4-1-2011

Construction of Delaunay Triangulations on the Sphere: A Parallel Approach

Veronica G. Vergara Larrea
Florida State University

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

CONSTRUCTION OF DELAUNAY TRIANGULATIONS ON THE SPHERE : A PARALLEL APPROACH

By

VERÓNICA G. VERGARA LARREA

A Thesis submitted to the
Department of Scientific Computing
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2011

The members of the committee approve the thesis of Verónica G. Vergara Larrea defended on April 1, 2011.

Max Gunzburger
Professor Directing Thesis

Anke Meyer-Baese
Committee Member

Janet Peterson
Committee Member

Jim Wilgenbusch
Committee Member

Approved:

Max Gunzburger, Chair, Department of Scientific Computing

Joseph Travis, Dean, College of Arts and Sciences

The Graduate School has verified and approved the above-named committee members.

Para Mikael, después de todo tenías razón. See you soon.

ACKNOWLEDGMENTS

I would like to express my gratitude to my adviser Max Gunzburger for giving me the opportunity to pursue my interests, and allowing me to explore and test my ideas in the development of this project. Also, I would like to thank both Paul van der Mark and Jim Wilgenbusch, who first introduced me to parallel computing and are the reason why I chose to get involved in this area. I also would like to recognize the help that Doug Jacobsen has given me throughout my thesis, his guidance was invaluable. For their understanding and support, I would like to acknowledge all my committee members.

To my friends and colleagues here at the Department of Scientific Computing and the High Performance Computing Center, thank you for all the support and encouragement, without you my experience here would not have been nearly as enjoyable.

Last but not least, I want to say thank you to my family and to Mikael, who despite being so far away always found ways to show me their love and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

This thesis explores possible improvements in the construction of Delaunay Triangulations on the Sphere by designing and implementing a parallel alternative to the software package STRIPACK. First, it gives an introduction to Delaunay Triangulations on the plane and presents current methods available for their construction. Then, these concepts are mapped to the spherical case: Spherical Delaunay Triangulation (SDT). To provide a better understanding of the design choices, this document includes a brief overview of parallel programming, that is followed by the details of the implementation of the SDT generation code. In addition, it provides examples of resulting SDTs as well as benchmarks to analyze its performance. This project was inspired by the concepts presented in Robert Renka's work [26] and was implemented in C++ using MPI.

CHAPTER 1

INTRODUCTION

The main goal of this thesis is to design and implement a parallel software package for the construction of Delaunay triangulations on the sphere. Throughout this document, we will refer to this particular triangulation as the Spherical Delaunay Triangulation (SDT). To better understand SDTs we will first describe the more familiar case of Delaunay Triangulations and map those concepts to the spherical case.

Delaunay Triangulations have been around for several decades now, they were first introduced by Boris Delone in 1934 in the paper *Sur la sphère vide (On the empty sphere)* that Delone wrote in memory of his mentor George Voronoi [15]. The main idea presented describes what is now known as the empty circumsphere criterion in Delaunay Triangulations that we will review in the next chapter.

In conjunction with their dual (i.e. Voronoi Tessellations), Delaunay Triangulations play a major role in a variety of disciplines ranging from computational geometry to atmospheric sciences. Both are regularly used today, for instance, in Scientific Computing as grids for finite element simulations; in Computer Graphics for surface and volume rendering; in Physics for N-body simulations; in Computational Biophysics: for protein folding prediction; in Computer Science for data mining; and in many other fields. Given the number of applications that rely on this particular kind of nonuniform grids, it is highly important to study and improve the efficiency of the algorithms used to generate them.

Thanks to the advances in high-performace computing, scientists are now capable of developing much more complex and accurate models by taking advantage of parallel programming. Parallel programming has allowed researchers to obtain results much faster than it was possible when using one uniprocessor machine. However, the models are limited by the discretization scheme used, their accuracy directly depends on the level of resolution of the underlying grid. Motivated by these facts, in recent years, new algorithms to efficiently construct and refine Delaunay Triangulations and Voronoi Tessellations have appeared [27, 7] with the aim of constructing meshes tuned to the specific problem studied (e.g. mesh refinement).

One case of Delaunay Triangulations that is of particular interest to the ocean and atmospheric sciences community is the Spherical Delaunay Triangulation (and its dual the Spherical Centroidal Voronoi Tessellation), which is a Delaunay Triangulation

whose generating points are constrained to the surface of the sphere. The main reason being that this quasi-uniform grid serves as a much more accurate and flexible representation of the surface of the Earth, because it can be locally refined. Also, it is appealing because it does not have the singularities present in the more traditional longitude-latitude grids. While SCVTs have gained popularity in the last few years, software packages are still scarce and of the few we have been able to find (see Ch. 3), most are serial implementations [1, 26]. We believe that a parallel implementation package would be a great contribution to the scientific community, whose aim is to be able to construct grids with millions of generators, a task that is very time-consuming with the current serial software available.

This project is inspired by Robert Renka's STRIPACK [26] serial software package and it is motivated by current research done on the construction of Spherical Centroidal Voronoi Tessellations (SCVT). In its early stages, the deterministic version of the code that generates SCVTs relied on the STRIPACK package to build an SDT as an intermediate step in the SCVT construction. As pointed out in [31], this triangulation step is the most time-consuming in the algorithm, and consequently we chose it as the best candidate to provide an improvement in performance. We will concentrate our effort in implementing an alternative parallel method to build the intermediate SDT with the hope that this software package can be used in the near future for the construction of SCVTs.

This thesis starts with a detailed description of Delaunay Triangulations, in Chapter 2, that highlights the specific case of SDTs. Then, Chapter 3 presents the different algorithms currently used in the field to construct Delaunay Triangulations and briefly discusses the packages available that implement them. Chapter 4 provides a brief introduction to parallel programming, which is followed by an in-depth description of our parallel implementation in Chapter 5. The last few chapters show our results (Chapter 6), summarize our conclusions (Chapter 7) and propose possible related areas for future exploration.

CHAPTER 2

DELAUNAY TRIANGULATIONS

Here, we first provide a formal definition of a Delaunay triangulation and state its main properties for the general case in d-dimensional space. For a more detailed description of DTs, see[14, 32]. Then, we apply these concepts to the specific case of Delaunay triangulations constrained to the surface of the unit sphere in $\mathbb{R}^3$, that we will refer to as Spherical Delaunay Triangulations (SDTs).

2.1 Delaunay Triangulation

Let $\mathcal{S}$ be a set of points in a d-dimensional Euclidean space E^d, and let $\Omega(\mathcal{S})$ represent its convex hull. Then, a triangulation T can be defined as the decomposition of $\Omega(\mathcal{S})$ into the set of simplices, $\{s_i\}_{i=1}^{k}$, such that:

- A point $p \in E^d$ is a vertex of a simplex if and only if $p \in \mathcal{S}$.
- For any pair of simplices $s_i, s_j \in T$, $s_i \bigcap s_j = \emptyset$.
- The union of all simplices $s_i \in T$ completely covers the domain bounded by $\Omega(\mathcal{S})$.

Now, let T_D be a triangulation with the property that every hypersphere circumscribed around the $d+1$ vertices of each d-simplex s contains no points other than the vertices forming s. Then, we call T_D a Delaunay Triangulation.

Definition 1 *We say a triangulation T_D of a set $\mathcal{S} \in E^d$ is a Delaunay triangulation, if and only if the circumsphere of every simplex $s \in T_D$ contains no other point in $\mathcal{S}$ but the vertices that form s.*

Furthermore, we say that T_D is uniquely defined if no $d+2$ points in $\mathcal{S}$ share a common hypersphere. A triangulation that satisfies this criterion also has the property that it maximizes the minimum angle of all its simplices.

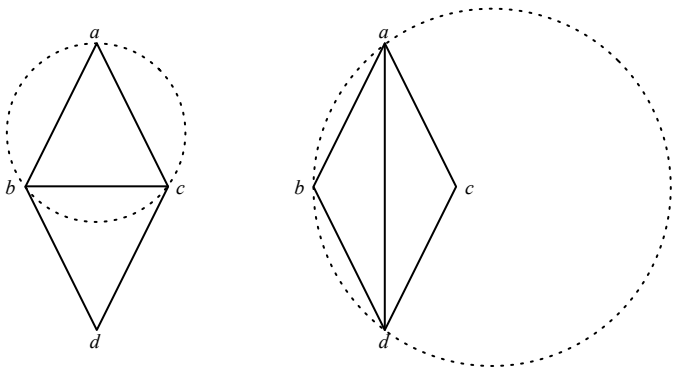

Figure 2.1: Delaunay triangulation (left). Non-Delaunay triangulation (right).

2.2 Constrained Delaunay Triangulations

In many applications, Delaunay Triangulations are used to describe a particular surface or volume, in those cases, we refer to the triangulation as a Constrained Delaunay Triangulation (CDT) because the set of points $\mathcal{S}$ is constrained to a closed domain, that we will call Γ. In Constrained Delaunay Triangulations, the Delaunay criterion needs to be relaxed to ensure that all edges in the triangulation lie in Γ.

Another kind of CDTs[1] occurs when in addition to the generating point set $\mathcal{S}$, we have a set of constraining non-crossing edges $\mathcal{E}$. In this case, the goal is to obtain the triangulation that resembles the Delaunay Triangulation most and that includes all the edges in $\mathcal{E}$ [12].

2.3 Delaunay Triangulations on the Sphere

One particular case of CDTs, of great interest for this thesis, occurs when the constraining domain Γ is the surface of the unit sphere. We will refer to this special case as a Spherical Delaunay triangulation (SDT), that is obtained when the points that we want to triangulate lie on the surface of the unit sphere. Since SDTs are the main focus of this thesis, let us describe them in greater detail.

