Universität Heidelberg
Interdisciplinary Center for Scientific Computing
Forensic Computational Geometry Laboratory

Master's Thesis

GPGPU Accelerated Iterative Filtering of Scalar Fields on Discrete Manifolds

Name:	Bryan Wolfford
Matrikelnummer:	3346612
Betreuer:	Hubert Mara and Susanne Krömker
Datum der Abgabe:	March 3, 2019

I affirm that I have written this Master's Thesis independently and with those sources and aids as specificed in accordance with the principles and recommendations of the "Verantwortung in der Wissenschaft" of the University of Heidelberg.

Abgabedatum: March 3, 2019

Abstract

Motivated by the increasing importance of 3D-data acquisition, and the difficulty inherent to filtering noise from data, given the current techniques, this thesis presents the Fast One-Ring smoothing filter for scalar fields on discrete manifolds as a scalable solution to smoothing large meshes of 3D-data, by utilizing commercially-available, General Purpose, Graphics Processing Units (GPGPUs). The filter convolves over each one-ring neighborhood in the mesh, and calculates the weighted mean function values of those neighborhood, by interpolating the function values toward the center point. Originally, the filter was implemented as a strictly-serial procedure, but this thesis presents an enhanced parallel variant of the algorithm, made possible by the introduction of GPGPUs. When tested in a battery of experiments including synthetic and acquired 3D-data, while the efficiency remain at 15% or below, the parallel variant achieved an increase in performance of upto 204%.

Aknowledgements

I wish the warmest regards for everyone who has supported me during my time here, especially for my wife, Kirsten, for all the sacrifices she made for the benefit of this research, and my son, Dexter, who is perpetually a ray of sunshine, even on the grayest of days.

Nomenclature

These symbols are used in equations, text, and figures throughout this paper. They are sorted by the order in which they are introduced.

	Geodesic Discs
Ω	a geodesic disc, see equation (2.9), page 9
Ω_v	a specific geodesic disc centered about the point $\mathbf{p}_v$, see equation (2.9), page 9
	Circle Sectors
$\mathbf{s}$	a circle sector, see equation (2.9), page 9
$\mathbf{s}_i$	a specific circle sector of geodesic disc Ω_v, see equation (2.9), page 9
A	an area of a circular sector, see equation (2.10), page 9
	Center of Gravity
$\mathbf{c}$	the center of gravity of circle sector $\mathbf{s}_i$, see equation (2.11), page 10
$\check{\ell}$	the distance from $\mathbf{p}_0$ to $\mathbf{c}$, see equation (2.11), page 10
	Discrete Meshes
$\mathcal{M}$	a mesh; the set including the sets of all points $\mathbf{p}$ and faces $\mathbf{t}$, see equation (2.21), page 16
	Points
$\mathcal{P}$	the set of points $\mathbf{p}_v$ in $\mathcal{M}$, see equation (2.15), page 14
$\mathbf{p}_v$	a specific point in $\mathcal{P}$, see equation (2.15), page 14
$\mathbf{p}_i$	also $\mathbf{p}_{i+1}$, and $\mathbf{p}_{i+2}$; one of three indirectly referenced points comprising a face, see equation (2.15), page 14
	Faces
$\mathcal{T}$	the set of faces $\mathbf{t}_k$ in $\mathcal{M}$, see equation (2.17), page 15
$\mathbf{t}_k$	a specific face in $\mathcal{T}$, see equation (2.17), page 15
	Edge Lengths
ℓ_i	the length of the edge opposite the point $\mathbf{p}_i$ of a specific face, see equation (2.18), page 15
$\ell_{v,i}$	the length of the edge between points $\mathbf{p}_v$ and $\mathbf{p}_i$, see equation (2.19), page 15

