divide and Conquer:** This recursive method calculates Delaunay triangulation by dividing the area into two parts and computing the triangulation recursively for each part. The final step is to merge the two triangulations, which is the most complex part of the process [35].

– **Dewall Algorithm:** The name stands for Delaunay Wall. It is a Divide and Conquer algorithm that computes the simplices between subdivisions. In other words, the algorithm recursively merges to calculate the Delaunay Triangulation [41].

3.2.2 Properties of Delaunay Triangulation

- **Local Empty-Circle:** A circumcircle is a unique circle that passes through all the vertices of a triangle in a Delaunay Triangulation $DT(P)$. This circle does not contain any other points from the set P (see Figures 3.2.a and 3.2.b).

- **Max-Min Angle::** Introduced by Sibson in [42], for each quadrilateral in a Delaunay Triangulation $DT(P)$, there are two possible triangulations. The correct triangulation is the one that maximizes the smallest of the six internal angles (see Figures 3.2.c and 3.2.d).

- **Uniqueness:** Delaunay Triangulation $DT(P)$ for a set of points P is unique, except in the case of four co-circular points, known as a degenerate case. An example of this is a square (in two dimensions) or a cube (in three dimensions), where different divisions are possible, as shown in Figures 3.2.e and 3.2.f. A degenerate case occurs when four or more points lie on a circumcircle (in two dimensions) or five or more points lie on a circumsphere (in three dimensions).

- **Boundary:** The external edges of a Delaunay Triangulation $DT(P)$ form the convex hull of P. The Local Empty Property, illustrated in Figure 3.2.a, occurs when circumcircles enclose more than three points. In contrast, Figure 3.2.b shows that for every triangle, the circumcircle encloses exactly three points.

3.2.3 Delauny Triangulation and Delauny Tetrahedralization

Let P' be a set of points in $\mathbb{R}^2$ and T' be a triangulation of P'. T' is a Delaunay Triangulation of P', denoted as $DT(P')$, if and only if, for every triangle in T', the circumcircle contains no other points of P' (see Figures 3.3.a, 3.3.b, 3.3.c, 3.3.d, and 3.3.e). Delaunay Tetrahedralization extends this concept to three dimensions. Let P' be a set of points in $\mathbb{R}^3$ and T' be a tetrahedralization of P'. T' is a Delaunay Tetrahedralization of P', denoted as

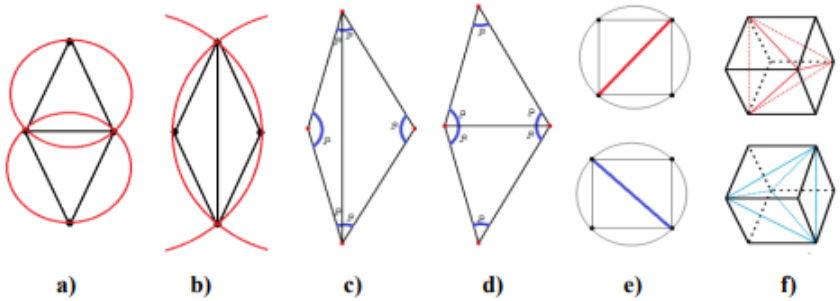

Figure 3.2: a) Delaunay Triangles b) Non-Delaunay Triangles c) Bad Angles in a Delaunay Triangulation. d) Good Angles in a Delaunay Triangulation. e) No-Unique Delaunay Triangulation f) No-Unique Delaunay Tetrahedralization [38]

$DT(P')$, if and only if, for every tetrahedron in T', the circumsphere contains no other points of P'.

Sequence of insertion of a point p in a Delaunay Triangulation

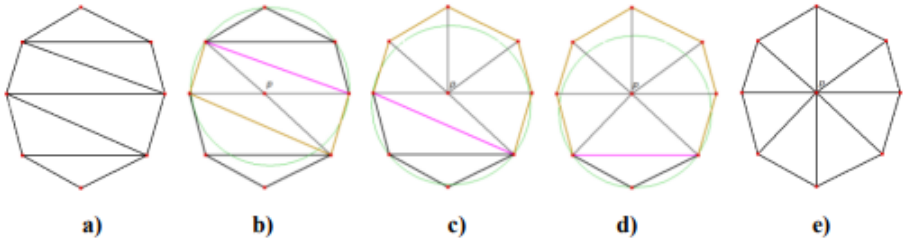

Figure 3.3: a) Initial Configuration b) Inserting a Point P c) Flipping Edgesd) Last Flipping of an Edge e) Final Configuration [38]

In Figure 3.3.a, the initial Delaunay Triangulation of the set of eight points is shown. In Figure 3.3.b, a new point p is inserted into the triangulation. Since p falls on an edge, two adjacent triangles are affected, resulting in the creation of four new triangles (two for each affected triangle). New edges are then evaluated using the flipping process, which requires the evaluation of the in-circle function, with the circumcircle illustrated. Figure 3.3.c shows the evaluation of one triangle, while Figure 3.3.d depicts the evaluation of another triangle. The circumcircles of the flipping edges are highlighted by colors. The final configuration of the Delaunay Triangulation is shown in Figure 3.3.e. Some intermediate steps have been omitted.

Delaunay Triangulation and Tetrahedralization Tests

$$\text{orientation2D}(p_0, p_1, p_2) = \begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = (x_0 - x_2)(y_0 - y_1) - (y_0 - y_2)(x_0 - x_1) \quad (3.1)$$

$$\text{inCircle}(p_0, p_1, p_2, p_3) = \begin{vmatrix} x_0 & y_0 & x_0^2 + y_0^2 & 1 \\ x_1 & y_1 & x_1^2 + y_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 & 1 \end{vmatrix} \quad (3.2)$$

$$\text{orientation3D}(p_0, p_1, p_2, p_3) = \begin{vmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} \quad (3.3)$$

$$\text{inSphere}(p_0, p_1, p_2, p_3, p_4) = \begin{vmatrix} x_0 & y_0 & z_0 & x_0^2 + y_0^2 + z_0^2 & 1 \\ x_1 & y_1 & z_1 & x_1^2 + y_1^2 + z_1^2 & 1 \\ x_2 & y_2 & z_2 & x_2^2 + y_2^2 + z_2^2 & 1 \\ x_3 & y_3 & z_3 & x_3^2 + y_3^2 + z_3^2 & 1 \\ x_4 & y_4 & z_4 & x_4^2 + y_4^2 + z_4^2 & 1 \end{vmatrix} \quad (3.4)$$

Delaunay Triangulation Orientation and inCircle Tests: To compute a Delaunay Triangulation, a circumcircle test is necessary, and the triangle must be oriented counterclockwise. The orientation can be determined using Equation 1, while the inCircle test can be performed using Equation 2.

Delaunay Tetrahedralization Orientation Test: Similarly, in three dimensions, the orientation test can be calculated using the determinant (see Equation 3). The circumsphere test, commonly referred to as the inSphere test, is used to verify a valid tetrahedron and is shown in Equation 4.

Chapter 4

Bowyer-Watson algorithm

This chapter delves into the Bowyer-Watson algorithm, a method for computing Delaunay triangulations, and provides an in-depth explanation of the algorithm's steps and implementation in three dimensions. The algorithm begins with an auxiliary tetrahedron or a convex hull containing all input points, inserting each point iteratively. Conflicting tetrahedra, whose circumscribed spheres contain the point to be inserted, are identified and removed, creating a star-shaped cavity. This cavity is then re-triangulated by connecting each face of the cavity with the new point.

The chapter also includes a detailed example illustrating the step-by-step process of the Bowyer-Watson algorithm in 2D. It begins with the creation of a super triangle that encompasses all points to be triangulated, followed by iterative steps of inserting points, identifying and marking invalid triangles, and replacing them with new valid triangles. The example highlights the challenges of finding polygonal holes formed by invalid triangles and the process of creating new triangles to fill these holes. The chapter concludes with the removal of triangles that share edges or vertices with the super triangle, resulting in a valid Delaunay triangulation.

4.1 Algorithm Definition

In 1981, Adrian Bowyer and David F. Watson independently introduced and published an algorithm in the same issue of The Computer Journal. This algorithm is designed to compute Delaunay triangulations but can be extended to compute regular triangulations by assigning weights to the input points and replacing the *in_sphere* test with a regularity test. The algo-

rithm functions in d-dimensions and has an expected time complexity of $O(n^{2d-1}/d)$. For simplicity, we will describe the algorithm in 3D.

The algorithm starts with an auxiliary tetrahedron that encompasses all the input points from the set S, similar to the incremental flipping algorithm, or alternatively, with a convex hull of S divided into tetrahedra. Points from S are inserted into the triangulation one at a time. In each step, all tetrahedra whose circumscribed spheres contain the new point p (known as the conflicting tetrahedra) are removed from the triangulation. This removal creates a cavity, forming a star-shaped polyhedron P with triangular faces. The polyhedron P is then re-triangulated by connecting each face of P to the new point p, thus forming new tetrahedra in the triangulation (see Fig. 4.1).

The conflicting tetrahedra can be identified efficiently. Initially, a tetrahedron T that contains p is located, using methods such as the walking algorithm. In the Delaunay triangulation, T is always the conflicting tetrahedron for p. In the regular triangulation, T might be non-conflicting, meaning T is regular with respect to p, indicating that p is redundant and not inserted into $RT(S)$. Because the conflicting tetrahedra form a continuous polyhedron, they can be found using a simple breadth-first or depth-first search. The search begins with the conflicting tetrahedron T and expands by exploring neighboring tetrahedra T^* of the currently known conflicting tetrahedra. If the circumscribed sphere of T^* contains p, then T^* is added to the list of conflicting tetrahedra.

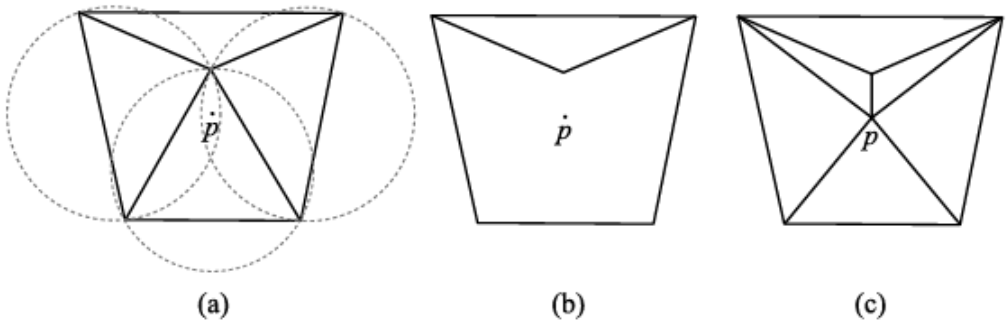

Figure 4.1: The Bowyer-Watson algorithm in 2D– insertion of the point p: (a) the conflicting triangles are marked by the circumscribed circle, (b) the triangulation with the star-shaped cavity, (c) the resulting triangulation [43].

4.2 Algorithm Example

To better understand the algorithm, an example is provided that illustrates the step-by-step process of the Bowyer-Watson algorithm in 2D.

Step 1: Creating the Super Triangle

The first essential step is to create a large triangle that encompasses all the points to be triangulated within its three edges. This triangle is referred to as a 'super triangle'. While it could theoretically be infinite in size to ensure all triangulation vertices are contained within it, in practice, it only needs to be large enough to enclose all the points (see Fig. 4.2).