Let $U \subset \mathbb{R}^3$ denote the set of points that lie on the surface of the sphere:

$$U = \{p \in \mathbb{R}^3 | \|p\| = 1\}, \tag{2.1}$$

where $\|\cdot\|$ represents the Euclidean norm. Then, we define an arc length metric

[1]Most of the literature considers only this kind of Delaunay Triangulation as a CDT, but we feel it is appropriate to distinguish it from the one mentioned in the previous paragraph.

$d(p,q) = \arccos(\langle p,q\rangle) \forall p,q \in U$, where $\langle\cdot,\cdot\rangle$ is the scalar product of p and q. Now, we can introduce a slight modification to the standard definition of a Delaunay triangulation (Def. 1) so that it applies to the unit sphere.

Definition 2 *Let $\mathcal{S} \subset U$ be a set of points $\{p_i\}_{i=1}^N$ on the unit sphere, then for $N \geqslant 3$, a Delaunay triangulation T_D is defined if the empty circumsphere criterion is satisfied for all spherical triangles covering the convex hull $\Omega(\mathcal{S})$.*

It is important to note that if all the points in $\mathcal{S}$ lie inside one hemisphere of U, then the convex hull $\Omega(\mathcal{S})$ has two regions, an interior smaller region and an exterior larger region and that the union of the two cover all of U. On the contrary, if the points lie in both hemispheres then $\Omega(\mathcal{S}) = U$.

In order to enforce and check that the triangulation satisfies the Delaunay criterion, we need to be able to find the circumcenter of a spherical triangle, and furthermore one that lies on the unit sphere. Using vector calculus, one can show that the circumcenter c_{ijk} of a triangle $(p_i, p_j, p_k) \in T_D(\mathcal{S})$ for $p_i, p_j, p_k \in \mathcal{S}$ is given by:

$$c_{ijk} = \frac{(p_j - p_i) \times (p_k - p_i)}{\|(p_j - p_i) \times (p_k - p_i)\|} \tag{2.2}$$

which allows us to define a circumcircle C_{ijk} with radius r_{ijk} using the arc length metric as the set of points p on the unit sphere where $d(c_{ijk}, p) = r_{ijk}$ [26]. This definition will be very useful later during the process used to construct the triangulation.

Another important relationship commonly used to build Delaunay Triangulations (on any dimension), is known as the "is left" relationship between a point p and an edge $p_1 \to p_2$, which is defined in Def. 3. In general, the "is left" relationship is used to determine whether points are oriented in a counter-clockwise or clockwise fashion.

Definition 3 *Let p, p_1, p_2 be three points in a set $\mathcal{S}$, and let $p_1 \to p_2$ be an edge between the two points. Then we say that point p "is left" of $p_1 \to p_2$ if the determinant of the matrix formed by p, p_1, p_2 is greater than or equal to zero.*

On the sphere, the "is left" relationship can be interpreted as a point lying in the left hemisphere defined by the great circle that passes between p_1 and p_2.

2.4 Delaunay Triangulation Dual

As it is widely mentioned in the literature Delaunay Triangulations are the dual graphs of Voronoi Tessellations. In other words, there is a one-to-one correspondence between nodes in the triangulation and regions in the tessellation, a similar relationship between the edges of the triangulation and the shared boundaries of each region and finally also a one-to-one correspondence between the circumcenters of the triangulation and the Voronoi vertices. This particular correspondence means that a

Delaunay triangulation can be obtained from a Voronoi Tessellation and vice-versa, a fact that has proven to be very useful since the transformation can be done in $O(n)$ time. The duality between the two becomes evident when we superimpose both graphs, as shown in Fig. 2.2

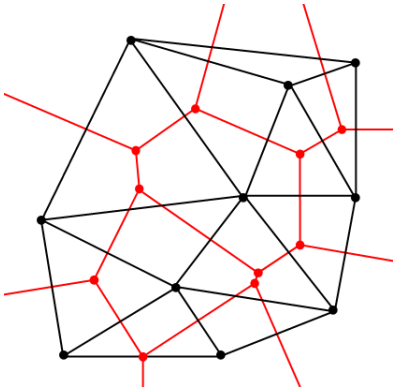

Figure 2.2: Voronoi-Delaunay duality

This property does not necessarily translate to the spherical domain. For instance, in the case when the SDT does not cover the entire sphere, additional Voronoi vertices will have to be selected from outside the SDT. The peculiarities of this transformation are discussed in greater detail in [26], however, these are outside the scope of this thesis.

2.5 Delaunay Triangulation Applications

We briefly mentioned earlier that Delaunay Triangulation are widely used. Here we want to emphasize a few of the different disciplines that rely on them, and the impact of the contribution they make with specific examples. The literature mentions Delaunay triangulations have been used for the following methods:

- Interpolation
- Mesh refinement
- Grid generation

Unsurprisingly, the above techniques are used in many fields including climate modeling, surface reconstruction, rendering, product testing, computer aided-design, nearest neighbor search, game development, collision detection, neural networks and many others.

CHAPTER 3

ALGORITHMS TO GENERATE DELAUNAY TRIANGULATIONS

In this chapter we describe the different classes of algorithms that are used in the literature to generate Delaunay Triangulations. There exist a variety of approaches and implementations for the construction of Delaunay Triangulations, however, these algorithms can be classified into five groups: incremental, divide and conquer, gift-wrapping, sweepline and lifting [28].

3.1 Incremental Construction Algorithms

Using an incremental construction algorithm the Delaunay triangulation is generated by adding points one by one to an existing Delaunay triangulation. After adding each point, the Delaunay criterion of all the triangles is checked to ensure that the new triangulation still satisfies the Delaunay definition. Many implementations are based on incremental construction, the Bowyer-Watson algorithm [10, 30] being one of the most popular methods. A slight variation of Bowyer-Watson known as the Lawson's swapping algorithm described in [21, 22] is of particular interest to us because it is the base for our parallel algorithm.

3.1.1 Bowyer-Watson Algorithm

The algorithm is fairly straightforward, it starts with one "infinite" triangle containing all the points $p \in \mathcal{S}$, then it adds one point at a time. At each step, the algorithm finds and deletes all the triangles in the triangulation T_D whose circumcircle contains p. Then the convex cavity resulting from the deletion of the triangles is re-triangulated, see Fig. 3.1.

Removing all triangles that do not satisfy the Delaunay criterion at every insertion guarantees that the triangulation will remain Delaunay at each iteration.

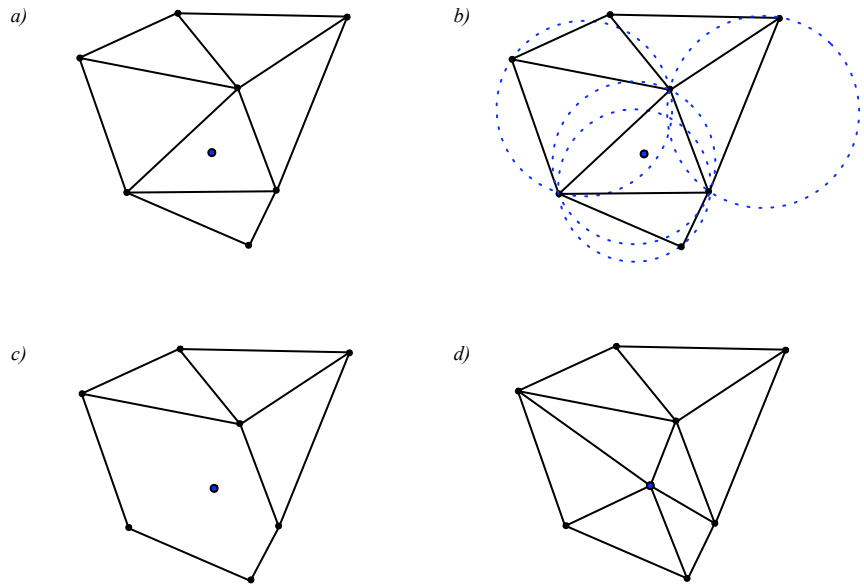

Figure 3.1: Bowyer-Watson Insertion

a) Point is added. *b)* Triangles whose circumcircle contains the point are found. *c)* Form a cavity by deleting all the non-Delaunay triangles. *d)* Connect the point to all the vertices in the cavity.

Algorithm 3.1 Bowyer-Watson Algorithm

Create an "infinite" triangle.
for every $p \in \mathcal{S}$ **do**
  Find all triangles in T_D whose circumcircle contains p and delete them
  Create edges from p to each vertex in the convex cavity formed in the previous step
end for
Remove the "infinite" triangle.

3.1.2 Lawson's Insertion Algorithm

As we mentioned, Lawson's algorithm is similar in some ways to Bowyer-Watson. The main difference lies in the fact that instead of deleting all non-Delaunay triangles, it makes uses of an edge flipping scheme to ensure that the triangulation remains Delaunay at every step. The edge flipping scheme is depicted in Fig. 3.2. It is important to note that when a flip is performed the Delaunayhood of the two new resulting triangles needs to be checked, this is done by the recursive *checkEdges* function shown in Alg. 3.3

Algorithm 3.2 Lawson's Insertion Algorithm

Create an "infinite" triangle.
for $p_i \in \mathcal{S}$ **do**
  Find triangle $\triangle abc$ in T_D that contains p
  Divide $\triangle abc$ into three subtriangles: $\triangle pab, \triangle pbc, \triangle pca$
  for each subtriangle $\triangle \in \{\triangle pab, \triangle pbc, \triangle pca\}$ **do**
    $checkEdges(p, \triangle)$
  end for
end for
Remove the "infinite" triangle.