	One-Ring Neighborhoods
$\mathcal{N}$	a family of sets of neighbors, the set of neighborhoods, see equation (2.23), page 16
$\mathcal{N}_v$	the set of points comprising the one-ring neighborhood about $\mathcal{P}_v$, see equation (2.22), page 16
$\mathbf{p}_i$	a one-ring neighbor of $\mathbf{p}_v$, see equation (2.22), page 16
	Function Values
$\mathcal{F}$	the set of function values f_v; a scalar field, see equation (2.24), page 18
f_v	a specific function value in $\mathcal{F}$, corresponding to $\mathbf{p}_v$, see equation (2.24), page 18
	The Shortest Edge Length
$\mathbf{p}_0$	the center point of $\mathcal{N}_v$, see equation (3.1), page 29
$\ell_{\min}(\mathbf{p}_v)$	the shortest edge length in $\mathcal{N}_v$, see equation (3.1), page 29
$\overline{\ell_{\min}}$	the shortest edge length in $\mathcal{M}$, see equation (3.2), page 29
	Interior Angles
α	the central angle of circle sector $\mathbf{s}_i$, see equation (3.4), page 30
β	the third angle with $\frac{\alpha}{2}$ and $\frac{\pi}{2}$, see equation (3.5), page 30
$\mathbf{p}'_j$	also $\mathbf{p}'_{j+1}$, the corners of the isosceles triangle circumscribing the circle sector, the intermediate location to where f_j is first interpolated or extrapolated, see equation (3.5), page 30
	Interpolation & Extrapolation
ζ	sector-wise constant ratio for interpolation derived from the law of sines, see equation (3.8), page 32
$\tilde{\ell}_j$	also $\tilde{\ell}_{j+1}$, the distances for interpolation of f_j and of f_{j+1} towards f_0, see equation (3.10), page 32
f'_j	also f'_{j+1}, the interpolated values of f_j and f_{j+1} toward f_0, see equation (3.12), page 32
	Weighted Means
$\check{f}$	the weighted mean function value at $\mathbf{c}$ of $\mathbf{s}_i$, see equation (3.13), page 33
f'_v	the one-ring weighted mean function value at $\mathbf{p}_v$, see equation (3.14), page 34
$\mathcal{F}'$	a scalar field, a set of weighted mean function values, the product of convolving the Fast One-Ring smoothing filter, see equation (3.15), page 35
	Building Neighborhoods
τ	the user-defined number of convolutions to perform, see equation (4.0), page 37
	Calculating Edge Lengths
ℓ_*	the procedure of calculating an edge's length using "Newton's Iteration", the most costly operation in the Fast One-Ring filter, due to use of $\sqrt{(\cdot)}$, see equation (4.0), page 39
$\mathcal{E}$	a set of pre-calculated edge lengths, see equation (4.0), page 39
	Convolving the Filter

$\tilde{f}$ the total volume of function values over Ω_v, see equation (4.0), page 40

$\tilde{A}$ the area of Ω_v, see equation (4.0), page 40

Parallel Variant of Build Neighborhoods

ρ the number of available processors, see equation (5.0), page 45

σ the "stride", the size of a block of work intended for a single processor, equal to the problem size divided by the number of available processors, see equation (5.0), page 45

$\check{\sigma}$ the index of the beginning of a stride, see equation (5.0), page 45

$\hat{\sigma}$ the index of the beginning of a stride, see equation (5.0), page 45

$\mathfrak{M}$ a set of mutexes, see equation (5.0), page 45

μ a specific mutex, see equation (5.0), page 45

Π the index representing a single processor, see equation (5.0), page 45

$\sim process$ a process to be run in parallel by a new thread, see equation (5.0), page 45

Parallel Recursive Census Neighborhoods

$\hat{n}$ census total of all neighbors in all neighborhoods in $\mathcal{N}$, see equation (5.1), page 46

$\widetilde{\mathcal{N}}$ a subset of $\mathcal{N}$, see equation (5.1), page 47

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction & Motivation

Motivated by the increasing importance of 3D-data acquisition for both industry and academia, and the inherent difficulties associated with processing that data in an efficient way, this thesis presents the parallel variant of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds as a meaningful solution.