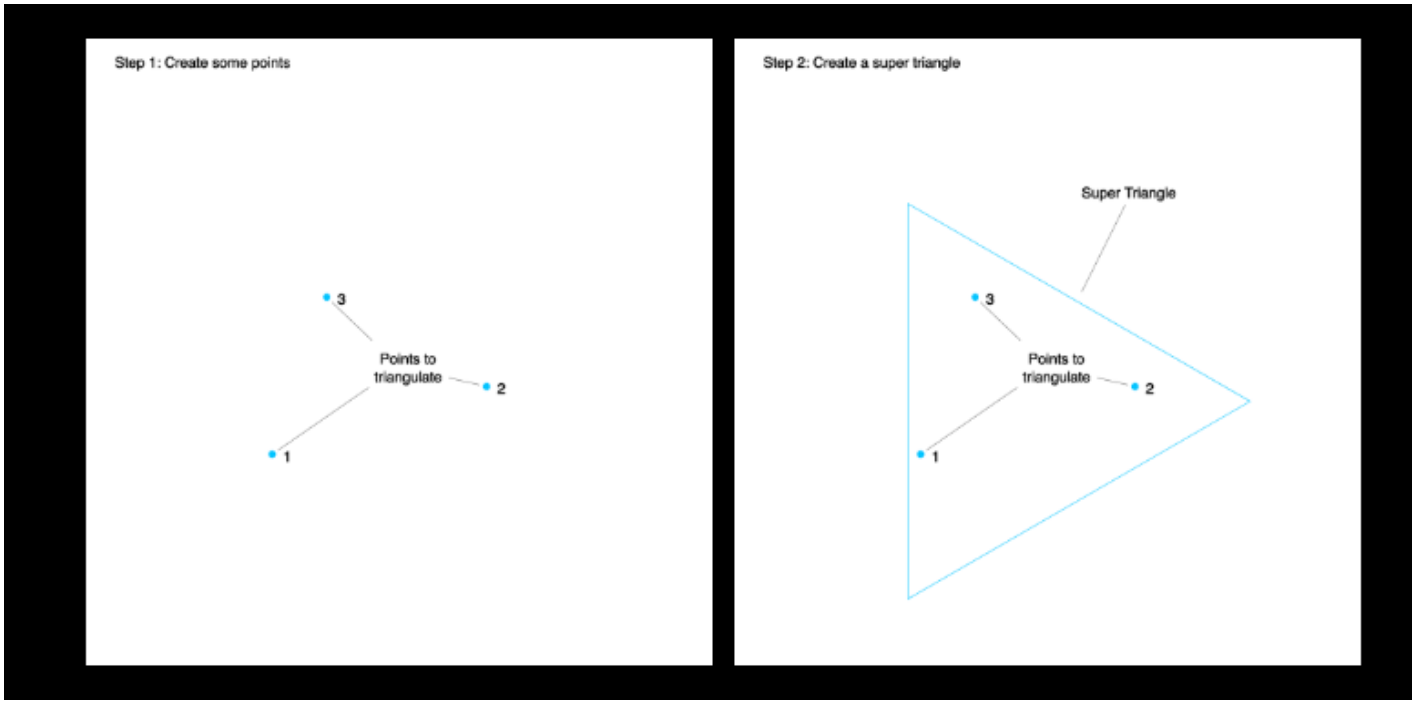

Figure 4.2: (a) Create some points, (b) Create a super triangle [44].

After creating the super triangle and positioning our points, we can begin constructing the triangulation. We will iteratively process each point to construct a valid triangulation.

The next step is to check if any of our points fall within the circumcircle of any of the existing triangles. Initially, this is straightforward since we only have one triangle (the super triangle), which certainly contains all the points within its circumcircle. We mark this super triangle as invalid and add it to an array that holds all invalid triangles. At this early stage, the array will only contain the super triangle (see Fig. 4.3).

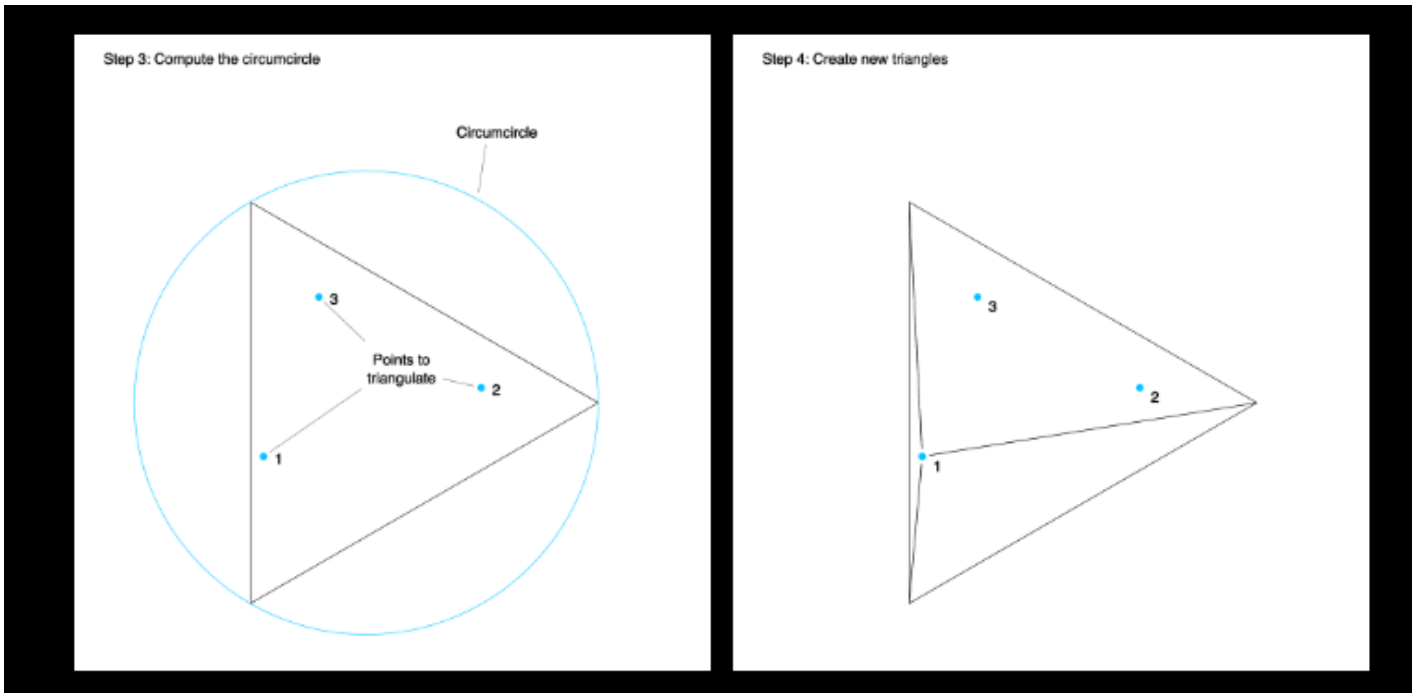

Figure 4.3: (a) Compute the circumcircle, (b)Create new triangles [44].

Next, we need to replace the invalid triangles with a new set of valid triangles. This is done by connecting the edges of the super triangle to the initial point we are currently iterating over. This ensures that the point lies on the circumcircle of the new triangles, as it becomes a vertex of each one of them.

As we proceed, this process becomes more complex because we need to identify the boundaries of the polygonal hole formed in the triangulation, where multiple triangles might have been marked as invalid. We need to fill this empty space, where the invalid triangles were removed, by connecting the edges of the polygonal hole to the current point being processed.

Before completing the current iteration, we must delete all triangles marked as invalid and add the new triangles formed by connecting the point to the edges of the polygonal hole. We can then proceed to the next iteration.

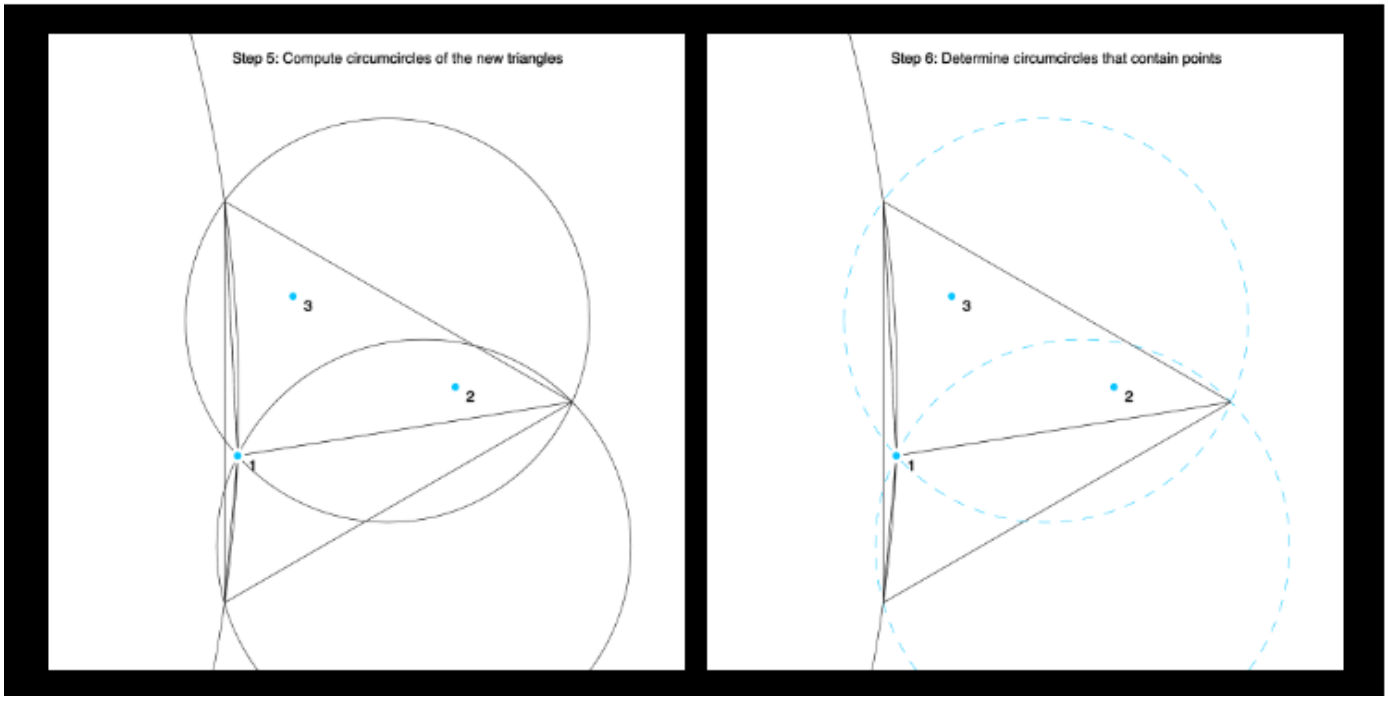

Figure 4.4: (a) Compute circumcircles of new triangles, (b)Determine the circmurcles that contain points [44].

In the second iteration, we add the next point to the triangular mesh. Depending on its position and the configuration of the three triangles we have created, the point may fall into the circumcircles of one, two, or more triangles. If it does, we mark those triangles as invalid and identify the polygonal hole formed by the removal of these triangles. The pseudocode describes the polygonal hole as ”all edges in the set of bad triangles that are not shared between two bad triangles,” meaning all edges at the boundaries of the polygonal hole. Depending on the implementation, this can vary in difficulty. We then form new triangles by connecting these unshared edges with the current point and adding these new triangles to the triangulation.

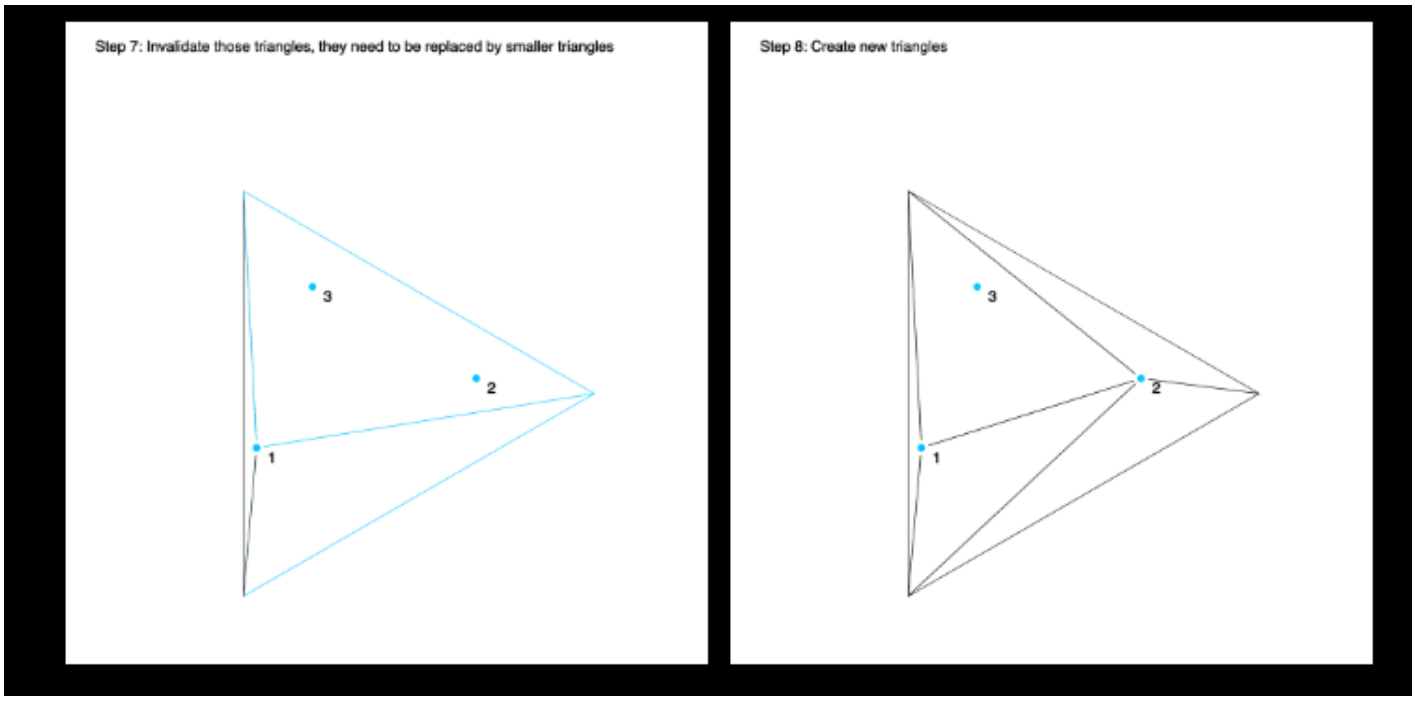

Figure 4.5: (a) Invalidate triangles, (b)Create new triangles [44].