Algorithm 3.3 $checkEdge(p, \triangle)$ function

if $\triangle$ is not Delaunay **then**
  Flip edge between $\triangle$ and the adjacent triangle opposite to p, forming $\triangle_1, \triangle_2$
  $checkEdge(p, \triangle_1)$
  $checkEdge(p, \triangle_2)$
end if

3.2 Gift-Wrapping Algorithm

The gift-wrapping algorithm, also known as the incremental search algorithm, starts with a 1-simplex (an edge) between two points in $\mathcal{S}$ whose circumcircle contains no other points in $\mathcal{S}$. Then, it searches for the next point to be added by selecting one that will form a triangle that meets the Delaunay triangle criterion. The same idea is repeated for every boundary edge of the current triangulation until all the points have been processed. For this algorithm, the main concern is how to efficiently find the next point to be added. One usual way to address this issue is to pick a point at random and then check if it lies inside of the existing triangle's circumcircle. If it does, then it becomes the next candidate, otherwise a new triangle is formed. A few authors have implemented this algorithm including Dwyer [16] and Tanemura [29].

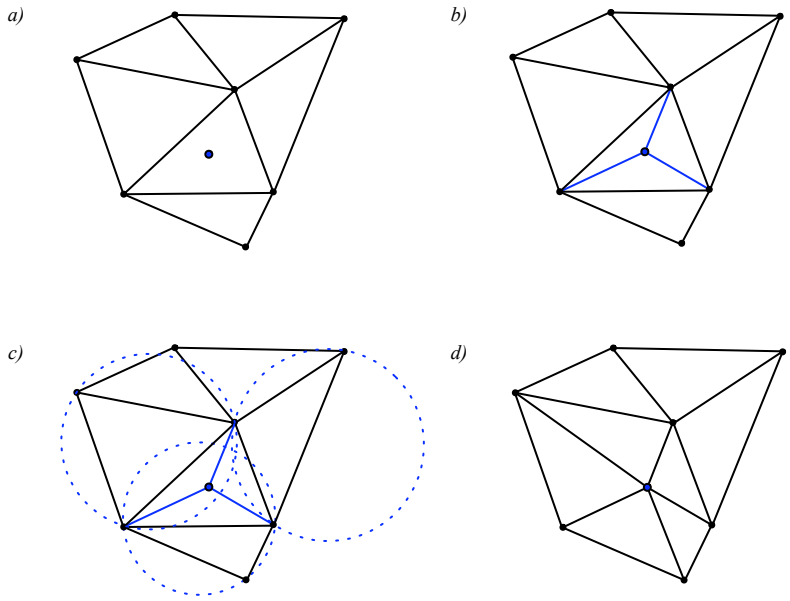

Figure 3.2: Lawson Insertion algorithm

a) Point is added. *b)* Point is connected to the vertices of the triangle containing it. *c)* Adjacent triangles are checked. *d)* Edges forming non-Delaunay triangles are flipped.

3.3 Sweepline Algorithm

As it name suggests, it uses a moving boundary to sweep across the plane and build the Delaunay triangulation, by adding legal edges when it encounters a site in $\mathcal{S}$, or when it reaches the top of the circumcircle of a triangle already in the front. This algorithm was first presented by Fortune for the construction of Voronoi tessellations [19] but because of the duality between the two, it is also used to build Delaunay triangulations. The sweepline algorithm has an expected time $O(n \log n)$.

3.4 Divide and Conquer Algorithms

The idea behind divide and conquer is fairly simple, as it names states, it consists of dividing the set of points $\mathcal{S}$ into two subsets. Then each subset is recursively divided until there are only three or fewer points in each subdivision. Once the last level of recursion has been reached, the triangulation is formed by merging neighboring triangulations. This approach has been used in the literature for building Delaunay triangulations in the plane, and one of the most famous examples is the one implemented by Guibas & Stolfi [20].

3.5 Lifting Algorithms

One clever idea used to construct Delaunay triangulations in any dimension d, consists in projecting the points onto a paraboloid in $(d+1)$-dimension and computing their lower convex hull (see Fig. 3.3).

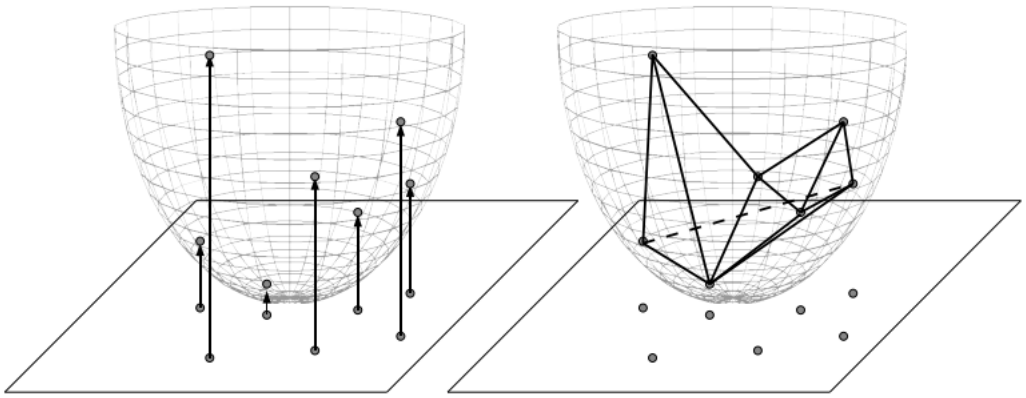

Figure 3.3: Lifting points in $\mathbb{R}^2$ to the paraboloid in $\mathbb{R}^3$ [32].

Once the convex hull has been constructed, all that is required to obtain the Delaunay triangulation is to project the edges back to their original d-space. This

reversal can be done in O(1) time by simply dropping the coordinate in the $(d+1)$ component. This approach, first pointed out by Brown[11] with respect to Voronoi Diagrams and by Edelsbrunner[17] and Seidel with respect to Delaunay triangulations, allows us to use any convex hull algorithm to solve the Delaunay triangulation problem.

3.6 Available Implementations for the Construction of Delaunay Triangulations

To our knowledge there exist a vast number of software packages that have been developed to compute Delaunay Triangulations (and Voronoi Tessellations). It would be impossible to name all here but we will mention some that we have found to be interesting and have been repeatedly mentioned in the literature because they have demonstrated to have a good performance.

3.6.1 Triangle

Jonathan Shewchuk's *Triangle*, is a fairly recent implementation in C for Delaunay triangulations in two dimensions. It is very robust as it allows the user to choose which algorithm to apply for the construction of the triangulation. Also, it is impressively fast, according to the *Triangle* website[27] a Delaunay triangulation of a million vertices can be computed on a Pentium II machine in less than 21 seconds! This software is already being used to compute Delaunay triangulations on the sphere (as an intermediate step to construct SCVTs) by Doug Jacobsen, who through stereographic projection maps the sphere into the plane and builds the Delaunay Triangulation with *Triangle*.

3.6.2 CGAL

The *Computational Geometry Algorithms Library (CGAL)* is a powerful C++ library[1], that implements a wide range of algorithms that are commonly used in computer graphics and computational geometry, including Delaunay Triangulations in two and three dimensions. One other thing we find appealing about *CGAL* is that it is an open source project and it is being actively developed, the most recent update was done in the last quarter of 2010 and a parallel version of their implementation seems to be in the works.

3.6.3 QHull

The *QHull* software package based on Barber's QuickHull algorithm[7, 8] that uses the lifting approach to compute the Delaunay triangulation of a given point set in d-space, by projecting the set to $d+1$ space and computing its convex hull. *Qhull* provides tools to compute Voronoi diagrams, convex hulls in up to nine dimensions.

3.6.4 TRIPACK

This software package is also known as *ALGORITHM 751* and was developed by Robert Renka in 1996[25]. It was originally written in Fortran 77 and it uses an incremental construction algorithm to compute constrained and non-constrained Delaunay triangulations in two dimensions.

3.6.5 STRIPACK

STRIPACK was also written by Robert Renka who modified the two dimensional version of the code in order to construct algorithms directly on the surface of the unit sphere. Just as its predecessor, STRIPACK[26] implements an incremental construction algorithm by using a linked list structure to store the relationships between points in the set to be triangulated. In Renka's implementation, every time a node is added it can either be inside the current triangulation or lie outside of it. Those two cases are handled differently, in the first case Lawson's algorithm is applied by using the edge flipping algorithm to ensure that the Delaunay criterion is preserved. In the latter case, the point is connected to the triangulation by adding edges from it to each point in the convex hull that is visible from the new point. Up to recently, STRIPACK has been the software package of choice to generate Delaunay Triangulations on the sphere because it was proven to perform reasonably well for a variety of point distributions. According to its author, STRIPACK has an $O(N \log N)$ complexity when the points are randomly inserted and a worst case of $O(N^2)$ when the points are arranged in latitude-longitude order.

CHAPTER 4

PARALLEL PROGRAMMING

Now that we have provided a detailed description of the problem this thesis tackles, we want to move on to presenting our method to solve it. However, before we can do that, it seems appropriate to start with a brief introduction to parallel programming, which will help clarify our choices during the design and implementation of our algorithm.

Parallel computing has been in the scene since the late 60's, when the first supercomputers designed by Seymour Cray were introduced to the public by the CDC [4, 6, 24]. In their early stages, the supercomputers available were mostly vector machines and they displayed parallelism at the instruction level. Shortly after, different architectures were designed and introduced including massively parallel computers (MPPs) and shared memory machines (multicore). The increasing number of machine architectures led the high performance community to develop libraries and interfaces that would provide an architecture independent standard and that would also facilitate the implementation of parallel programs in these highly complex systems. Today, parallelism is most commonly achieved in two different ways. One, suited for shared memory systems, is OpenMP that uses preprocessor directives to parallelize sequential programs. The other one, mainly used in distributed systems, is to use the MPI interface based in the message passing model. Both methods will be discussed in more detail in this chapter.

The main goal of parallel programming is to be able to take advantage of the state of the art multicore and multiprocessor architectures to solve problems faster and cheaper. More specifically, we say that a parallel approach is advantageous when we can do the same amount of work in less time, or we can do more work in the same amount of time. To quantify the performance of a parallel approach measurements like speed-up and efficiency are often used.