The work presented in this thesis was conducted with support from the Heidelberg University Excellence Initiative through the DFG, as well as the Forensic Computational Geometry Laboratory (FCGL) in cooperation with the Graduate School for Mathematical and Computational Modeling (HGS MathComp) at the Interdisciplinary Center for Scientific Computing (IWR). The algorithms presented may be implemented directly as source code, as was already done in CUDA enhanced C++, in order to generate the synthetic 3D-data and conduct the experiments used for analysis. The serial algorithm is also readily available as part of the GigaMesh framework, which is maintained by the FCGL.

1.1. Motivation

From industrial quality scanning [14], to computerized analysis of documents and artifacts within the Digital Humanities [15], the demand for high-definition 3D-data is only increasing. And in light of recent tragedies befalling global physical archives, like the museum fire which destroyed Brazil's oldest museum and its 20 million artifacts [11], or the numerous war zones occupying archaeologically important sites, 3D-scanning will continue to increase in importance.

However, raw 3D-data is typically not suitable for analysis [26, p. 25-32], so processing and pre-processing with a smoothing filter becomes necessary. In dense, high-resolution meshes, for example, one can see noise propagating as jagged outlines in segment boundaries of connected components when visualizing the output of filters designed for analysis, like the Multi-Scale Integral Invariants filter (MSII) [27, s. 3.2].

One major complication slowing the development of filters for 3D-data has been that the window size of any filter must remain static for the duration of its convolution, in order for the output response to be mapped correctly back onto the input field [21, p. 106-112]. While it is trivial to define a static-sized filter for convolving regular

meshes like raster images, it is a complex and complicated task to create the same for convolving acquired 3D-data, whose one-ring neighborhoods are subsets of non-planar meshes embedded in $\mathbb{R}^3$, uniformly irregular, with completely arbitrary shapes, sizes, and counts of members [26, p. 29] [27, s. 3.2].

Another problem which arises, is that with high-resolution 3D-scanning, the data output is often comprised of millions, or tens of millions, of points per scanned item and features several hundred points per mm^2 [27, 25,144] [14, 4]. At that scale, serial algorithms processing that data can no longer be included in the regular workflow of a scientist analyzing the artifacts, because each operation can easily take hours or days to complete.

Fortunately, with the introduction of General Purpose, Graphics Processing Units (GPGPUs) to the commercial market, individual research groups now have the opportunity to exploit the parallel processing power of Single Instruction, Multiple Data (SIMD) systems, without needing access to an institutional supercomputer. Therefore, the motivation for designing a smoothing filter, which can efficiently convolve over large, irregular, acquired 3D-data by utilizing commercially available GPGPUs, was realized. Thus came the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, and the research presented in this thesis.

1.2. Related Work

As filtering noise from acquired 3D-data is a motivating topic, the adaptation of filters already established for regular, two-dimensional meshes to be used with irregular, non-planar, acquired 3D-data is a topic of current research. One such filter for de-noising meshes while preserving sharp-edges is the "The Bilateral Filter for Point Clouds" [18], which presents an adaptation of the bilateral filter two-dimensional images for use with point clouds and parallel processing.

Because it is possible to adapt a filter, that was designed for two-dimensional data, to be convolved on 3D-data, the entire field of digital image processing presents itself an opportunity for related work. For example the filters for feature extraction include smoothing [21, p. 299], edge [21, p. 331] and motion detection [21, p. 397], as well as more complicated topics, such as three-dimensional face recognition from two-dimensional images [1].

As a consequence of the design of the the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, larger filter responses are only obtained through more convolutions of the filter. Additionally, as smoothing filters are also characterized as a diffusion-reaction system [21, p. 474], the on-going research on the topic of numerical optimization of such systems, especially in regards to the stability of algorithms remains relevant.