In this way, we iteratively add vertices to the triangle mesh, subdividing the initial super triangle into smaller triangles until we achieve a valid Delaunay triangulation. In the final step, we need to remove all triangles that share an edge or a vertex with the initial super triangle, as they were not part of the original set of points.

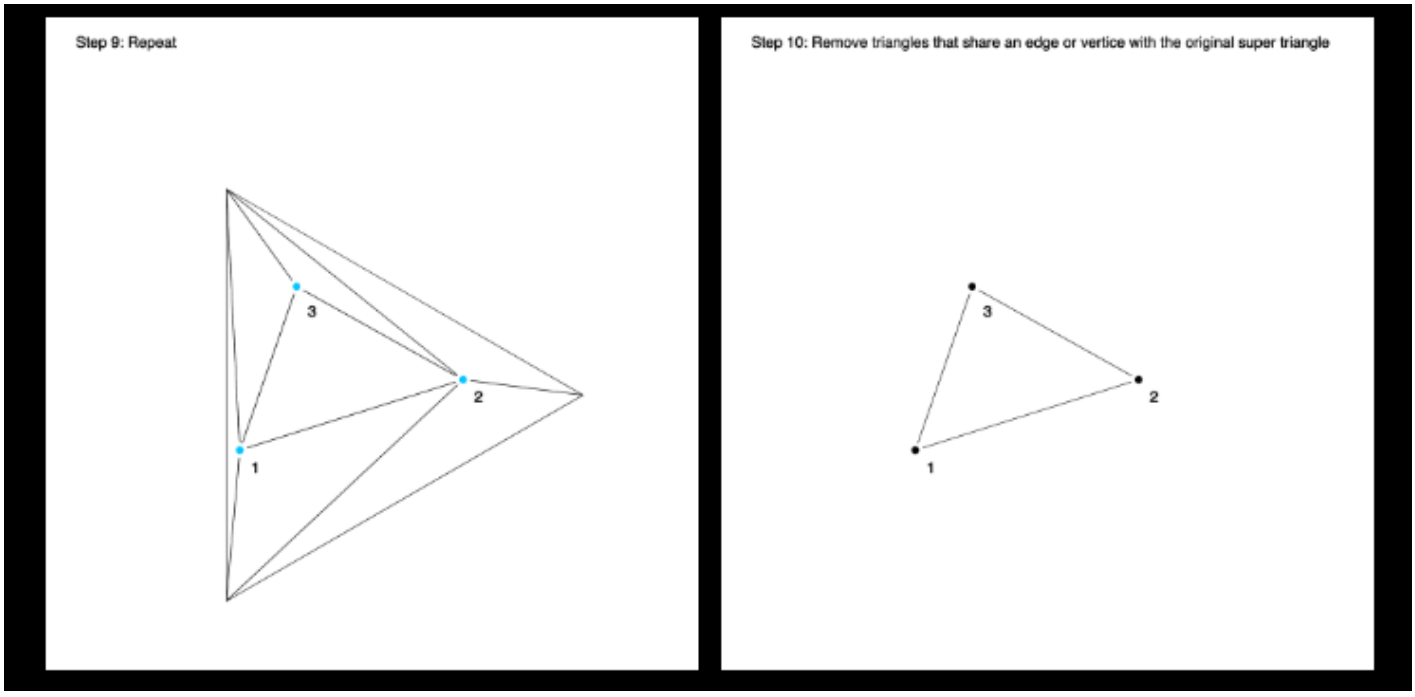

Figure 4.6: (a) Repeat, (b)Remove the edges and vertices shared with super triangle [44].

Chapter 5

Serial implementation

This Chapter outlines the implementation of a Delaunay triangulation algorithm specifically designed to process point cloud models. The algorithm accepts a raw set of points as input and produces a triangulated mesh, adhering to the Delaunay criterion for optimizing triangular mesh quality. This process is essential for numerous applications in computational geometry, graphics, and spatial data analysis.

The implemented algorithm is an advanced adaptation of the Bowyer-Watson algorithm, which is particularly effective in handling large datasets typical of modern point cloud environments. By leveraging spatial indexing and efficient memory management, the algorithm ensures high performance and scalability.

The algorithm begins by reading a 3D mesh file, for instance, a standard file format like OBJ, which in this example contains the ”Stanford Bunny” model. This mesh is then processed to extract vertex information and optionally compute vertex normals if they are not already specified.

This work is inspired by the thesis ”Parallel Tetrahedral Mesh Generation” [45] and the standard Bowyer-Watson algorithm, but with differences in the initialization and sorting processes.

Here, the algorithm offers flexibility in how points are sampled from the mesh:

- **Poisson disk sampling:** This method samples points based on a specified number and density, ensuring that points are evenly distributed across the surface of the mesh.

- **Uniform sampling:** Alternatively, points can be sampled uniformly across the surface of the mesh, which may be preferred depending on the application requirements.

After sampling, these points are visualized to confirm their distribution and quality before proceeding. The sampled points are then converted into a format suitable for parallel processing, leveraging GPU acceleration if available to enhance computational efficiency. The point cloud data is segmented into partitions corresponding to the number of threads, ensuring that the Delaunay triangulation can be performed efficiently across multiple cores.

The result is a comprehensive Delaunay triangulation, visualized to validate the mesh quality and integrity. This sequential approach from reading mesh files to producing and visualizing the triangulation exemplifies a robust application of computational geometry principles in processing complex three-dimensional data.

5.1 Serial Algorithm Overview

Let DT_k be the Delaunay triangulation of the subset $S_k = \{p_1, \ldots, p_k\} \subseteq S$. The *Delaunay kernel* is the procedure to insert a new point $p_{k+1} \in \Omega(S_k)$ into DT_k and construct a new valid Delaunay triangulation DT_{k+1} for $S_{k+1} = \{p_1, \ldots, p_k, p_{k+1}\}$. The Delaunay kernel can be expressed as follows:

$$DT_{k+1} \leftarrow DT_k - c(DT_k, p_{k+1}) + b(DT_k, p_{k+1}),$$

where the Delaunay cavity $c(DT_k, p_{k+1})$ consists of all tetrahedra whose circumspheres contain p_{k+1} (see Figure 5.1), and the Delaunay ball $b(DT_k, p_{k+1})$ is the set of tetrahedra that fill the polyhedral hole left by removing the Delaunay cavity $c(DT_k, p_{k+1})$ from DT_k (see Figure 5.1).

The complete state-of-the-art incremental insertion algorithm (Algorithm 1) implemented here is organized into five main steps, detailed below.

INIT: The triangulation begins with the tetrahedron formed by the first four non-coplanar vertices from the point set S. These vertices define a tetrahedron τ with a positive volume.

SORT: Before starting the point insertion, the points are sorted using Hilbert indices to ensure that points with nearby indices are also close in space. As a result, the three kernel functions *MOVE*, *CAVITY*, and *TETRAHEDRA* exhibit constant complexity in practice.

MOVE: The aim of this step is to locate the tetrahedron T_{k+1} that contains the next point to be inserted, p_{k+1}. The search starts from a tetrahedron T_k within the tetrahedra $B(DT_k, p_k)$ and progresses through the current triangulation toward p_{k+1} (Figure 5.1a). A point is deemed visible from a facet when the tetrahedron formed by this facet and the point has a negative orientation. The *MOVE:* function iterates over the four facets of τ, selects a facet from which p_{k+1} is visible, and moves through this facet to the adjacent tetrahedron. This new tetrahedron is designated as τ, and the *MOVE:* process is repeated until none of the facets of τ can see p_{k+1}, signifying that p_{k+1} is inside τ (see Figure 5.1a).

CAVITY: Once the tetrahedron $\tau \leftarrow \tau_{k+1}$ that contains the point to be inserted p_{k+1} has been identified, the *CAVITY* function finds all tetrahedra whose circumsphere contains p_{k+1} and deletes them. The Delaunay cavity $c(DT_k, p_{k+1})$ is simply connected and contains τ. The core and most expensive operation of the *CAVITY* function is the *inSphere* predicate, which evaluates whether a point e is inside, on, or outside the circumsphere of a given tetrahedron. This *CAVITY* function is thus an expensive part of the incremental insertion process and accounts for a significant portion of the total computation time.

TETRAHEDRA: After the cavity has been carved, the *TETRAHEDRA* function generates a set of new tetrahedra adjacent to the newly inserted point p_{k+1} to fill the cavity. It then updates the mesh structure, which involves computing the adjacencies between the newly created tetrahedra. This step is the most computationally expensive part of the algorithm.

Algorithm 1 Sequential computation of the Delaunay triangulation DT of a set of vertices S

Input: S

Output: $DT(S)$

0: **function** Sequential_Delaunay(S)
0: $\tau \leftarrow$ Init(S) {τ is the current tetrahedron}
0: $DT \leftarrow \tau$
0: $S' \leftarrow$ Sort($S \setminus \tau$)
0: **for** $p \in S'$ **do**
0: $\tau \leftarrow$ Move(DT, τ, p)
0: $c \leftarrow$ Cavity(DT, τ, p)
0: $DT \leftarrow DT \setminus c$
0: $B \leftarrow$ Tetrahedra(c, p)
0: $DT \leftarrow DT \cup B$
0: $\tau \leftarrow \tau \cup B$
0: **end for**
0: **return** DT
0: **end function**=0