In order to solve a given computational problem using parallel computing, it needs to be broken into independent pieces that multiple computing resources can process concurrently. The resources utilized are usually a collection of CPUs that either belong to the same computer (multicore) or are connected through a network (computer clusters). In the next section we will see how different architectures are classified and briefly compare them.

4.1 Parallel Computers Architecture Classification

Over the years, parallelism has been classified in multiple ways, but one of the earliest and now most commonly used classifications was given by Flynn in 1966 [18], and it is known as Flynn's Classical Taxonomy. Using this classification, all computers architectures fall in one of four different groups depending on the number of instruction sets and data streams they can process, these groups are shown in Table 4.1.

Table 4.1: Flynn's Taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Out of these architectures, the most common one available to the general public used to be the SISD because most personal computers were only capable of processing one instruction set a time. However, in recent years, the push for multicore processors changed that. Most personal computers nowadays come with more than one processor and are capable to run parallel programs right off the shelf.

In contrast, among high-performance enthusiasts and in the scientific community the most common kind of parallel computer was and continues to be the MIMD, although vector machines remain very popular[1]. MIMD systems can be of very diverse nature and include computers connected via a local network, specialized machines with many cores and anything in between. Although not universally accepted, MIMD machines can be of two types: multiprocessors to represent machines that have shared memory, and multicomputers in which each computer has its own private memory.

Of all the classes, MISD is probably the least common and least successful, one of its main characteristics is that all processing elements share a global clock and perform different calculations synchronously on the same data stream. The lack of applications for this type of architecture is probably the reason why it is often neglected.

SIMD architectures, on the other hand, did become popular because they are fairly easy to program and scale well. In SIMD machines, one control unit sends calculation requests to all the data processing units available, which synchronously perform the same operation on their own data. The main difference between SIMD and MIMD computers is that in the latter, each processing element can perform different actions at any given time (i.e. they work independently).

[1]In the literature, there is some debate in regards to which category in Flynn's Taxonomy do vector machines fall. For the most part, they are considered to be SIMD machines.

4.2 Parallel Memory Models

Parallel machines inside the MIMD class can be further divided into two groups depending on the memory layout they have. Namely, those groups are: the distributed memory model and the shared memory model.

4.2.1 Distributed Memory Model

The distributed memory model is commonly seen in large computer clusters, where multiple computers are connected through a network, and because each one of them have private address spaces they require an explicit means to communicate data between them. Fig. 4.1 depicts the basic structure of a distributed memory machine. As we can see, the individual processors have their own memory, therefore, in order to share data it must be sent across the network. The message passing model was introduced to make this communication possible and as its name suggests, requires processors to send data they want to share in the form of messages. It is a fairly simple model, however, writing programs under this model is not as straightforward because it requires a considerable amount of effort during the design of the data transfer in order to avoid deadlocks. One interface that has become the standard in the development of parallel program for distributed memory architectures is the Message Passing Interface (MPI).

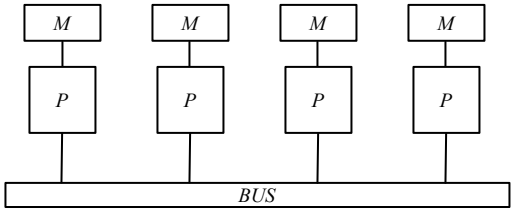

Figure 4.1: Distributed Memory Architecture

Message Passing Interface (MPI). The Message Passing Interface (MPI) originated in the early 90's and was developed to exploit the computing power of massively parallel processors (MPPs). In the high performance community, specially, MPPs are particularly popular because they allow scientists to solve problems that are too large to be handled by one machine. The set of library routines that MPI provides facilitate the development of parallel programs that can be run in large distributed memory machines. One possible drawback is that, in general, sequential programs parallelized using MPI need to be completely restructured, which can be a daunting task. However, due to its portability and because it is well suited for computer clusters with hundreds and even thousands of nodes, it is the method of choice for large scale modeling.

4.2.2 Shared Memory Model

In the shared memory model the collection of processors share a common memory, and thus, do not need to explicitly communicate data between them. Fig. 4.2 shows the memory layout for the simplest version of this kind of architecture. In reality, systems are much more complex and they usually have local (private) caches at each processor and may also include a bigger shared cache.

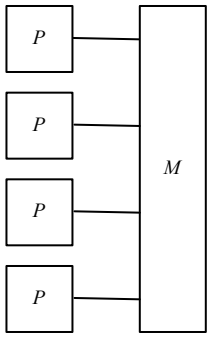

Figure 4.2: Shared Memory Architecture

Removing the need of explicit communication of data between processors is definitely an advantage especially during the development of the parallel application, however, it comes at a cost. Since the memory is shared by all processing units, other issues need to be addressed in order to maintain a consistent view of memory. A common complication in shared memory programming is known as a race condition and can occur when multiple processors access the same shared variable simultaneously. The result of such an operation becomes non-deterministic because depending on the exact order in which the variable was accessed, the value obtained will be different. For example, if two threads are competing for one shared variable the final result could be the value computed by the first thread or the value computed by the second thread or even worse a mixture of the results of both threads. In order to avoid race conditions and to maintain cache and memory coherence, synchronization mechanisms such as mutexes, locks and critical sections, are often used in shared memory programming to ensure that only one process is able to update the value of a shared variable at any given time.

OpenMP. OpenMP is one of the most common application programming interfaces (APIs) used today for shared memory parallel programming. It was first introduced to the public in the late 90's at the Supercomputing conference and was developed with the intention to provide an architecture-independent way of programming shared memory machines. Writing parallel programs with OpenMP in a shared memory computer is much simpler than programming in a distributed memory ar-

chitecture. Parallelism with OpenMP is provided through the use of directives that are placed directly in the sequentiial code and tell the compiler how to process and distribute sections of the code that are to be run in parallel.

SMP revisited. In recent years, the well known symmetric multiprocessing (SMP) architecture has been revamped, before it used to refer only to identical machines that shared one main memory, nowadays , however, it is also used to refer to any collection of machines that share a global address space (real or virtual). This became possible thanks to dynamic systems developed to allow large collections of computers to share the same address space via virtualization. This cutting edge technology allows for simpler programming because it delegates the task required to maintain a consistent view of memory among processors to the hardware, as opposed to the distributed memory solution where all data needs to be explicitly transferred among processors.

4.3 Parallel Programming Models

In the previous section we presented how parallelism can be classified according to the architecture of the system that will be used. Here, we will classify parallel computing according to the kind of parallel approach implemented. There are two distinct parallel programming models: one that focuses on different data sets, and another that creates components that specialize on independent tasks.

4.3.1 Data Parallel Model

In the data parallel model, as its name suggest, the input (data) for the computational problem is divided into chunks. Another characteristic of this model is that every processor runs the same set of instructions except that they do so in different subsets of the data, each one has its own input and produces its own subset of the output. To complete the computation the individual outputs are collected to form the final result. This model can be used in both SIMD and MIMD architectures.

4.3.2 Task Parallel Model

In contrast to the model described above, task parallelism is achieved by identifying sections of the instruction set that can run independently. The program is divided into subsets of instructions (tasks) that can be computed concurrently by different processors on the entire output. This model fits well in MISD and MIMD architectures.

4.4 Measuring Performance

In the parallel computing world, two measures are used to determine whether a parallel approach is beneficial to solve a particular problem. These measures are speedup (S_P) and efficiency (E_P) and we define them below.

4.4.1 Speedup

As mentioned earlier, the expected result of using a parallel approach over a serial one is that we will be able to solve a problem faster. Measuring speedup, we can actually determine whether we are achieving this desired result or not.

Definition 4 *Let t_S be the total time necessary for a serial algorithm to complete executing a program, and let t_P be the time it takes to solve the same problem using P processors. Then, speedup S_P is defined as:*

$$S_P = \frac{t_S}{t_P} \tag{4.1}$$

From the definition we can see that in the ideal case we should obtain an speedup $S_P = P$. However, in practice, this is rare and when it does happen it is usually seen for embarrassingly parallel problems. The reason is that t_P accounts for both communication and computation time, as given by Eq. 4.2, and in general t_{comm} grows as we increase the number of processors assigned to solve the problem.

$$t_P = t_{comp} + t_{comm} \tag{4.2}$$

4.4.2 Efficiency

Efficiency allows us to quantify how much of the total parallel compute time, is each processor actually working versus how much time it is idle, possibly waiting for other processors.

Definition 5 *For a given algorithm that uses P processors and has a speedup S_P, the parallel efficiency E_P is given by:*

$$E_P = \frac{S_P}{P} \tag{4.3}$$

In the ideal case, when $E_P = 1$, all processors are being utilized throughout the computation and there is no waste of resources. However, in practice, this is highly unlikely due to the fact that synchronization needs to occur.

4.5 Other Important Factors

Another major concern in parallel programming is whether or not a parallel algorithm is scalable, where we refer to as an scalable algorithm as one whose efficiency does not go to zero as we increase the number of processors to infinity[23]. Scalability is an important factor because it determines whether or not there will continue to be a benefit in assigning more processors to run one parallel program. In general, fully scalable codes are difficult to achieve, except in the case where the problem studied is embarrassingly parallel. The reason is that having a very large number of processors implies that more communication needs to occur between them and therefore both efficiency and speedup are compromised.

Parallel programming has played a huge role in the development of more accurate computer models and it will continue to do so in the future. Regardless of how fast individual processors get, parallel machines are a better option because they can collect the compute power of a very large number of resources at a cheaper monetary value. We believe parallel programming will soon be the norm, especially in the sciences, and while this chapter is by no means exhaustive, it showcases a few of the most popular choices available with the goal of encouraging others in reading more about the subject [24, 6].