In our research, the parallel variant of the Fast One-Ring smoothing filter was implemented using the proprietary CUDA framework [**?**], however, there exists another framework which enables the use of non-NVIDIA produced GPGPUs for parallel processing. It is called OpenCL and it is developed by The Khronos Group [2] to enable general processing on most Graphics Processing Units (GPUs), regardless of manufacturer.

Other frameworks for developing software for parallel processing, which are not used

in this thesis include: OpenMP [3] which enables the use of distributed computation across multiple machines and networks, and POSIX Threads, or pthreads [25, p. 195-210], for use in the limited SIMD environment, made available by all modern day Central Processing Units (CPUs).

1.3. Structure this Thesis

This document is structured into the following chapters: Chapter 2 briefly covers many theories, concepts, and frameworks across multiple fields of study, in order to focus on a few specific topics which have a direct influence on research presented in this thesis. These include topics from set theory, linear algebra, geometry, and topology. The Fast One-Ring smoothing filter for scalar fields on discrete manifolds is presented in Chapters 3, 4, and 5. First, Chapter 3 introduces the mathematical foundations upon which the filter was designed. Next, Chapter 4 defines the serial algorithm for implementing the filter. Then, Chapter 5 analyzes the serial filter in order to present the parallel variant of the Fast One-Ring smoothing filter algorithm. The example meshes used in experimentation are described in Chapter 6, before both the filter response as well as the performance of the parallel algorithm are then evaluated. Finally, the conclusion for this thesis and an outlook for future enhancements is provided in Chapter 6.

Chapter 2

Background

As computational geometry is founded on many other fields of study in both mathematics and computer science, likewise the Fast One-Ring smoothing filter for scalar fields on discrete manifolds also relies on the same. In this chapter, we briefly cover many theories, concepts, and frameworks across many fields of study, opting for breadth over depth, in order to focus only on the specific topics which have a direct influence on research presented in the rest of this thesis.

Section 2.1 covers the elementary, theoretical basis for the mathematics utilized in the design of the Fast One-Ring smoothing filter, including various topics in set theory, linear algebra, geometry, and topology. Next, Section 2.2 defines several symbols which will be used throughout the rest of this thesis while discussing in relative detail the primitives of 3D-data, as well as the nature of discrete meshes, one-ring neighborhoods, the differences between acquired and synthetic 3D-data, and the treatment of scalar fields as function values. Then in Section 2.3, various topics regarding the design and analysis of algorithms for parallel processing are briefly discussed.

2.1. Elementary Theoretical Basis

This section discusses an abbreviated collection of the elementary, theoretical concepts that comprise the basis upon which the Fast One-Ring smoothing filter was designed. These important concepts and symbols are collected into sections, including limited topics from: set theory in Section 2.1.1, linear algebra in Section 2.1.2, geometry in Section 2.1.3, and topology in Section 2.1.4.

2.1.1. Set Theory

A set is one of the most fundamental concepts in mathematics. Informally, a set is a collection of distinct objects, and is also itself considered to be an object. In this thesis, sets are denoted using capital, calligraphic letters, including: $\mathcal{E}$, $\mathcal{F}$, $\mathcal{M}$, $\mathfrak{M}$, $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{T}$. Each of these will be defined and described in detail when introduced individually, later in this thesis. In this section, we will introduce the concepts and nomenclature used to define a set, reference special sets, determine the cardinality of a set, as well as apply membership, relational, and binary operators on sets.

2.1.1.1. Set Definitions

As several different sets are used throughout the remainder of the thesis, we must first discuss how it is one may define a set and its membership, including "intensional definitions" and "extensional definitions". An intensional definition uses semantic rules and symbols, where each symbol can be translated into words so that the definition may be coherently read aloud. For example,

$$\mathcal{P} := \{\, \mathbf{p}_v \mid v \in \mathbb{N},\ \text{and } 1 \leq v \leq v_{max} \,\} \tag{2.1}$$

which should be read as, "the set $\mathcal{P}$ is defined as the set of all points $\mathbf{p}_v$, such that the index v is a member of the set of natural numbers, and v is a number from 1 to the maximum index of points in the set."