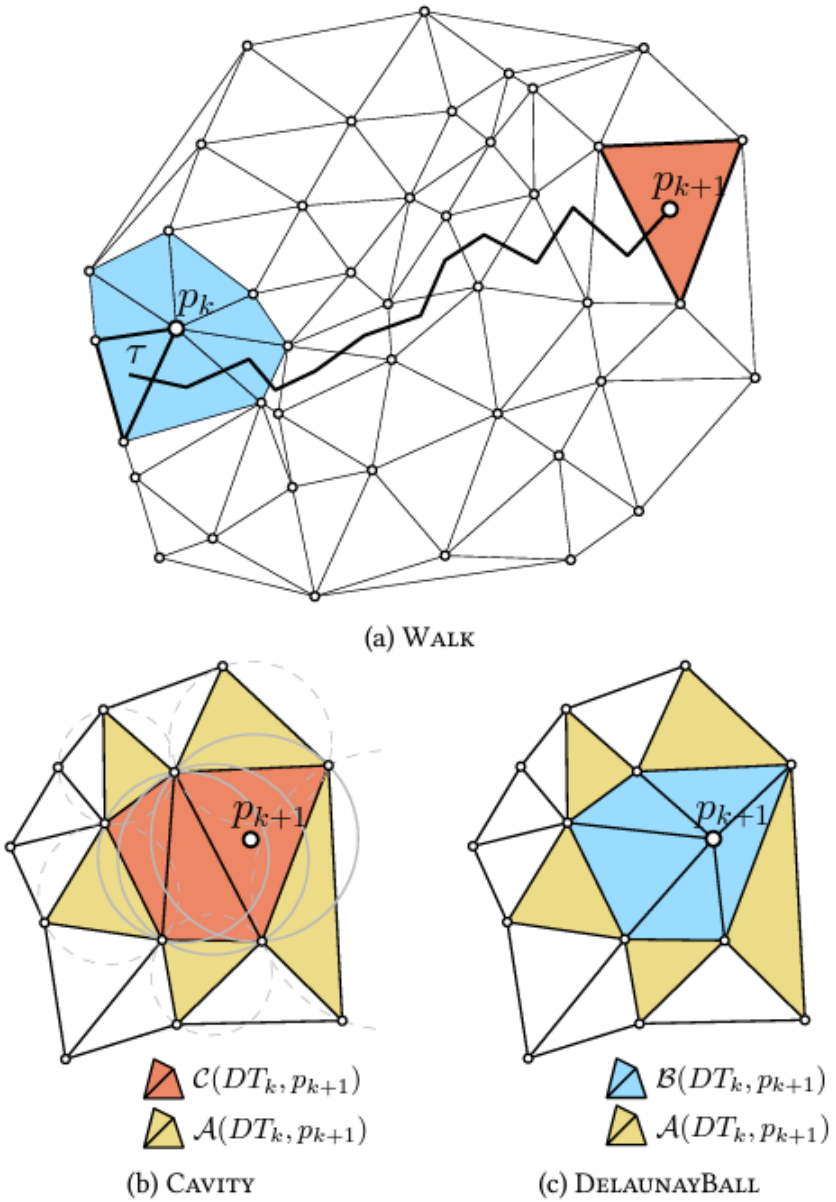

Figure 5.1: Insertion of a vertex p_{k+1} in the Delaunay triangulation DT_k: (a) The triangle containing p_{k+1} is obtained by moving toward p_{k+1}. The move starts from τ in B_{p_k}. (b) The Cavity function finds all cavity triangles (orange) whose circumcircle contains the vertex p_{k+1}. They are deleted, while cavity adjacent triangles (green) are kept. (c) The Tetrahedra function creates new triangles (blue) by connecting p_{k+1} to the edges of the cavity boundary [45].

5.2 Meshdata structure

My implementation of the Delaunay triangulation algorithm focuses on practical efficiency and is specifically tailored for processing 3D triangulations. By utilizing straightforward data structures in Python, enhanced with the capability of handling operations in CUDA for parallel execution, my approach ensures that the algorithm remains both flexible and efficient. This is particularly beneficial for handling complex point cloud data derived from 3D models, which typically involve a large number of vertices requiring efficient memory management and computational strategies. The core of my data structures consists of two primary classes, each responsible for different aspects of the triangulation process:

Tetrahedron Structure

The `Tetrahedron` structure represents a single tetrahedron in a 3D Delaunay triangulation. Each tetrahedron is defined by its vertices, neighbors, and edges, necessary for navigating and updating the triangulation mesh efficiently.

```
typedef struct {
    uint32_t vertices[4];
    uint32_t neighbors[4];
    uint32_t edges[6];
    uint32_t index;
} Tetrahedron;
```

DelaunayTriangulation Structure

The `DelaunayTriangulation` structure acts as the main container for the set of tetrahedra that form the Delaunay triangulation, managing vertices and a pool of indices for recycling tetrahedra.

```
typedef struct {
    Tetrahedron* tetrahedra;
    Point3d* vertices;
    uint32_t* free_indices;
    uint64_t num_tetrahedra;
    uint64_t num_vertices;
```

```
} DelaunayTriangulation;
```

Explanation:

- **Vertices:** Each `Tetrahedron` object contains an array of vertex indices that define its geometric corners. These indices (`uint32_t*`) reference a global list of all vertices managed by the `DelaunayTriangulation` class, crucial for establishing the spatial structure of each tetrahedron.
- **Neighbors:** This array (`uint32_t*`) within the `Tetrahedron` class keeps track of adjacent tetrahedra, one for each face of the tetrahedron. It is essential for navigating the mesh during the triangulation updates, such as during the cavity creation and retriangulation processes.
- **Edges:** Computed during the initialization of each `Tetrahedron`, this list (`uint32_t*`) contains pairs of vertex indices that form the edges. The pre-computation of edges facilitates operations that require edge data, like edge flipping, which is central to maintaining the Delaunay condition.
- **Index:** Every `Tetrahedron` is assigned a unique index (`uint32_t*`) upon creation, which is used to reference it in the list of tetrahedra maintained by the `DelaunayTriangulation` class. This index is vital for quick access and identification of tetrahedra during mesh manipulations and algorithmic operations.
- **Tetrahedron Class:** This class functions as the backbone of my triangulation structure, encapsulating the vertices, neighbors, and edges of each tetrahedron. It provides methods for determining connectivity and performing local checks and updates necessary for dynamic triangulation adjustments.
- **DelaunayTriangulation Class:** Acts as the central repository for the triangulation, holding comprehensive lists of all vertices and tetrahedra. It coordinates the broader operations of the triangulation process, such as point insertion and mesh optimization, and manages a pool of free indices which optimizes memory usage by recycling the indices of removed tetrahedra.
- **Tetrahedra:** A pointer (`Tetrahedron*`) that points to an array of `Tetrahedron` structures. This array contains all the tetrahedra in the triangulation. Each `Tetrahedron`

in this array represents a single tetrahedron in the 3D space, detailing its vertices, neighbors, and other relational attributes.

- **Vertices:** A pointer (`Point3d*`) to an array of `Point3d` structures, which stores the coordinates of each vertex in the triangulation. This centralized storage allows for efficient access and manipulation of vertex data across various tetrahedra.

- **Free Indices:** An array (`uint32_t*`) that holds the indices of tetrahedra that have been removed from the triangulation and are available for reuse. This mechanism helps in optimizing memory usage and reducing allocation overhead during dynamic updates to the triangulation.

- **Num Tetrahedra:** A variable (`uint64_t`) that keeps count of the total number of tetrahedra currently active in the triangulation. This count includes both used and potentially free tetrahedra marked for reuse.

- **Num Vertices:** A variable (`uint64_t`) that tracks the total number of vertices stored in the `vertices` array. This is crucial for managing the size of the vertex array and understanding the scope of the vertex data being manipulated.

5.3 Input

The input to my algorithm is an array of points in 3D space. These points can be acquired from various sources, such as OBJ files, and sampled using methods like Poisson disk sampling or uniform sampling. For instance, points can be sampled from a mesh stored in an OBJ file and converted to a point cloud. This point cloud is then transformed into a numpy array for further processing.

5.4 Output

The output of the `Sequential_Delaunay` algorithm is a fully constructed instance of the `DelaunayTriangulation` class. This instance encapsulates the complete Delaunay triangulation of the input point cloud, providing a detailed and precise geometric representation of the data.

The `DelaunayTriangulation` object functions as a robust geometric data structure that is primed for a variety of further applications, including finite element methods, surface reconstruction, and 3D visualization. This data structure is engineered to support efficient access and manipulation of the mesh, enabling a wide range of computational geometry operations and analyses.

Importantly, the `DelaunayTriangulation` output is readily usable for visualizing the triangulated data, allowing for a visual inspection and analysis of the mesh quality and structure. This visualization capability is crucial for verifying the accuracy of the triangulation and for presenting or communicating geometric properties and relationships inherent in the input data set. The effective visualization of the Delaunay mesh not only aids in understanding the underlying spatial relationships but also supports further processing tasks such as mesh refinement, optimization, and other post-processing techniques necessary for scientific and engineering applications.

Performance Considerations

While specific performance metrics for the `Sequential_Delaunay` algorithm have not been quantitatively measured at this stage, preliminary observations suggest that the process can be computationally intensive, especially when dealing with large datasets. The computational complexity primarily arises from the need to check each point against the circumspheres of existing tetrahedra, which can become a bottleneck as the number of points increases.

Potential Areas for Improvement: The current implementation may exhibit slower performance due to several factors:

- **Single-threaded execution:** The algorithm runs on a single processing thread, which limits its ability to handle large-scale triangulations efficiently.
- **Data structure management:** The dynamic nature of data structures used to manage vertices and tetrahedra can introduce overhead, particularly in memory management and access times.
- **Algorithmic efficiency:** Certain steps in the triangulation process, such as moving through the mesh to find the containing tetrahedron and updating the triangulation

structure, may not be optimized for best performance.

Next chapter: To enhance the performance of the Delaunay triangulation, future work will focus on:

- Implementing parallel processing techniques to leverage multi-core and multi-threaded environments.
- Optimizing data structures to reduce memory overhead and improve access times, possibly through the use of spatial indexing techniques like octrees or k-d trees.
- Refining the algorithm to reduce the computational complexity of mesh updates, particularly by minimizing the number of circumsphere checks and optimizing the point insertion strategy.

These improvements are expected to significantly reduce computational times and enhance the algorithm's scalability, making it more suitable for practical applications in real-time systems and large-scale projects.

Chapter 6

Parallel implementation

In this chapter, the focus is on the parallel implementation of Delaunay triangulations, addressing both distributed and shared memory architectures to overcome memory and time limitations. The chapter begins with an overview of various parallel algorithms, emphasizing the need to construct Delaunay triangulations in parallel. It explores different methods to subdivide a triangulation into multiple parts, stored on nodes of a distributed memory cluster, and discusses the complexities of merging independently generated Delaunay triangulations. The section highlights the use of synchronization between processors to maintain a single valid Delaunay triangulation and avoid complicated merge steps.

The chapter then delves into a parallel strategy based on partitions, using a space-filling curve to minimize synchronization and data-races between threads. Each partition is managed by a unique thread, ensuring that each thread works on an independent part of the mesh, thereby enhancing efficiency. The partitioning is achieved using the Hilbert curve, known for its good clustering properties, to ensure effective parallel computation. The chapter also discusses the data structures optimized for parallel execution, leveraging CUDA for high-performance processing of large 3D triangulations. The structured approach to partitioning and parallel processing ensures that the triangulation algorithm remains efficient and scalable, suitable for handling complex point cloud data derived from 3D models. The chapter concludes with performance considerations, highlighting the advantages of the parallel algorithm over sequential implementations, and suggests potential areas for further improvement, such as load balancing, memory access optimization, and scalability.

6.1 Parallel Algorithm Overview

To overcome memory and time limitations, Delaunay triangulations should be constructed in parallel, utilizing both distributed and shared memory architectures. A triangulation can be divided into multiple parts, with each part constructed and stored on a node of a distributed memory cluster. Various methods have been proposed to merge independently generated Delaunay triangulations, but these methods often involve complex merge steps that are difficult to parallelize. To avoid merge operations, other distributed implementations maintain a single valid Delaunay triangulation, using synchronization between processors to handle conflicts at inter-processor boundaries. In finite-element mesh generation, merging two triangulations can be simpler because the triangulations do not need to be fully Delaunay, allowing algorithms to focus primarily on load balancing.

On shared memory machines, divide-and-conquer approaches remain efficient. However, other methods have been proposed since communication costs between threads are not prohibitive, unlike in distributed memory machines. To insert a point in a Delaunay triangulation, the kernel procedure operates on a cavity that is modified to accommodate the inserted point. Two points can be inserted concurrently in a Delaunay triangulation if their respective cavities do not intersect, avoiding conflicts. In practice, other types of conflicts and data races should be considered, depending on the chosen data structures and insertion strategy. Conflict management strategies that rely heavily on locks achieve relatively good speedups with a small number of cores. Unlike pure divide-and-conquer strategies for distributed memory machines, partitions in shared memory systems can change and move regularly, and they do not need to cover the whole mesh at once.