CHAPTER 5

PARALLEL IMPLEMENTATION FOR SDT CONSTRUCTION

In this chapter we describe the design and implementation of our parallel algorithm for the construction of Spherical Delaunay Triangulations. As mentioned earlier, our algorithm is based on the software package STRIPACK written by Robert Renka. We want to emphasize that our algorithm is not a direct translation of STRIPACK, in fact, along the way we introduced modifications we believe are better suited for a parallel environment. One main difference lies in the fact that our code uses the "infinite" triangle approach whereas STRIPACK allows point insertion outside the current triangulation. Because of this difference, we were able to explore an alternative mechanism for point location to that presented in Robert Renka's software package. The "infinite" initial triangulation method used allow us to easily define the sphere partitions necessary for a parallel algorithm.

We will start with a description of the STRIPACK software package, in specific we are concerned with its `trmesh` function, which is in charge of computing the Spherical Delaunay Triangulation of a given set of points in the sphere. Then, we present how we adapted STRIPACK so that it can be used in a parallel environment.

5.1 More on STRIPACK

As described by its author, STRIPACK, is a software package that uses an incremental algorithm to build Delaunay triangulations and Voronoi diagrams on the sphere. The package uses a linked list data structure to store the edge relationship between nodes, each node has its own adjacency list that contains all the nodes that are connected to it.

In order to build a Delaunay triangulation, all that is necessary is to call the function `trmesh` by giving it the list of nodes as input. As a preprocessing step, the triangulation function computes the nodes that are nearest, second nearest, third nearest, etc. to each unprocessed node. This allows the package to locate the point with respect to the triangulation faster and obtain an $O(N \log N)$ expected time. The following step simply calls the `addnod` function for each unprocessed node, which

calls three major subroutines: `trfind` and `intadd` or `bdyadd`, depending on whether the node is located inside the triangulation or outside.

5.2 Going parallel

Before any code can be written, a few decisions in regards to the design of the algorithm need to be made. First, we need to determine if we will use a task parallel or a data parallel model. For this particular case, since most of the steps in the algorithm are dependent on each other, we chose the latter. Step two is to decide whether we will use a shared memory or a distributed memory approach. Since both have advantages as we saw in Ch. 4, we opted to use two different techniques: exclusively distributed memory and a hybrid distributed/shared model. In the next sections we will provide the details of how our algorithm was designed.

5.2.1 Dividing Up the Sphere

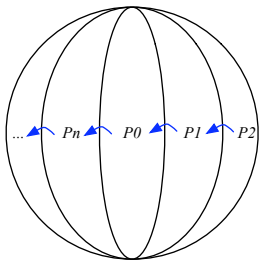

Figure 5.1: Divided Sphere

First, since we are using a data parallel model, we need to implement code that it is capable to run in a section of the sphere. We choose to follow the example of [20] for two dimensional Delaunay triangulations, that divides the set of points in *vertical stripes* along the x axis. The main difference is that in our case we are partitioning the sphere along the longitudinal axis ϕ, as shown in Fig. 5.1. Each section will contain points whose longitudinal coordinate lies in the interval $[\phi_0, \phi_{max}]$, imposing no restrictions in the latitudinal coordinate θ. The number of divisions is chosen according to the number of processors available, but at least four or more processors should be used if the entire sphere needs to be triangulated.

For the set of points $\mathcal{S}$ on the surface of the sphere, we say that G_i is a subset of points in $\mathcal{S}$, such that every point $g \in G_i$ has its longitudinal coordinate $g_\phi \in [\phi_0^{G_i}, \phi_{max}^{G_i}]$. Here ϕ has the same definition as that given by spherical coordinates.

How to divide the sphere. Let P be the number of processors available and p be the current processor. Then we define global variables Φ_0, Φ_{max}, Θ_0 and Θ_{max} that delimit the section of the sphere that will be triangulated. Note that all of these can be set so that the entire sphere is triangulated. This allows us to compute the global value $\Delta\Phi$ using Eq. 5.1.

$$\Delta\Phi = \frac{\Phi_{max} - \Phi_0}{P} \tag{5.1}$$

Now, we define the local variables $\phi_{0,p}$ and $\phi_{max,p}$ as determined by Eqs. 5.2 and 5.3, that will be the delimiters that each processor p will use to determine its own subset of points.

$$\phi_{0,p} = p\ \Delta\Phi \tag{5.2}$$
$$\phi_{max,p} = (p+1)\ \Delta\Phi \tag{5.3}$$

where $p = 0, 1, 2, ..., P-1$ for P available processors.

In order to find the points of $\mathcal{S}$ that will be in each G_p for $p = 1, ..., N$, each processor picks out the points that lie in its region from the entire point set $\mathcal{S}$. Since this action can be completed simultaneously and independently by each processor, and there is no communication involved; we believe that this is an advantageous approach, specially for the case when a dynamic SMP server can be used.

5.2.2 Point Generation

Of course, before we can begin to construct a Delaunay triangulation, we need to be able to generate points on the surface of the sphere. There are several probabilistic methods to do so that incorporate a wide range of distributions, however, for now we will stick to a deterministic approach by creating points at fixed intervals in both latitude and longitude.

Creating points on the sphere. In our implementation we generate points on the sphere by specifying the number of points N_ϕ per latitudinal circle, and the number of N_θ of latitudinal circles. Then, the total number of points $N = N_\phi N_\theta + 1$ when $\Theta_{max} - \Theta_0 < \pi$, or $N = N_\phi N_\theta + 2$ when $\Theta_{max} - \Theta_0 = \pi$.

5.2.3 Building an SDT

For the construction of the SDT we will adapt Lawson's algorithm (see Alg. 3.2) so that it can be applicable to the sphere. As we saw in Chapter 3, the algorithm starts with an "infinite" simplex s that contains all the points p in the set $\mathcal{S}$. In the plane, this simplex is represented by a triangle big enough to contain all the points in its interior. An analogous idea can be used in the sphere by using a spherical triangle

big enough to cover the section of the sphere that we want to triangulate (so we will refer to it as "huge" instead of "infinite"). The algorithm is summarized in Alg. 5.1.

Algorithm 5.1 SDT algorithm

Create the "huge" spherical triangle.
for every $p \in \mathcal{S}$ **do**
  Locate the triangle $\triangle abc \in T_D$ that contains p
  Break $\triangle abc$ by creating edges from p to each a, b and c
  Verify Delaunayhood of each $\triangle pab, \triangle pbc, \triangle pca$
end for
Remove the "huge" spherical triangle.

As in Lawson's algorithm, we will use an edge flipping technique to verify the Delaunayhood of each subtriangle, the method used for verification is outlined in Alg. 5.2, where $d(\cdot, \cdot)$ is the arc length as defined in Chapter 2 and $\triangle_{opp}$ represents the triangle that is adjacent to $\triangle_p$ and it is directly opposite to vertex p.

Algorithm 5.2 $checkEdge(p, \triangle_p, \triangle_{opp})$

Compute the circumcenter c_{opp} and circumradius r_{opp} of $\triangle_{opp}$
if $d(p, c_{opp}) < r_{opp}$ **then**
  $flip(\triangle_p, \triangle next)$, forming $\triangle_1, \triangle_2$
  $checkEdge(p, \triangle_1, \triangle_{opp_1})$
  $checkEdge(p, \triangle_2, \triangle_{opp_2})$
end if

Even though our function is recursive it is guaranteed to stop because the algorithm assumes that in the previous iteration T_D is globally Delaunay and the insertion of p only affects Delaunayhood in the neighborhood of p. Thus, the only triangles that can become non-Delaunay are those adjacent to the triangle containing p.

5.2.4 Point Location

The insertion algorithm outlined has one obvious bottleneck, how do we locate the triangle in T_D that contains the point p? Using a brute force approach is very straightforward, all we need to do is check all the existing triangles until we find one that contains p. Obviously, this method is very inefficient, for that reason, we decided to use a history tree, also known in the literature as a *Delaunay Tree* [9], to store the changes in the triangulation with the insertion of every point. A Delaunay tree allows us to locate p in the triangulation much faster because it reduces the search space from all triangles in the current triangulation to only the number of triangles that are ancestors of the triangle containing p. This technique can be thought of as the 3-ary analogue of a Binary Search Tree.

Delaunay tree. Since our algorithm starts with one "huge" spherical triangle and we know that all points inserted will lie in its interior, we will take our "huge" triangle as the root of our history tree. In our algorithm a triangle can only be broken into three subtriangles that we will refer to as child triangles. Therefore, every node in our tree will have three children. However, because edge flips can occur, we will also allow nodes to have two children and an empty branch, to represent the case when two new triangles are formed by flipping their common edge. In such case, both old triangles will become parents of the two new triangles.

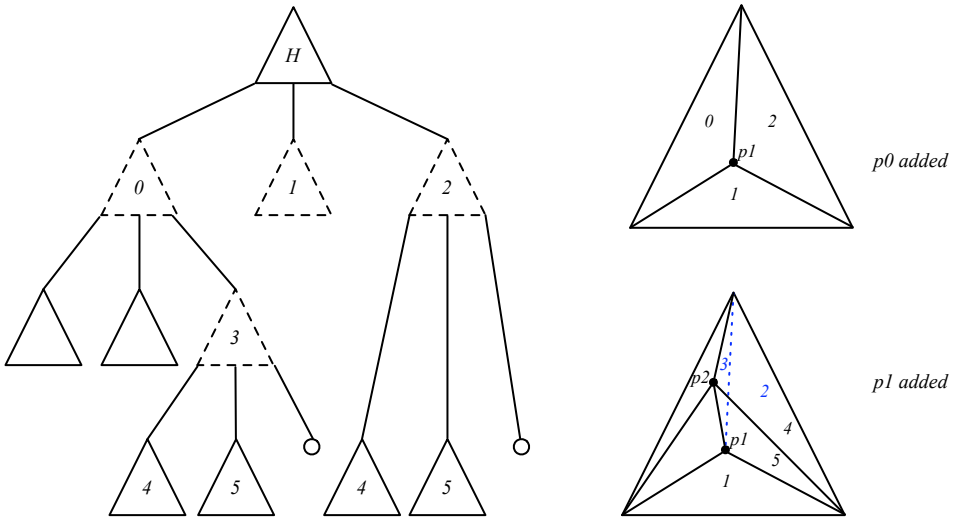

Figure 5.2: Delaunay Tree after insertion of two points resulting in an edge flip.