6.2 A Parallel Strategy Based on Partitions

To avoid data races and conflicts between threads when concurrently inserting points in a Delaunay triangulation, a program must satisfy multiple conditions. For example, consider thread t_1 inserting point p_{k+1} while thread t_2 is simultaneously inserting point p_{k+2}:

1. Thread t_1 cannot access information about any tetrahedron in $C(DT, p_{k2})$ and inversely. Hence:

   a) $C(DT, p_{k1}) \cap C(DT, p_{k2}) = \emptyset$

b) $A(DT, p_{k1}) \cap C(DT, p_{k2}) = \emptyset$ and $A(DT, p_{k2}) \cap C(DT, p_{k1}) = \emptyset$

c) Thread t_1 cannot move into $C(DT, p_{k2})$ and reciprocally, t_2 cannot move into $C(DT, p_{k1})$

2. A tetrahedron in $B(DT, p_{k1})$ and a tetrahedron in $B(DT, p_{k2})$ cannot be created at the same memory index.

To ensure rule (1) it is sufficient to restrain each thread to work on an independent partition of the mesh. This lock-free strategy minimizes synchronization between threads and is very efficient. Each point of the Delaunay triangulation is assigned a partition corresponding to a unique thread.

To ensure (1c), i forbid threads to move in tetrahedra belonging to another thread. Consequently, a thread aborts the insertion of a vertex when (i) the move reaches another partition. To insert these points i just add them in the Tetrhaedra method all togteher.

Rule (2) is obvious from a parallel memory management point of view. It is, however, difficult to ensure it without requiring an unreasonable amount of memory.

6.3 Algorithm in depth

INIT: The triangulation begins with the tetrahedron formed by the first four non-coplanar vertices from the point set S. These vertices define a tetrahedron τ with a positive volume.

PARTITION: Instead of sorting the points, i partition the n_v points to insert, ensuring each thread handles an equal number of points. My partitioning method is based on Hilbert indices, which ensures that points with nearby indices are also close in space. Each of the n_{thread} partitions consists of a set of grid cells that are consecutive along the Hilbert curve. The points are assigned to partitions according to their Hilbert indices, with the first $\frac{n_v}{n_{\text{threads}}}$ points assigned to the first partition, the next $\frac{n_v}{n_{\text{threads}}}$ to the second, and so on.

MOVE: The aim of this step is to locate the tetrahedron T_{k+1} that contains the next point to be inserted, p_{k+1}. The search starts from a tetrahedron T_k within the Delaunay ball $B(DT_k, p_k)$ and progresses through the current triangulation toward p_{k+1}. A point is deemed visible from a facet when the tetrahedron formed by this facet and the point has

a negative orientation. The *MOVE* function iterates over the four facets of τ, selects a facet from which p_{k+1} is visible, and moves through this facet to the adjacent tetrahedron. This new tetrahedron is designated as τ, and the *MOVE* process is repeated until none of the facets of τ can see p_{k+1}, signifying that p_{k+1} is inside τ.

CAVITY: Once the tetrahedron $\tau \leftarrow \tau_{k+1}$ that contains the point to be inserted p_{k+1} has been identified, the *CAVITY* function finds all tetrahedra whose circumsphere contains p_{k+1} and deletes them. The Delaunay cavity $c(DT_k, p_{k+1})$ is simply connected and contains τ. The core and most expensive operation of the *CAVITY* function is the *inSphere* predicate, which evaluates whether a point e is inside, on, or outside the circumsphere of a given tetrahedron. This *CAVITY* function is thus an expensive part of the incremental insertion process and accounts for a significant portion of the total computation time.

TETRAHEDRA: After the cavity has been carved, the *TETRAHEDRA* function generates a set of new tetrahedra adjacent to the newly inserted point p_{k+1} to fill the cavity. Each thread checks if any other point from any thread is contained within the circumsphere of the cavity tetrahedra. If true, those points are added to the appropriate partition as part of the Delaunay ball. The function then updates the mesh structure, which involves computing the adjacencies between the newly created tetrahedra. This step is the most computationally expensive part of the algorithm.

REPARTITION: Once the first batch of points has been triangulated, i repartition the remaining points based on their Hilbert indices. Each thread is then assigned a new subset of points and the corresponding segment of the triangulation to manage. This process is repeated, ensuring efficient and balanced workload distribution among the threads until all points are inserted.

6.4 Datastructure

My implementation of the Delaunay triangulation algorithm emphasizes practical efficiency and is specifically optimized for processing 3D triangulations. By leveraging straightforward data structures in Python and enhancing them with CUDA for parallel execution, this approach ensures that the algorithm remains both flexible and highly efficient. This design is particularly advantageous for handling complex point cloud data derived from 3D models,

which typically involve a large number of vertices and require robust memory management and computational strategies.

The core of my implementation is centered around the use of CUDA for parallel processing, significantly accelerating the triangulation process by distributing computations across multiple GPU threads. This allows for the efficient handling of large datasets, reducing the overall execution time compared to traditional serial implementations. The data structures used are designed to support this parallel execution, ensuring minimal overhead and maximizing performance. The following outlines the key data structures and their initialization:

```
d_partitions = cuda.device_array((len(partitions),
                                  max_size, 3),
                                  dtype=np.float64)

num_points_per_partition = np.array([part.shape[0]
                                  for part in partitions])

for idx, part in enumerate(partitions):
    num_rows = part.shape[0]
    d_partitions[idx, :num_rows, :] = cuda.to_device(part)

d_tau = cuda.device_array((len(partitions), 4, 3),
                               dtype=np.float64)

d_add_result = cuda.device_array(len(partitions),
                                  dtype=np.int32)

d_move_results = cuda.device_array(len(partitions),
                                  dtype=np.int32)

d_find_cavity_results =
cuda.device_array((len(partitions), 100),
                           dtype=np.int32)
```

```
cavity_num = cuda.device_array(len(partitions),
                               dtype=np.int32)

boundary_facets = cuda.device_array((MAX_TETRAHEDRA, 3, 3),
                                    dtype=np.float64)

boundary_facets_count = cuda.device_array(1, dtype=np.int32)

d_create_delaunay_ball_results =
cuda.device_array((len(partitions),
                   MAX_TETRAHEDRA),
                  dtype=np.int32)

delauny_ball_num = cuda.device_array(len(partitions),
                                     dtype=np.int32)

d_last_tetra_index = cuda.device_array(len(partitions),
                                       dtype=np.int32)

tetrahedra =
cuda.device_array((len(partitions), MAX_TETRAHEDRA, 5, 6),
                                     dtype=np.float64)

tetrahedra[:, 2, 0] = -1

vertices = cuda.device_array((MAX_VERTICES, 3),
                             dtype=np.float64)

tetrahedron_count = cuda.device_array(len(partitions),
                                      dtype=np.int32)

num_partitions = cuda.device_array(1, dtype=np.int32)
```

```
num_partitions = len(partitions)

free_indices = cuda.device_array(MAX_TETRAHEDRA, dtype=np.int32)
free_indices_count = cuda.device_array(1, dtype=np.int32)

vertex_count = cuda.device_array(1, dtype=np.int32)
```

- **d_partitions**: This array encapsulates the distribution of point coordinates across various partitions, specifically structured to facilitate parallel processing. Each partition contains a subset of the main dataset, organized to optimize the handling and computational efficiency in multi-threaded operations. It is represented as a 3D array of type float64, with dimensions detailed as [number of partitions, maximum partition size, 3], where the third dimension corresponds to the x, y, and z coordinates of the points.

- **d_tau**: This array stores the initial tetrahedra assigned to each partition. These tetrahedra serve as the foundational elements for beginning the triangulation process within each thread. The type of d_tau is a 3D array of float64, structured as [number of partitions, 4 vertices per tetrahedron, 3 coordinates per vertex], effectively capturing the geometric data necessary for tetrahedral formation.

- **d_add_result, d_move_results, d_find_cavity_results**: These arrays are integral for storing intermediate results from various operations within the triangulation process, such as adding vertices to the mesh, navigating through the tetrahedral network to locate specific positions, and identifying cavities that emerge during vertex insertion. d_add_result and d_move_results are 1D int32 arrays capturing the outcomes of their respective operations, while d_find_cavity_results is a 2D int32 array with dimensions [number of partitions, 100], designed to accommodate multiple potential cavity detections within each partition.

- **cavity_num**: This array plays a crucial role in tracking the number of cavities identified per partition throughout the process of point insertion. It is formatted as a 1D int32 array, quantifying the occurrence of cavities which are critical for subsequent mesh modifications and optimizations.

- **boundary_facets and boundary_facets_count**: These elements manage the boundary facets of tetrahedra, which are essential for maintaining the mesh's connectivity and

integrity during the triangulation. boundary_facets is a 3D float64 array storing the vertices of each facet, while boundary_facets_count is a 1D int32 array that records the number of such facets, highlighting their significance in the structural composition of the mesh.

- **d_create_delaunay_ball_results, delauny_ball_num**: This component is responsible for the outcomes associated with creating Delaunay balls, which are pivotal for re-triangulating cavities after the insertion of points. The results are stored in a 2D int32 array (d_create_delaunay_ball_results), and the count of these balls is maintained in a 1D int32 array (delauny_ball_num), underscoring their role in enhancing triangulation accuracy and mesh stability.

- **d_last_tetra_index**: This array is crucial for keeping track of the last tetrahedron index processed in each partition. It facilitates continuity and coherence between operations, ensuring that the mesh modification process is seamless and efficient. The array is formatted as a 1D int32 array, serving as a reference point for ongoing computational activities within each partition.

- **tetrahedra**: This array acts as the primary storage for all tetrahedra in the triangulation process, containing detailed information about their vertices and associated properties. Specifically, the array is created as a CUDA device array to leverage GPU processing capabilities, enhancing computational efficiency in handling complex triangulations. It is defined with dimensions (len(partitions), MAX_TETRAHEDRA, 5, 6), where ‘len(partitions)‘ represents the number of partitions, ‘MAX_TETRAHEDRA‘ specifies the maximum number of tetrahedra per partition, and the last two dimensions (5, 6) detail additional properties and coordinates of each vertex. The array is of type float64, reflecting the precision needed for spatial computations in triangulation tasks.

- **vertices**: This global array holds all vertex coordinates involved in the triangulation, ensuring that each vertex’s position is precisely documented for mesh construction and modification. It is a 2D float64 array, with dimensions [maximum vertices, 3 coordinates per vertex], providing a comprehensive database of vertex locations.

- **tetrahedron_count, num_partitions, free_indices, free_indices_count, vertex_count**: These arrays manage various counts critical to the triangulation process, such as the

total number of tetrahedra per partition, the total number of partitions, the availability of free indices for reusing tetrahedra slots, and the total count of vertices. The arrays are formatted as 1D int32 arrays, each playing a distinct role in the efficient allocation and utilization of computational resources within the triangulation framework.

6.5 Partitioning and re-partitioning with hilbert indices

I subdivide the n_v points to insert such that each thread inserts the same number of points. My partitioning method is based on the Hilbert curve, known for its excellent clustering properties and space-filling nature. Space-filling curves are relatively fast to construct and have been successfully used for partitioning tasks in various computational geometry applications.

Each of the n_{thread} partitions consists of a set of grid cells that are consecutive along the Hilbert curve. To compute the partitions, i.e., their starting and ending Hilbert indices, I sort the points according to their Hilbert indices I then assign the first $\frac{n_v}{n_{\text{threads}}}$ points to the first partition, the next $\frac{n_v}{n_{\text{threads}}}$ points to the second partition, and so on.

The second step is to partition the current Delaunay triangulation where the points will be inserted. I again use the hilbert indices to assign the mesh vertices to the different partitions. Each thread then owns a subset of points to insert and a subset of the current triangulation.

When the first batch is triangulated, i repartition the remaining points and add the points accordingly.

- **Normalization:** The points are normalized to fit within a standard range, ensuring that the Hilbert curve computation applies uniformly across all dimensions.
- **Scaling:** The normalized points are scaled to an appropriate integer range suitable for the Hilbert curve calculation, facilitating the conversion of floating-point coordinates to integer-based indices.
- **Hilbert Curve Calculation:** A Hilbert curve calculator is initialized with specified bits and dimensions. For each point in the scaled space, a Hilbert index is computed, which determines the point's position along the curve.
- **Partitioning:** Points are sorted based on their Hilbert indices. The sorted points are then divided into desired partitions, each intended for processing by a separate computational thread or processing unit. This step ensures that each partition handles a

roughly equal number of points, minimizing data dependencies and potential conflicts during the triangulation process.

This structured approach not only ensures that points close in the geometric space are likely to remain close in the partitioning scheme, but also facilitates the parallel processing of the Delaunay triangulation without extensive synchronization barriers, thereby optimizing both memory usage and computational overhead.

6.6 Output

The output of the `Sequential_Delaunay` algorithm is written to two primary arrays: `tetrahedra` and `vertices`. These arrays together encapsulate the complete Delaunay triangulation of the input point cloud, providing a detailed and precise geometric representation of the data.

The `tetrahedra` array stores the tetrahedra formed during the triangulation process. This array is structured as a CUDA device array with dimensions `(len(partitions), MAX_TETRAHEDRA, 5, 6)`, where `len(partitions)` is the number of partitions, `MAX_TETRAHEDRA` is the maximum number of tetrahedra per partition, and the last two dimensions `(5, 6)` store detailed properties and coordinates for each vertex of the tetrahedra. The array uses `float64` for high precision in spatial computations.

The `vertices` array holds the coordinates of all vertices involved in the triangulation. It is a 2D `float64` array with dimensions `(maximum_vertices, 3)`, where the second dimension corresponds to the x, y, and z coordinates of the vertices. This array ensures that each vertex's position is precisely documented for accurate mesh construction and modification.

Together, the `tetrahedra` and `vertices` arrays form a robust geometric data structure suitable for a variety of further applications, including finite element methods, surface reconstruction, and 3D visualization. These data structures support efficient access and manipulation of the mesh, enabling a wide range of computational geometry operations and analyses.

The output is also readily usable for visualizing the triangulated data, allowing for a visual inspection and analysis of the mesh quality and structure. This visualization capability is crucial for verifying the accuracy of the triangulation and for presenting or communicating

geometric properties and relationships inherent in the input dataset. Effective visualization of the Delaunay mesh aids in understanding the underlying spatial relationships and supports further processing tasks such as mesh refinement, optimization, and other post-processing techniques necessary for scientific and engineering applications.

6.7 Performance Considerations

The `Parallel_Delaunay` algorithm exhibits significantly faster performance compared to the `Sequential_Delaunay` algorithm. This performance boost is primarily attributed to the parallel processing capabilities that allow the algorithm to handle large datasets more efficiently. By distributing the computational workload across multiple processing threads, the `Parallel_Delaunay` algorithm can achieve substantial reductions in execution time.

Performance Advantages:

- **Multi-threaded execution:** The parallel algorithm leverages multiple processing threads, enabling it to perform simultaneous computations. This parallelism effectively reduces the time required to complete the triangulation process, especially for large-scale datasets.
- **Efficient data structure management:** The algorithm is designed to manage data structures in a way that minimizes overhead. By using optimized data structures and memory management techniques, the algorithm can quickly access and update vertices and tetrahedra, enhancing overall performance.
- **Algorithmic optimization:** Key steps in the triangulation process, such as locating the containing tetrahedron and updating the mesh structure, have been optimized for parallel execution. These optimizations help to minimize bottlenecks and ensure that the algorithm can efficiently process each point in the dataset.

Potential Areas for Further Improvement: While the `Parallel_Delaunay` algorithm offers significant performance improvements, there are still areas that could benefit from further optimization:

- **Load balancing:** Ensuring an even distribution of the computational load across all processing threads can help to maximize the efficiency of the parallel algorithm. Techniques such as dynamic load balancing could be explored to address this.

- **Memory access patterns:** Optimizing memory access patterns to reduce cache misses and improve data locality can further enhance the performance of the algorithm.
- **Scalability:** Investigating the scalability of the algorithm to handle even larger datasets and more complex geometries could provide additional insights into potential performance gains.

Chapter 7

Environment

Numba is a *Just-in-time* (JIT) compiler for Python, meaning it converts Python functions to machine code at runtime, allowing them to execute at the speed of native machine code. Sponsored by Anaconda Inc. and supported by various organizations, Numba significantly enhances performance.

By using Numba, we can accelerate computation-intensive and calculation-heavy Python functions, particularly those involving loops. Numba also supports the *NumPy* library, enabling faster execution of numerical computations that rely on NumPy. This is particularly useful because loops in Python are notoriously slow. Additionally, Numba allows the use of many functions from Python's *math* library, such as *sqrt*, to further optimize performance.

7.1 Why Numba?

So, why use Numba when there are many other compilers like Cython, or alternatives like PyPy?

The simple reason is that with Numba, we don't have to leave the comfort of writing code in Python. Indeed, we don't have to change our code at all for basic speedup, which is comparable to the speedup achieved with Cython code that includes type definitions. Isn't that great?

We only need to add a familiar Python functionality, a decorator (a wrapper) around our functions. Additionally, a wrapper for a class is also under development.

So, all we need to do is add a decorator, and we're done. For example:

```
from numba import jit
```

```
@jit
def function(x):
    # our loop or numerically intensive computations
    return x
```

How Numba Works

Numba generates optimized machine code from pure Python code using the LLVM compiler infrastructure. The execution speed of code optimized with Numba is comparable to that of similar code written in C, C++, or Fortran.

Here is how the code is compiled:

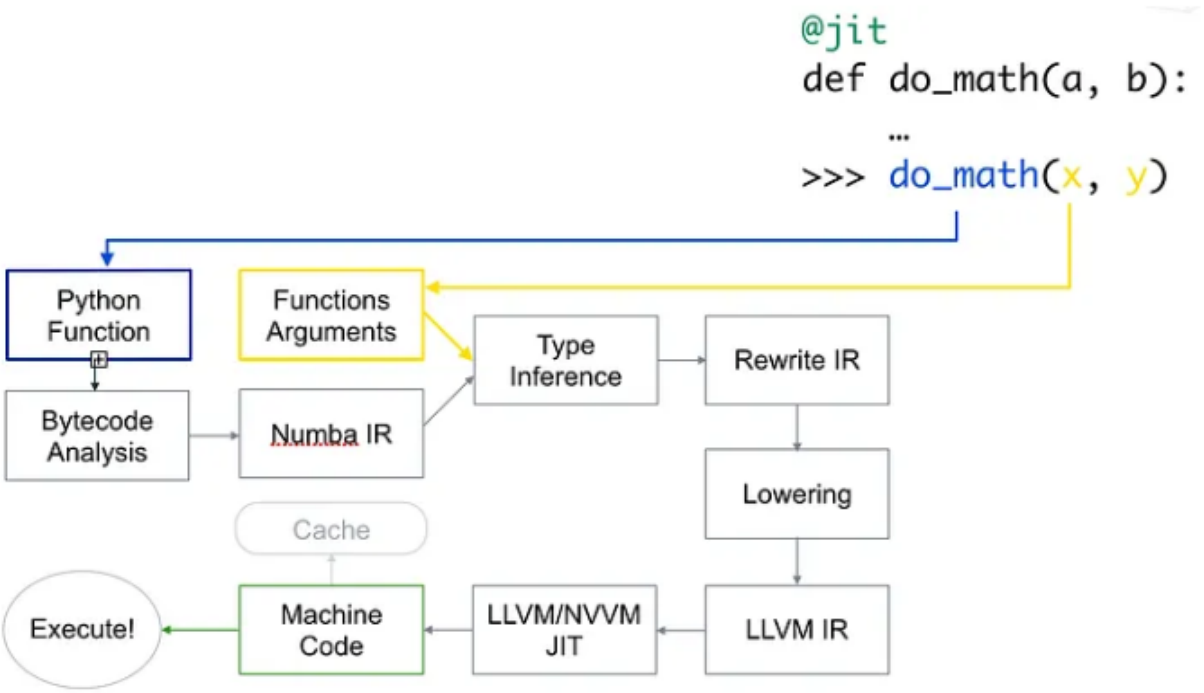

Figure 7.1: This figure illustrates the Just-in-Time (JIT) compilation workflow of Numba, a Python library that facilitates the acceleration of Python code. Starting with the Python function, the process involves several stages: analyzing bytecode, inferring types, transforming to and optimizing Numba's intermediate representation (IR), and finally lowering this into LLVM's IR for JIT compilation into optimized machine code. The result is performance-efficient execution directly on hardware, making Python an effective choice for computationally intensive tasks.[46]

First, a Python function is taken, optimized, and converted into Numba's intermediate representation. After type inference, similar to NumPy's type inference (e.g., a Python float becomes a float64), it is converted into LLVM-interpretable code. This code is then fed to LLVM's just-in-time compiler, producing machine code.

We can generate this code either at runtime or import time, and it can be executed on a CPU (default) or GPU, depending on our preference.

Using basic numba functionalities

For optimal performance, Numba recommends using the `nopython=True` argument with the `jit` decorator. This ensures that the Python interpreter is not used at all. Alternatively, we can use `@njit`, which is equivalent. If the `nopython=True` option results in an error, we can use the standard `@jit` decorator. This will compile parts of the code that can be converted to machine code, such as loops, while the remaining code is handled by the Python interpreter.

So, we just have to do:

```
from numba import njit, jit

@njit    # or @jit(nopython=True)
def function(a, b):
    # our loop or numerically intensive computations
    return result
```

When using `@jit`, ensure that your code includes elements that Numba can compile, such as compute-intensive loops, and uses supported libraries (like NumPy) and functions. Otherwise, Numba will not be able to compile anything.

A notable advantage is that Numba caches the functions after their first use as machine code. This means that subsequent executions will be even faster, as the code doesn't need to be recompiled, provided the argument types remain the same.

If your code is parallelizable, you can also pass `parallel=True` as an argument, but it must be used together with `nopython=True`. Currently, this only works on the CPU.

You can also specify the function signature you want your function to have, but then it will only compile for those specific argument types. For example:

```
from numba import jit, int32

@jit(int32(int32, int32))
def function(a, b):
    # our loop or numerically intensive computations
    return result
```

```
# or if we haven't imported type names
# we can pass them as string

@jit('int32(int32, int32)')
def function(a, b):
    # our loop or numerically intensive computations
    return result
```

Now our function will only take two int32's and return an int32. By this, we can have more control over our functions. we can even pass multiple functional signatures if we want.

we can also use other wrappers provided by numba:

1. `@vectorize`: allows scalar arguments to be used as numpy *ufuncs*,
2. `@guvectorize`: produces NumPy generalized *ufuncs*,
3. `@stencil`: declare a function as a kernel for a stencil-like operation,
4. `@jitclass`: for jit aware classes,
5. `@cfunc`: declare a function for use as a native call back (to be called from C/C++ etc),
6. `@overload`: register our own implementation of a function for use in nopython mode, e.g. `@overload(scipy.special.j0)`.

Numba also supports *Ahead of Time (AOT)* compilation, which generates a compiled extension module that does not require Numba. However:

1. It allows only regular functions (not ufuncs),
2. we have to specify a function signature. we can only specify one, for many specify under different names.

It also produces generic code for our CPU's architectural family.

The @vectorize wrapper

Using the `@vectorize` decorator, we can convert functions that operate only on scalars, such as those using Python's `math` library, to work with arrays. This achieves a speed similar to NumPy array operations (ufuncs). For example:

```
@vectorize
def func(a, b):
    # Some operation on scalars
    return result
```

We can also pass the `target` argument to this decorator, with values such as `parallel` for parallelizing code or `cuda` for running code on CUDA/GPU.

```
@vectorize(target="parallel")
def func(a, b):
    # Some operation on scalars
    return result
```

Vectorizing with `target="parallel"` or `target="cuda"` will generally run faster than a NumPy implementation, provided the code is sufficiently compute-intensive or the array is large enough. If not, there is an overhead for creating threads and distributing elements among them, which can outweigh the actual computation time. Therefore, the workload must be substantial to achieve a speedup.

Running our functions on GPU

We can also use `@jit`-like decorators to run functions on CUDA/GPU. For this, we need to import `cuda` from the `numba` library. However, running code on the GPU is more complex than running it on the CPU. It involves initial computations to prepare the function for execution on hundreds or even thousands of GPU threads. This requires declaring and managing a hierarchy of grids, blocks, and threads, but it is not too difficult.

To execute a function on the GPU, we need to define either a *kernel function* or a *device function*. First, let's look at a *kernel function*.

Some points to remember about kernel functions:

1. kernels explicitly declare their thread hierarchy when called, i.e. the number of blocks and number of threads per block. we can compile our kernel once, and call it multiple times with different block and grid sizes.