To help clarify how our structure is constructed, let us go over a simple example illustrated in Fig. 5.2, in which two points p_1 and p_2 have been inserted. In the first step, the root triangle H is divided in three child triangles 0, 1 and 2. Then, p_2 is inserted and it was found to lie inside triangle 0, which results in 0 being broken into three triangles. Now, assume that $\triangle_0$ and $\triangle_2$ are neighboring triangles and that the insertion of p_2 resulted in an edge flip between $\triangle_3$ and $\triangle_2$, then, the resulting new triangles 4 and 5 are assigned as children of both $\triangle_3$ and $\triangle_2$.

5.2.5 Storing the Triangulation

Since the approach we are using is incremental, we will use an array to store all the triangles in the Delaunay tree. To represent each individual simplex we designed a triangle data structure that we named `Tri` that stores information about the vertices forming the simplex as well as pointers to its three (or less) adjacent simplices. As

a convention[1], we define a simplex by its three vertices (in counter-clockwise order) p_0, p_1, p_2 with adjacent neighbors $\triangle_{p_0}, \triangle_{p_1}, \triangle_{p_2}$, where $\triangle_{p_i}$ represents the adjacent simplex that is directly opposite to p_i. In Fig. 5.3 we illustrate the data structure and how neighboring relationships are assigned.

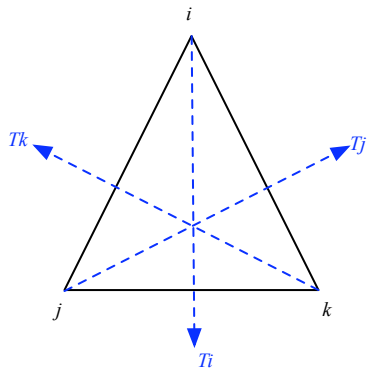

Figure 5.3: `Tri` data structure

5.2.6 Merging Neighboring Triangulations

In our parallel approach we are dividing the sphere into vertical *stripes* and assigning each one to a different processing element. Once the individual triangulations are completed they need to be merged together to form the triangulation for the entire sphere. To accomplish that we will use a technique similar to that presented by Guibas & Stolfi [20], in which neighboring triangulations are stitched together using an incremental construction algorithm. The basic idea is to create a Delaunay triangulation to fill the gap between two contiguous *stripes*. To do so, let us consider two contiguous *stripes*: G_1 and G_2, and let L be the subset of points that are located in the right boundary of G_1, and R be the subset of points located in the left boundary of G_2. Then, we can build an SDT between L and R by using Alg. 5.3, that will guarantee the Delaunayhood of the triangulation formed between L and R.

5.2.7 Additional Considerations

Now that the we have presented how a local triangulation can be built for a section of the sphere, we want to take the specific case that will apply to one of our *stripes*. As mentioned earlier, Lawson's algorithm starts with one "infinite" simplex that contains all the points. For our *stripe* however, one simplex is not enough, so we chose to initialize our algorithm with two adjacent "huge" simplices, this way,

[1]This convention has been previously used for Delaunay Triangulations by different authors, including R. Renka

Algorithm 5.3 *stitch*(L, R)

Let $L = \{l_k\}_{k=1}^{N_L}$, and $R = \{r_k\}_{k=1}^{N_R}$
$l_{base} := l_1$, and $r_{base} := r_1$
while $i < N_L$ and $j < N_R$ **do**
  $p_l := l_i$
  $p_r := r_j$
  $C :=$ circumcircle of $\triangle(l_{base}, r_{base}, p_l)$
  if $p_r \in C$ **then**
    Add triangle $\triangle(l_{base}, r_{base}, p_r)$
    $r_{base} := p_r$
    $j = j + 1$
  else
    Add triangle $\triangle(l_{base}, r_{base}, p_l)$
    $l_{base} := p_l$
    $i = i + 1$
  end if
end while
if $i < N_L$ **then**
  while $i < N_L$ **do**
    $p_l := l_i$
    Add triangle $\triangle(l_{base}, r_{base}, p_l)$
    $l_{base} := p_l$
    $i := i + 1$
  end while
else
  if $j < N_R$ **then**
    while $j < N_R$ **do**
      $p_r := r_j$
      Add triangle $\triangle(l_{base}, r_{base}, p_r)$
      $r_{base} := p_r$
      $j := j + 1$
    end while
  end if
end if

a point can be found inside the initial triangulation regardless of the hemisphere where it is located. Here we can take advantage of the `Tri` data structure by setting the top and bottom "huge" simplices as neighbors of each other. As a consequence the algorithm is able to maintain Delaunayhood across the boundary between the north and south hemispheres. Without establishing this adjacency relationship for our starting triangulation, we would be forced to apply the stitching algorithm at the boundary between the top and bottom simplices to complete the local triangulation. The former was chosen because it was a much simpler approach and preserved the neighboring of all the triangles inside the "stripe", regardless of the hemisphere they are located.

5.3 Establishing Communication

Since we chose to divide the sphere only in one direction, we can think of the topology for our network of processors as a ring (which won't necessarily match the physical topology of the network). The algorithm has enough parallelism that each processing element does need to communicate with its neighbors during the construction of its local triangulation. It is only at the final phase (the merging stage) that communication between contiguous processors will occur.

By design, each processor p_i will receive boundary data from its left neighbor p_{i-1} and fill the gap between the two triangulations. In other words, p_1 is responsible for merging the right boundary of p_0 with its left boundary, p_2 for merging the right boundary of p_1 to its left boundary, and so on until we get to p_0 that is responsible for filling the triangulation between p_P and its own triangulation. When all the local merges are completed, the root processing element gathers the triangles from all other processors to obtain the global triangulation.

CHAPTER 6

PERFORMANCE ANALYSIS

Now that we have described the design chosen for our software package in detail, in this chapter we proceed to analyze its performance and behavior. We will take time measurements for two distinct cases: increasing number of points and increasing number of processors. For the first case, we will use a constant number of processors and increase the number of generating points. Whereas, for the latter case, we will maintain the number of points constant and increase the number of processors that construct the SDT.

To benchmark our code we will use the timing functions included with the MPI Interface to measure how much time does the SDT construction take. To get better granularity we will create enough timers that will allow us to measure the amount of time the code spends in each of the *triangulation* and *merge* phases. In order to determine whether our parallel approach has a better performance than Robert Renka's code, we will also time the construction of SDTs using STRIPACK. In this case, since STRIPACK is a serial FORTRAN 90 code, we will use for our benchmarking the timer modules provided during our Scientific Computing graduate course[1].

In the following sections we present the SDTs constructed using our code in conjunction with the results obtained from our benchmarks.

6.1 Spherical Delaunay Triangulations

We ran our code to generate SDTs with inputs ranging from a few tens of points to hundreds of thousands of points. Here, we present a small sample of the SDTs obtained for 10000 points using 8 processors (Fig. 6.1), for 10000 points using 16 processors (Fig. 6.2) and for 20000 points using 32 processors (Fig. 6.3). All the cases shown use as generators, points distributed on the sphere at constant latitude and constant longitude. The points were generated using the method described in Section 5.2.2.

[1]The FORTRAN 90 timer module was developed by Dr. Erlebacher who provided the code during his Scientific Programming class

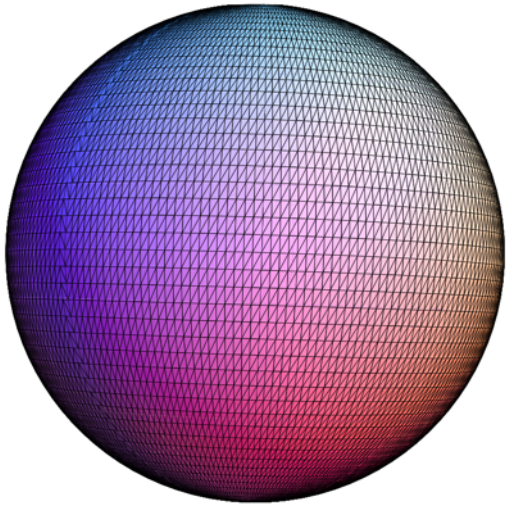

Figure 6.1: 10000 point-SDT constructed using 8 processors

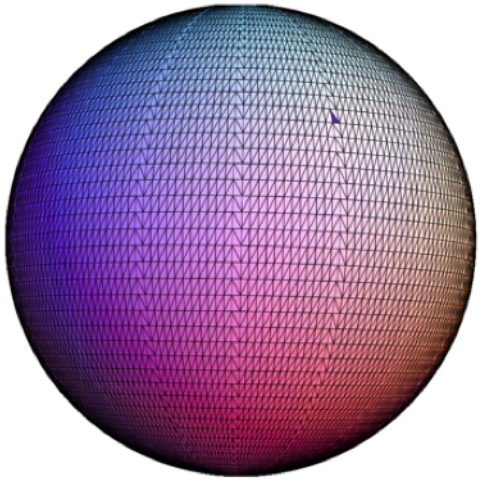

Figure 6.2: 10000 point-SDT constructed using 16 processors

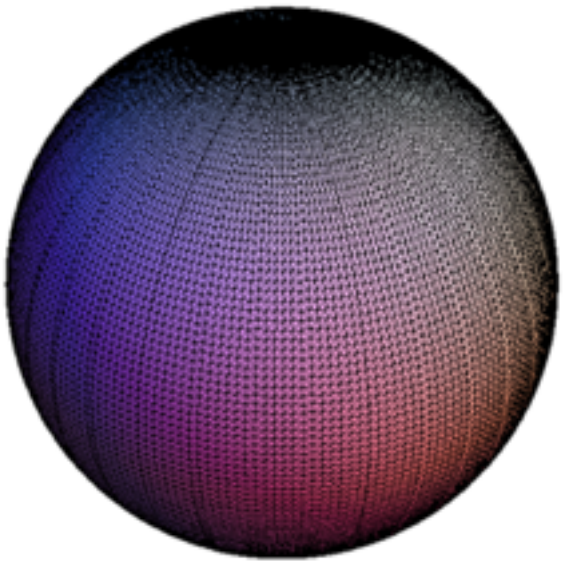

Figure 6.3: 20000 point-SDT constructed using 32 processors

6.2 SDT Construction Time

In Fig. 6.4 we show the benchmarks obtained with 4, 8, 16, 32 and 64 processors for generating point sets of different size and compare them with the total time STRIPACK takes to generate SDT of the same size. As we can see, for smaller point sets, STRIPACK gives a faster construction time than our parallel code for the number of processor tested. However, as we increase the size of the point set, our code begins to outperform STRIPACK. When we use a larger number of processors, we are able to obtain better performance much quicker, i.e. for smaller point sets.