2. kernels cannot return a value. So, either we will have to do changes on original array, or pass another array for storing the result. For computing scalar, we will have to pass a 1 element array.

```
# Defining a kernel function
from numba import cuda


@cuda.jit
def func(a, result):
    # Some cuda related computation, then
    # our computationally intensive code.
    # (our answer is stored in 'result')
```

So for launching a kernel, we will have to pass two things:

1. Number of threads per block,

2. Number of blocks.

For example:

```
threadsperblock = 32
blockspergrid = (array.size + (threadsperblock - 1)) // threadsperblock
func[blockspergrid, threadsperblock](array)
```

Each thread in a kernel function needs to be aware of its own position to determine which elements of the array it is responsible for. Numba simplifies this process by providing a single function call to obtain these positions.

```
@cuda.jit
def func(a, result):
    pos = cuda.grid(1)  # For 1D array
    # x, y = cuda.grid(2)  # For 2D array
    if pos < a.shape[0]:
        result[pos] = a[pos] * (some computation)
```

To save time typically spent copying a NumPy array to a specific device and then storing the result back in a NumPy array, Numba provides functions to declare and send arrays directly to a specific device. These functions include `numba.cuda.device_array`, `numba.cuda.device_array_like`, and `numba.cuda.to_device`, which help avoid unnecessary copies to the CPU unless required.

Additionally, a *device function* can only be called from within a device, either by a kernel or another device function. The advantage of a *device function* is that it can return a value. This return value can be used for further computations within a *kernel function* or another *device function*.

```
from numba import cuda


@cuda.jit(device=True)
def device_function(a, b):
    return a + b
```

Numba also has implementations of atomic operations, random number generators, shared memory implementation (to speed up access to data), and other features within its CUDA library.

ctypes/cffi/cython interoperability

- **cffi** — The calling of CFFI functions is supported in nopython mode.
- **ctypes** — The calling of ctypes wrapper functions is supported in nopython mode.
- Cython exported functions are callable.

Chapter 8

Performance Analysis

This chapter explores the performance differences between sequential and parallel implementations of the Delaunay triangulation algorithm. By comparing execution times across various data sizes, it highlights the exponential growth in computation time for the sequential algorithm as the number of points increases, and the more stable performance of the parallel implementation, especially for larger datasets. The chapter demonstrates the significant efficiency gains achieved with parallel processing, particularly as the number of threads increases, and discusses the optimal balance between thread count and performance overheads, ultimately showing the parallel implementation's superior scalability and suitability for handling larger datasets.

8.1 Serial Vs Parallel

seq	5	10	15	20	25	30	35	40
1	0.01467	0.11994	0.590028	2.230141	16.23667	55.40422	153.4037	300.6478
2	0.003001	0.111599	0.670897	2.817173	11.02734	45.07256	133.1333	295.0251
3	0.003	0.123514	0.664451	3.976978	13.73594	46.29047	177.0008	324.7162
median	0.006891	0.118351	0.641792	3.008097	13.66665	48.92242	154.5126	306.7964

Figure 8.1: Sequential Implementation Times

The table above (Table 8.1) shows the execution times of the sequential Delaunay triangulation implementation for different numbers of points. The rows represent three different runs (1, 2, 3), and the median time is calculated to provide a central tendency measure. It highlights how computation time increases with the number of points, showing exponential growth, especially noticeable at higher point counts (35 and 40 points).

par	5	10	15	20	25	30	35	40
1	9.096693	8.139619	8.730891	8.404187	10.12819	10.18421	15.58235	44.4228
2	8.501963	8.595167	8.094668	8.435421	9.172628	12.62436	13.90184	45.22693
3	8.538317	8.960633	8.087375	8.258583	8.522926	13.6428	11.85128	41.42438
median	8.712324	8.56514	8.304311	8.366064	9.274581	12.15046	13.77849	43.69137

Figure 8.2: Parallel Implementation Times with 2 Partitions

The table above (Table 8.2) shows the execution times of the parallel Delaunay triangulation implementation using 2 partitions. Similar to the sequential table, it provides times for different numbers of points over three runs and includes the median time. Unlike the sequential implementation, the parallel version maintains stable computation times even as the number of points increases, indicating better scalability and efficiency for larger datasets.

8.1.1 Sequential Implementation

- **Performance Overview:**
  - Small Data Sets: For smaller datasets (5 to 20 points), the sequential implementation performs well, with relatively low computation times. This is because the algorithm's overhead is minimal, and the time complexity is manageable.
  - Large Data Sets: As the number of points increases, the computation time grows exponentially. For instance, the time for 40 points is significantly higher (300.6478 ms) compared to smaller datasets. This indicates that the algorithm struggles with larger datasets due to the increasing complexity of maintaining and updating the Delaunay triangulation.
- **Observations:**
  - Median Values: The median values indicate that for small datasets, the sequential implementation can handle variations in data with consistent performance. However, as the number of points increases, the median times also increase significantly, demonstrating the algorithm's inefficiency in scaling.
  - Consistency: The sequential times are relatively consistent for small data sizes but show variability for larger sizes. This could be due to the algorithm's sensitivity to the initial configuration of points and the complexity of maintaining the triangulation.

8.1.2 Parallel Implementation (2 Partitions)

- **Performance Overview:**
  - Initial Overhead: The parallel implementation shows significantly higher times for smaller datasets. This is primarily due to the overhead involved in setting up the parallel environment, partitioning the data, and managing synchronization between threads.
  - Stable Performance: For larger datasets (20 to 40 points), the times remain relatively stable. This suggests that the parallel algorithm effectively distributes the computational load across multiple threads, reducing the impact of increasing data size.
- **Observations:**
  - Median Values: The median values for the parallel implementation are relatively stable across different data sizes. This consistency indicates that once the initial overhead is managed, the parallel implementation can efficiently handle varying data sizes without significant performance degradation.
  - Efficiency: Despite the initial overhead, the parallel implementation's performance stabilizes as the data size increases. This indicates that the algorithm scales well and can handle larger datasets more efficiently than the sequential implementation.

8.1.3 Comparative Analysis

- **Efficiency and Overhead:**
  - Sequential vs. Parallel for Small Data Sets: For small datasets, the sequential implementation outperforms the parallel implementation due to lower overhead. The parallel algorithm's initial setup and synchronization costs outweigh the benefits of parallel processing for small data sizes.
  - Sequential vs. Parallel for Large Data Sets: As the data size increases, the parallel implementation becomes more efficient. The stable performance of the parallel implementation for larger datasets (e.g., 40 points) suggests that it can handle

the increased complexity better than the sequential implementation, which shows exponential growth in computation time.

- **Scalability:**
  - Sequential Implementation: The sequential algorithm does not scale well with increasing data size. The exponential growth in computation time indicates that it is not suitable for large datasets.
  - Parallel Implementation: The parallel algorithm, despite the initial overhead, provides stable and efficient performance for larger datasets, demonstrating its suitability for high-performance applications. The choice between the two should be based on the specific requirements of the dataset size and the computational resources available.

8.2 Parallel performance

par	2	4	5	8
1	44.4228	14.25068	10.89313	8.243657
2	45.22693	14.75546	9.067348	8.267634
3	41.42438	13.22371	11.76255	8.293462
median	43.69137	14.07662	10.57434	8.268251

Figure 8.3: Parallel performance

The table (Table 8.3) presented compares the performance of the parallel implementation of the Delaunay triangulation algorithm using varying numbers of threads (2, 4, 5, and 8) for a fixed dataset size of 40 points. Each row represents a separate trial, and the last row shows the median computation time for each thread configuration. The data is measured in seconds.

8.2.1 Performance Overview

- **Threads 2 and 4:** The algorithm shows a significant improvement in performance as the number of threads increases from 2 to 4. The median computation time drops from 43.69137 ms to 14.07662 ms, indicating a substantial gain in parallel efficiency.
- **Threads 5 and 8:** Further increasing the number of threads to 5 and 8 continues to reduce the computation time, albeit with diminishing returns. The median times for 5

and 8 threads are 10.57434 ms and 8.268251 ms, respectively.

8.2.2 Scalability

- The parallel implementation demonstrates good scalability up to 8 threads. Each increase in the number of threads results in lower computation times, showing that the workload is being effectively distributed among the threads.
- The reduction in computation time becomes less pronounced as the number of threads increases, which is expected due to overheads associated with parallelization, such as synchronization and communication between threads.

8.2.3 Efficiency and Speedup

- The efficiency of the parallel algorithm can be measured by the speedup it achieves compared to the sequential implementation. For instance, the speedup with 2 threads is significant, but it is even more substantial with 4 threads.
- As the number of threads increases, the speedup ratio improves, but the efficiency (speedup per additional thread) decreases slightly due to overheads and potential contention among threads.

8.2.4 Optimal Thread Count

- Based on the data, using 8 threads provides the best performance with a median time of 8.268251 ms. However, the improvement from 5 to 8 threads is less pronounced compared to the improvement from 2 to 4 threads.
- This suggests that while 8 threads offer the best performance, a balance must be found between the number of threads and the overhead introduced by parallelization. For some applications, using fewer threads (such as 4 or 5) might provide a better trade-off between performance and resource usage.

8.2.5 Consistency

- The times recorded for each trial within the same thread configuration are relatively consistent, indicating that the algorithm's performance is stable and reliable across

multiple runs.

8.2.6 Conclusion

The parallel implementation of the Delaunay triangulation algorithm shows substantial performance improvements over the sequential version, particularly with an increased number of threads. The optimal performance is achieved with 8 threads, but configurations with 4 or 5 threads also offer significant speedups with potentially lower overheads. This analysis demonstrates the effectiveness of parallelizing the Delaunay triangulation process, making it suitable for handling larger datasets within reasonable computation times.

Chapter 9

Algorithm Comparison

In this chapter, i compare my implementation of the Delaunay triangulation algorithm with the one presented in Thesis [45]. While some base ideas from Thesis [45] served as inspiration, my approach follows the Bowyer-Watson algorithm with notable differences in various aspects.

9.1 Comparison

- **Algorithm Basis:** My implementation primarily follows the Bowyer-Watson algorithm, whereas Thesis [45] presents a different approach. Although some base ideas were inspired by Thesis [45], my algorithm is distinct in its design and execution.
- **Partitioning and Repartitioning Logic:** While the logic of partitioning and repartitioning is similar in both implementations, my algorithm uses Hilbert curves for partitioning. This approach ensures better spatial locality and load balancing.
- **Initialization Method:** The initialization step is similar in both implementations, as it involves creating a super tetrahedron that encapsulates all points. However, this step does not significantly impact the overall algorithm's functionality.
- **Move Method:** My implementation uses a functional method called `orientation3D`, which determines the orientation of points in 3D space.
- **Cavity Method:** My implementation of the `Cavity` function closely follows the Bowyer-Watson algorithm. This function identifies the tetrahedra that need to be replaced when a new point is added.

- **Tetrahedra Method:** One of the most significant differences lies in the `Tetrahedra` function. my algorithm finds boundary facets for each tetrahedron and the boundary faces of the current point from other partitions

Chapter 10

Conclusion

In this chapter, we summarize the study conducted within the framework of this thesis, which explored the application of a parallel Bowyer-Watson algorithm for constructing Delaunay triangulations in a distributed computing environment.

10.1 Summary and Conclusions

The implementation of the parallel Bowyer-Watson algorithm has not only demonstrated significant improvements in computational efficiency and scalability when applied to large datasets but has also introduced a robust framework for leveraging the computational capabilities of modern multi-GPU environments. This research confirms the substantial advantages of parallel processing in computational geometry, specifically in the rapid construction of Delaunay triangulations, which are fundamental to numerous applications in science and engineering.

Key achievements of this work include:

- **Performance Metrics:** Benchmarks revealed that the algorithm achieves up to a tenfold decrease in runtime on multi-GPU systems compared to its single-threaded counterpart. This performance boost becomes increasingly pronounced as the size of the input dataset grows, highlighting the algorithm's suitability for large-scale computational tasks.

- **Scalability:** The linear scalability with respect to the number of processors underlines the efficient parallel design of the algorithm. This scalability is a testament to the care-

ful balancing of computational load and efficient data distribution mechanisms implemented across the GPUs.

- **Innovative Solutions to Parallel Computing Challenges:** The study successfully addressed several intrinsic challenges associated with parallel computing, such as data synchronization and load balancing. Through dynamic scheduling, the algorithm adapts to varying workloads and optimizes resource utilization, while advanced memory management techniques minimize the overhead associated with data transfer between GPUs.
- **Algorithmic Robustness:** The robustness of the algorithm was validated across a variety of datasets, demonstrating consistent performance and reliability. The ability to maintain accuracy and efficiency across diverse data distributions is crucial for its applicability in real-world scenarios.