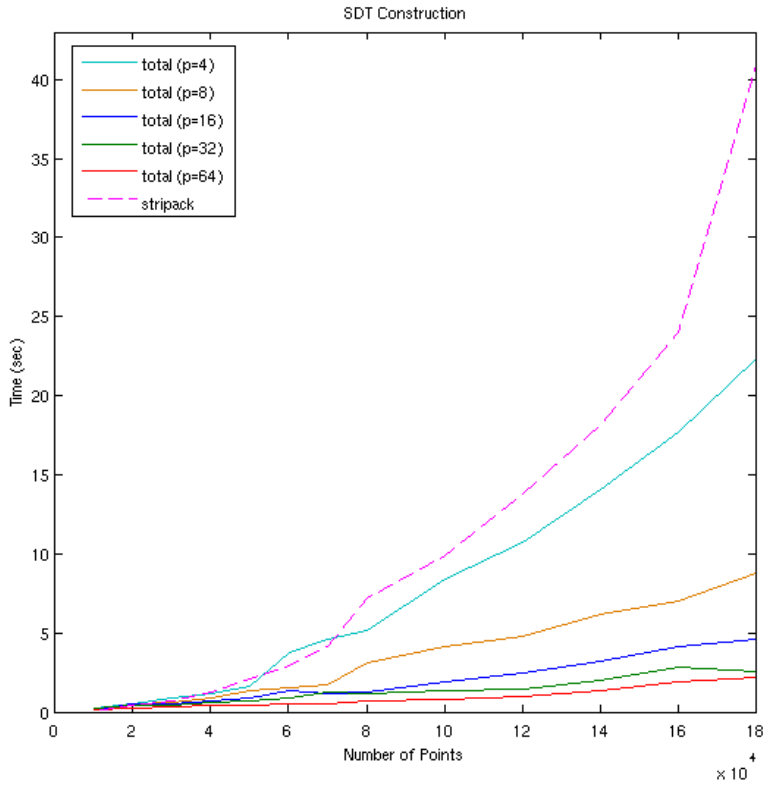

Figure 6.4: SDT construction time compared with STRIPACK time.

We decided to create two different timers, one that measures the time it takes for all processors to complete their portion of the triangulation and another to calculate the time the stitching process takes. The latter, includes the time it takes for the root processor to collect the triangulation from each of the processors. In Fig. 6.5 we show the benchmarks for only the first timer, i.e. exclusively the timings for the triangulation.

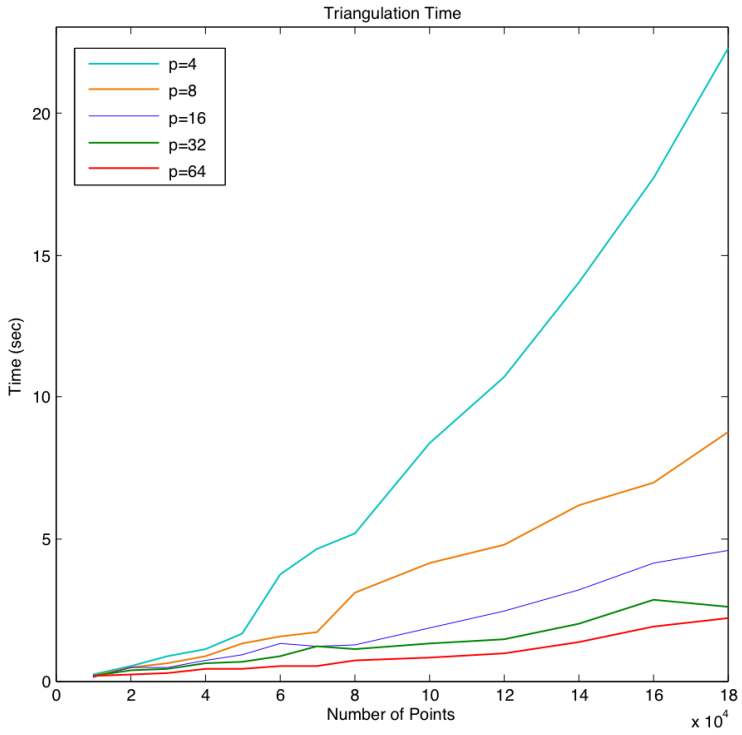

Figure 6.5: Triangulation time using $P = 4, 8, 16, 32, 64$.

To better appreciate the contribution of the merging phase to the construction, in Fig. 6.6 we show the triangulation only time and total time to complete the triangulation.

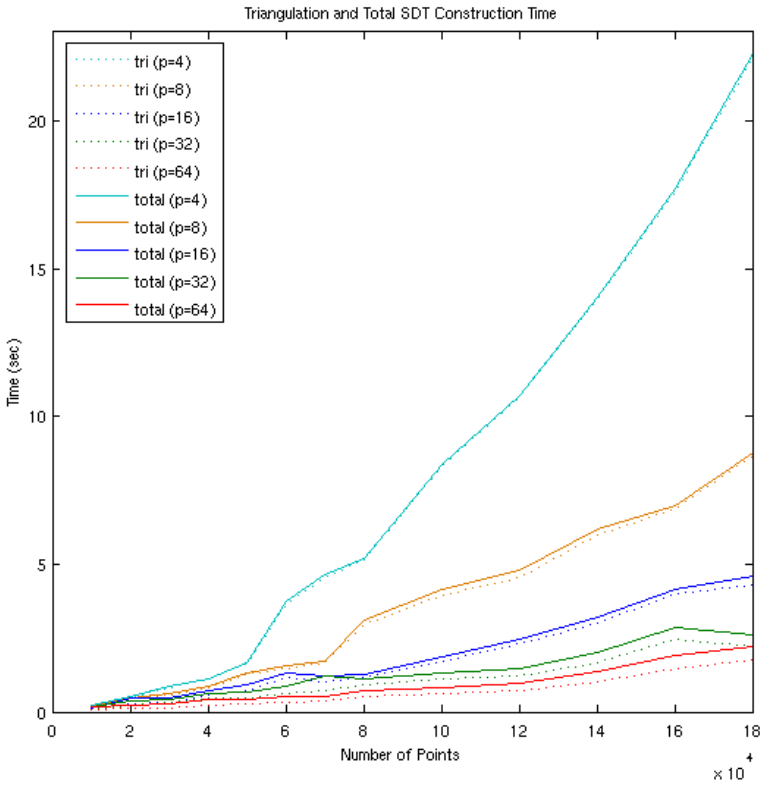

Figure 6.6: Triangulation time compared to total construction time.

As we can see, the merging phase represents only a small percentage of the total construction time and, as we expected, it increases with the total number of processors and with the size of the generating point set. We also note that when we use 32 processors the merging time significantly contributes to the total construction time. One possible explanation is that for the point distribution we are using 32 divisions can potentially increase the time it takes to identify the boundary points.

6.3 Speedup and Efficiency

To illustrate our code's performance we plotted the timings for constant number of points, see Fig. 6.7. The plots shows how the total construction time rapidly reduces when we increase the number of processors for a given triangulation. One observation

we can make here is that the speedup varies, only in some cases we obtain close to or better than linear speedup.

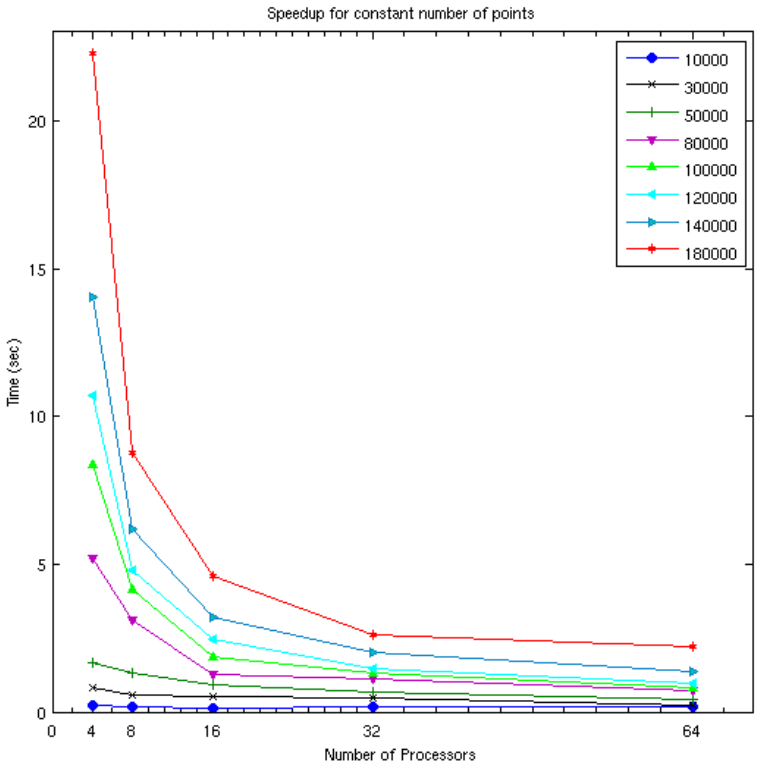

Figure 6.7: Triangulation time for constant number of points.