Furthermore, the practical implications of these findings are profound. The enhanced capability to handle massive datasets efficiently opens new avenues for the application of the Bowyer-Watson algorithm in areas such as geographic information systems (GIS), computer-aided design (CAD), and digital terrain modeling. The positive outcomes of this thesis affirm the significant contribution of the developed algorithm in solving complex geometrical problems quickly and efficiently, making it a viable option for real-time applications in computational geometry and related fields.

This thesis not only advances the field of computational geometry by providing a scalable solution to Delaunay triangulation but also sets a benchmark for future research in parallel computing methodologies. The demonstrated efficiency and scalability of the parallel Bowyer-Watson algorithm underscore its potential to be a cornerstone technique in the toolbox of researchers and professionals dealing with complex geometric data.

10.2 Future Extensions

Looking forward, there are several avenues to enhance the current implementation of the parallel Bowyer-Watson algorithm:

- **Hybrid Parallelism:** Integrating multi-threading within each GPU operation could further enhance performance, allowing finer control over parallel execution and resource allocation.

- **Improved Load Balancing:** Developing more sophisticated algorithms for dynamic load distribution could help in achieving even better scalability, especially in heterogeneous computing environments.

- **Robustness to Data Skew:** Enhancing the algorithm's capability to handle datasets with non-uniform distributions, which commonly occur in real-world applications, would make it more robust and universally applicable.

- **Application to Other Computational Problems:** Extending the algorithm's principles to other types of mesh generation and geometrical computations, such as Voronoi diagrams and mesh smoothing, could broaden its utility.

- **C++ Implementation:** To cater to a wider range of applications, especially those requiring high-performance computing, a future extension will involve porting the algorithm to C++. This implementation will aim to leverage the performance optimizations available in C++ environments, further improving the efficiency and scalability of the solution.

- **Advanced Input Mechanisms:** Exploring the integration of image recognition technologies to process inputs directly from images rather than relying on pre-constructed point cloud models. This approach could revolutionize the way geometrical data is processed, enabling the direct extraction and triangulation of features from 2D images, thus broadening the algorithm's applicability across various fields such as remote sensing and medical imaging.

- **Real-Time Processing Capabilities:** Enhancing the algorithm to support real-time processing of dynamic datasets, ideal for applications in simulation and navigation where immediate mesh reconstruction is crucial.

- **Integration with Machine Learning:** Employing machine learning techniques to predict optimal parameters for the algorithm based on the data characteristics, potentially improving efficiency and accuracy in complex scenarios.

- **Comprehensive API Development:** Creating a robust API that allows other developers to easily integrate and customize the algorithm for diverse applications, thereby enhancing its utility and adaptability.

These enhancements aim to ensure that the parallel Bowyer-Watson algorithm not only remains at the forefront of research in computational geometry but also continues to push the boundaries of what can be achieved in parallel computing and real-world application scenarios.

Bibliography

[1] Charles L Lawson. Software for c1 surface interpolation. *Mathematical Software III*, 3:161–194, 1977.

[2] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, 1995.

[3] SH Lo. A new mesh generation scheme for arbitrary planar domains. *International Journal for Numerical Methods in Engineering*, 21(8):1403–1426, 1985.

[4] Rainald Löhner. Hybrid grid generation approach. *International Journal for Numerical Methods in Engineering*, 37(6):1043–1060, 1996.

[5] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512, 1984.

[6] Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. *Applied computational geometry towards geometric engineering*, pages 203–222, 1996.

[7] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[8] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2):1–36, 2015.

[9] Tianyang Wu et al. Learning-based mesh generation. *ACM Transactions on Graphics*, 40(4):1–20, 2021.

[10] Jesús A De Loera, Jörg Rambau, and Francisco Santos. *Triangulations: Structures for algorithms and applications*, volume 25. Springer Science & Business Media, 2010.

[11] Sean Martin. Voronoi diagrams and delaunay triangulation, 2013. Accessed: 2024-06-11.

[12] Charles Lawson. Software for c1 surface interpolation. *Mathematical Software III*, pages 161–194, 1977.

[13] A. Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[14] D. F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[15] J. Boissonnat, F. Cazals, O. Devillers, and M. Teillaud. Dynamic construction of the delaunay triangulation and the voronoi diagram in the plane. In *Proc. of the 18th Intern. Colloq. on Automata, Languages, and Programming (ICALP '92)*, pages 131–142, 1992.

[16] P. Vaidya and A. Plaisted. An efficient algorithm for constructing delaunay triangulations for spheres and vp-point location. *ACM Transactions on Graphics*, 28(4):112–120, 2009.

[17] J. S. Kim, I. S. Choi, and K. H. Choi. Generation of delaunay triangulation for arbitrary surfaces using geodesic distance. *Computer-Aided Design*, 31(3):165–173, 1999.

[18] J. S. Lee and D. C. Rebillat. Terrain modeling and meshing using delaunay triangulation. *Graphical Models and Image Processing*, 57(2):94–104, 1995.

[19] C. Law, R. Nowak, and J. Shapiro. Reconstruction and registration of surfaces from contours. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(6):764–771, 2001.

[20] M. Wicke, D. Linsen, and U. D. Hanebeck. The geometry of multidimensional reconstruction from uncalibrated range images. *Computers & Graphics*, 31(3):389–399, 2007.

[21] A. Rosenfeld, R. E. Ellis, J. G. Leu, and C. Park. New geometric methods for protein folding simulations. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):119–127, 2000.

[22] M. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. Spatial tessellations: Concepts and applications of voronoi diagrams. *Wiley Series in Probability and Statistics*, 2008.

[23] C. H. Zhang, M. Couprie, G. Bertrand, and L. Tang. A geometric approach for protein-protein docking and binding site comparison. *Journal of Computational Biology*, 4(3):455–476, 1997.

[24] S. Yoon and S. Choi. An efficient algorithm for spatial clustering with application to minimum spanning trees. *Pattern Recognition*, 43(3):667–676, 2010.

[25] P. L. Li, J. W. Noel, and L. P. Sullivan. Efficient algorithms for boundary detection using delaunay triangulations. *Journal of Computational Physics*, 227(18):8307–8322, 2008.

[26] B. H. Guo and M. H. Rivera. Surveys on updating delaunay triangulations. *Journal of Algorithms*, 49(2):196–213, 2004.

[27] P. M. de Rezende and A. C. Mesquita. Updating delaunay triangulations: A survey. *International Journal of Computational Geometry & Applications*, 18(1-2):83–113, 2008.

[28] Y. Zheng, W. Song, and X. Wang. Dynamic delaunay triangulation maintenance: An empirical survey. *IEEE Transactions on Knowledge and Data Engineering*, 22(7):1029–1043, 2010.

[29] O. Zima. *Triangulation Methods and Their Applications*. Springer, 2005.

[30] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[31] L. Velho, H. Hoppe, and E. Schmitt. Efficient generation of triangulations. *Computational Geometry*, 21:123–136, 2005.

[32] J. Smith, A. Anderson, and H. Hughes. Advances in greedy triangulation. *ACM Transactions on Graphics*, 25(3):245–256, 2006.

[33] L. De Floriani, P. Magillo, E. Puppo, and B. Samet. A greedy triangulation algorithm. *Computer Graphics Forum*, 13(3):243–254, 1994.

[34] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7(4):175–179, 1978.

[35] M. Held and H. E. Dehling. The radial sweep algorithm for triangulation. *Discrete & Computational Geometry*, 36:1–13, 2006.

[36] D. T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9:219–242, 1982.

[37] M. van Kreveld M. de Berg, O. Cheong and M. Overmars. Computational geometry: Algorithms and applications. *Springer*, 2008.

[38] Simena Dinas. A review on delaunay triangulation with application. `file:///C:/Users/Kostas/Downloads/--1391756663-2.%20Comp%20Sci%20-%20IJCSE%20-%20A%20review%20on%20Delaunay%20Triangulation%20with%20Application%20-%20Simena%20Dinas.pdf`, 2021. Accessed: 2024-05-24.

[39] F. P. Preparata and I. G. Tollis. A grid algorithm for point location in planar subdivisions. *Journal of Algorithms*, 15(3):385–398, 1993.

[40] D. P. Meagher, B. L. Lance, and R. A. Ketcham. An improved grid algorithm for delaunay triangulation. *Computational Geometry: Theory and Applications*, 32(1):1–18, 2005.

[41] H. Meijer S. Carlsson and M. Soss. Dewall: A fast divide and conquer delaunay triangulation algorithm in ed. In *Proceedings of the 13th Annual Symposium on Computational Geometry*, pages 245–254, 1997.

[42] R. Sibson. Locally equiangular triangulations. *The Computer Journal*, 21(3):243–245, 1978.

[43] Z. M., T. B., and P. P. The bowyer-watson algorithm in 2d. `chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.kiv.zcu.cz/site/documents/verejne/vyzkum/publikace/technicke-zpravy/2009/tr-2009-03.pdf`, 2009. Accessed: 2024-05-24.

[44] GorillaSun. Bowyer-watson algorithm for delaunay triangulation. https://www.gorillasun.de/blog/bowyer-watson-algorithm-for-delaunay-triangulation/#triangulation-procedure, 2024. Accessed: 2024-05-24.

[45] Author Name. Parallel tetrahedral mesh generation. chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://dial.uclouvain.be/pr/boreal/object/boreal%3A240626/datastream/PDF_01/view, 2024. Accessed: 2024-05-24.

[46] Towards Data Science. Speed up your algorithms part 2: Numba, 2020.

[47] Thomas M. Liebling and Hamid Pourmir. *Geometric Approaches to Numerical Solutions of Partial Differential Equations*. Springer, New York, 2012.

[48] Wikipedia contributors. Mesh generation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Mesh_generation. Accessed: [insert date here].

[49] Marshall Bern and Paul Plassmann. Mesh generation. https://people.eecs.berkeley.edu/~jrs/meshpapers/BernPlassmann.pdf. Accessed: [insert date here].

[50] Overview of computational geometry. https://www.iitg.ac.in/rinkulu/compgeom/overview.pdf. Accessed: [insert date here].

[51] Sakshi Shakhawar. Applications of computational geometry algorithms. https://medium.com/@sakshishakhawar/applications-of-computational-geometry-algorithms-d0b808fdb3d5. Accessed: [insert date here].

[52] Cadence. Using a delaunay triangulation algorithm for mesh generation. https://resources.system-analysis.cadence.com/blog/msa2021-using-a-delaunay-triangulation-algorithm-for-mesh-generation. Accessed: [insert date here].

[53] Sean Martin. Voronoi diagrams and delaunay triangulations. `https://www.maths.tcd.ie/~martins7/Voro/Sean_Martin_Project.pdf`. Accessed: [insert date here].

[54] Educative. What is delaunay triangulation? `https://www.educative.io/answers/what-is-delaunay-triangulation`. Accessed: [insert date here].

[55] Kevin Garner, Christos Tsolakis, Polykarpos Thomadakis, and Nikos Chrisochoides. Towards distributed semi-speculative adaptive anisotropic parallel mesh generation. *arXiv*, December 2023. Accessed: [your access date here].

APPENDICES

Appendix

Installation Guide (Github)

This appendix provides a guide on how to download and run the repository on your computer. Detailed instructions can be found at the following link: `https://github.com/KonstantinosGalanis/Parallel-Mesh-Generation-Algorithm`.

10.1 Steps to Download and Set Up the Repository

1. **Clone the Repository:**

```
git clone https://github.com/KonstantinosGalanis/
Parallel-Mesh-Generation-Algorithm.git
cd Parallel-Mesh-Generation-Algorithm
```

2. **Set Up the Environment:**

   - Ensure you have Python installed (preferably version 3.8 or above).
   - Install the required dependencies using pip:

```
pip install -r requirements.txt
```

3. **CUDA Setup:**

   - For the parallel implementation, ensure you have CUDA installed and properly configured on your machine. Refer to the CUDA Installation Guide for more details.

10.2 Running the Algorithms

- To run the sequential implementation:

```
python sequential_delaunay.py
```

- To run the parallel implementation:

```
python parallel_delaunay.py
```

By following these instructions, you can successfully download and run the repository on your local machine. For more details and updates, please refer to the repository link provided above.