In Fig. 6.8, we compare the total construction times for combinations of point sets and number of processors that result in constant work done by each processor. Ideally, we would want each curve in this graph to be parallel to the x-axis, since the triangulation at each processor should take the same amount of time. Our plot suggests, that our code deviates somewhat from this expected behavior. After careful consideration, we believe the loss of efficiency appears in the point location scheme, when points that lie very close to the boundary of a triangle in the Delaunay tree trigger the search of extra branches.

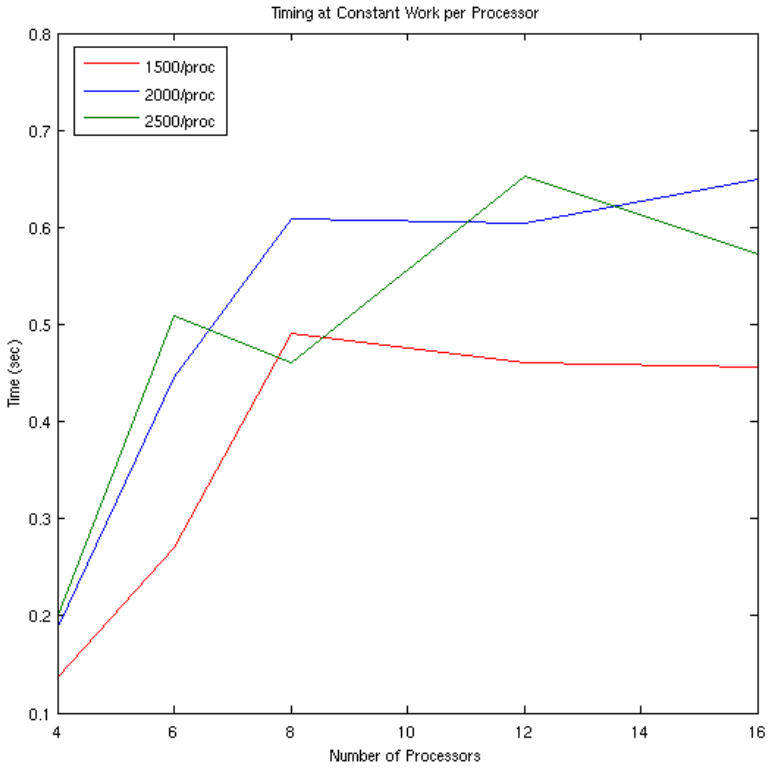

Figure 6.8: Triangulation time for constant work per processor.

CHAPTER 7

CONCLUSION

In this document, we first defined Delaunay Triangulations in n dimensions, the a generalization that we later constrained to present a definition for the main subject of this thesis: Spherical Delaunay Triangulations. To further support our motivation to study SDTs, we briefly talked about their importance in a variety of fields. We also described the relationship between Delaunay Triangulations and their dual Voronoi Tessellations that allowed us to reiterate the importance of generating more efficient algorithms for their construction.

Later, we briefly discussed common methods for the construction of Delaunay Triangulations and outlined currently available software packages. One of them, STRIPACK, was described in more detail given that it played a very important role in the conception of this project and the design of our parallel algorithm.

Motivated by our interest in high performance computing, we provided a concise introduction that recounted the history of supercomputers, described the different programming and memory models available, and overviewed two of the standard techniques used to develop parallel software, namely, MPI and OpenMP.

Next, we gave a detailed description of the design and the implementation of our parallel SDT generation algorithm. And finally, in the previous chapter, we used our code to generate SDTs from point sets of different sizes and obtained benchmarks in order to quantify its performance. Our analysis led us to conclude that there is an obvious benefit of using our parallel approach over the serial alternative.

7.1 Future Work

While our results indicate that our parallel software package can significantly speed up the construction of SDTs, we believe more research is necessary in order to obtain maximal efficiency. After reviewing the different techniques used in the field to build Delaunay Triangulations, we realized that there are plenty techniques to explore, especially because parallel algorithms for their construction are scarce.

In this project, we chose one approach that to our knowledge has not been previously used for the construction of Delaunay Triangulations on the Sphere. Our study

taught us that one of the most time consuming parts of our code is the point location with respect to the current triangulation. A valuable contribution would be to explore alternate searching techniques on the Delaunay tree, for instance, Delaunay tree sampling.

During the merging phase, our algorithm stitches two neighboring triangulations, however, there are several other techniques we have seen in the literature including the allowance of an overlap between sections of the given surface. One consequence of this approach is that a post-processing phase needs to be added to eliminate all the duplicated triangles near section boundaries.

Another aspect worth exploring is to observe the change in behavior for different point distributions. Here, we considered the simplest case, i.e. when points are distributed uniformly over the sphere, but it would be interesting to extend this research to encompass other kinds of point distribution.

Finally, a possible production alternative would be to utilize division and merging methods, similar to those presented in this thesis, and to use STRIPACK itself as the triangulation algorithm. We have shown in this document that STRIPACK performs really well when the point set is small, which suggests that as long as the number of divisions is large and enough processors are available, STRIPACK can still show us an advantage.

BIBLIOGRAPHY

[1] Cgal, http://www.cgal.org/.

[2] http://www.blackpawn.com/texts/pointinpoly/default.html.

[3] http://www.ics.uci.edu/~eppstein/gina/delaunay.html.

[4] http://www.phy.ornl.gov/csep/ca/node10.html.

[5] http://www.softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm.

[6] Gabrielle Jost Barbara Chapman and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008.

[7] C. Barber. qhull, http://www.qhull.org/.

[8] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, 22(4):469–483, 1996.

[9] J D Boissonnat and M Teillaud. The hierarchical representation of objects: the delaunay tree. In *Proceedings of the second annual symposium on Computational geometry*, SCG '86, pages 260–268, New York, NY, USA, 1986. ACM.

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

[11] Kevin Q. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9(5):223 – 228, 1979.

[12] L. P. Chew. Constrained delaunay triangulations. In *Proceedings of the third annual symposium on Computational geometry*, SCG '87, pages 215–222, New York, NY, USA, 1987. ACM.

[13] Paul Chew. http://www.cs.cornell.edu/home/chew/oldDelaunay.html, 2005.

[14] Mark de Berg. http://www.cs.uu.nl/geobook/.

[15] Boris N. Delaunay. Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR*, (6):793–800, 1934.

[16] Rex A. Dwyer. Higher-dimensional voronoi diagrams in linear expected time. *Discrete Comput. Geom.*, 6:343–367, May 1991.

[17] Herbert Edelsbrunner and Raimund Seidel. Voronoi diagrams and arrangements. *Discrete and Computational Geometry*, 1:25–44, 1986. 10.1007/BF02187681.

[18] M. J. Flynn. Very high-speed computing systems. *Proceedings IEEE*, pages 1901–1909, Dec. 1966.

[19] Steven Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2:153–174, 1987. 10.1007/BF01840357.

[20] Leo J. Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 221–234, New York, NY, USA, 1983. ACM.

[21] C L Lawson. *Software for C^1 surface interpolation*, pages 161–194. Academic Press, 1977.

[22] C L Lawson. C^1 surface interpolation for scattered data on the surface of a sphere. 14:177–202, 1984.

[23] Dave Mount. `www.cs.umd.edu/class/spring2010/cmsc754/Lects/lect14.pdf`, 2010.

[24] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., 1997.

[25] R. J. Renka. Algorithm 751: Tripack: a constrained two-dimensional delaunay triangulation package. *ACM Trans. Math. Softw.*, 22:1–8, March 1996.

[26] Robert J Renka. Algorithm 772: Stripack: Delaunay triangulation and voronoi diagram on the surface of a sphere. *ACM Transactions on Mathematical Software*, 23(3):416–434, 1997.

[27] Jonathan Shewchuk. Triangle, `http://www.cs.cmu.edu/~quake/triangle.html`.

[28] Peter Su and Robert L. Scot Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, pages 61–70, New York, NY, USA, 1995. ACM.

[29] Masaharu Tanemura, Tohru Ogawa, and Naofumi Ogita. A new algorithm for three-dimensional voronoi tessellation. *Journal of Computational Physics*, 51(2):191 – 207, 1983.

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

[31] Geoffrey Womeldorff. Spherical Centroidal Voronoi Tessellations: Point generation and density functions via images. Master's thesis, Florida State University, School of Computational Science, June 2008.

[32] Henrik Zimmer. Voronoi and delaunay techniques. July 2005.

BIOGRAPHICAL SKETCH

Verónica G. Vergara Larrea is originally from *La Mitad del Mundo*: Quito, Ecuador, but has been living in the United States for about six years now. When she began her college career back in her hometown she knew she was interested in Computer Science but her desire to explore the world and learn about other cultures led her to a small liberal arts school in Portland, OR, where she continued her studies and majored in Mathematics and Physics.

After obtaining her Bachelor's degree, Verónica decided to enter the IT world and became a Software Quality Assurance Analyst, which she soon discovered was not the best fit for her as it did not give her enough room to explore new areas. Eager to face a new challenge, she chose to apply for a graduate degree and the uniqueness of the Department of Scientific Computing caught her eye. In 2009, she started her graduate studies at Florida State University and a year later began working as a research assistant under the supervision of Dr. Max Gunzburger.

In the near future, Verónica will join the Rosen Center of Advanced Computing at Purdue University to work as a Scientific Application Analyst and collaborate with scientists across all departments to promote the use of high performance computing as a way to accelerate their research.