The other option, extensional definition, is denoted by enclosing the list of members in curly brackets and optionally invoking an ellipsis (. . .) for continuing into infinity. For example,

$$\mathbb{N} = \{\, 0,\ 1,\ 2,\ 3,\ \ldots \,\} \tag{2.2}$$

which in words, means "the set of natural numbers is the set which includes every integer from zero[1] until infinity."

It is allowed for one to list a set member two or more times in a definition. For example, $\{\, 4,\ 2,\ 2 \,\}$. However, it is identical to the set $\{\, 4,\ 2 \,\}$ per the axiom of extensionality, which states that two definitions of sets, which differ only in that one of the definitions lists members multiple times, define the same set.

2.1.1.2. Special Sets

There exist some sets which are used in mathematical literature with such frequency as to demand their own standardized symbols. In this thesis, we have already encountered $\mathbb{N}$, as defined in Equation 2.2, which denotes the set of all natural numbers, but more often we will use the set of all real numbers $\mathbb{R}$, which can be described as "the set of all positive or negative, rational or irrational numbers, including zero". In short, any number which is not imaginary. This special set, is also usually written with a superscript denoting a specific dimensionality, as in $\mathbb{R}^3$ for three-dimensional data.

2.1.1.3. Cardinality

Many times throughout this thesis, we reference the cardinality of set, which simply means the total count of its membership, and is denoted by two enclosing bars. For example, if $\mathcal{M}$ is defined as the set $\{\mathcal{P}, \mathcal{T}\}$, then $|\mathcal{M}|$, read the cardinality of $\mathcal{M}$, equals two. Cardinality is a very important metric[2] for the research presented in this thesis. For example, in reasoning about the number of faces in a mesh or discussing the count of points in a neighborhood.

[1] In some literature, zero is excluded from the set of natural numbers.

[2] Closely related to cardinality is the concept of a census which, as used in this thesis, is devoid of any special symbol, but is defined as the total count of all members, of every member, of a multidimensional set or family of sets. An algorithm for computing a census is presented in Section 5.1.2.

2.1.1.4. Membership & Relational Operators

If an object is said to be a member of a set, it is written with the symbol $\in$ and expressed as either "belonging to", or "being an element of", or simply being "in" the set. Similarly, a set may be the subset of another set, written with the symbol $\subseteq$, meaning that every member of the first set is also a member of second set. A superset works exactly in reverse, written with the symbol $\supseteq$, it is the identity where every member of the second set is also a member of first set.

In mathematical notation, the membership and relational operators are written as[3]

$$B \in \{A,\ B,\ C\} \tag{2.3}$$

$$\{A,\ B\} \subseteq \{A,\ B,\ C\} \tag{2.4}$$

$$\{A,\ B,\ C\} \supseteq \{A,\ B\} \tag{2.5}$$

Furthermore, a "family of sets" is defined as a multidimensional set, or rather, a set of sets. In general, each member of a family of sets is a set which shares some common quality with the other members of the family. For example, in Section 2.2.5 we will define the set of sets of neighbors as a family of sets, with each member being a set of points comprising a single neighborhood. In this example, it would also be correct to reference to it as simply, a set of neighborhoods.

2.1.1.5. Binary Operations

Among all the basic binary operations one can perform on a set[4], in this thesis, we exclusively use the union operation, denoted as $\cup$, whose output is the set of all objects that are members of either set, or both. For example,

$$\{0,\ 1,\ 2\} \cup \{2,\ 3,\ 4\} = \{0,\ 1,\ 2,\ 3,\ 4\} \tag{2.6}$$

Also noteworthy, are the commutative and associative properties of the union [20] operation, which are analogous to the similarly named properties of the addition operator for scalar values. These properties say that the order of the sets, and how the sets are grouped, do not change the final result of the operation, for example:

$$\begin{aligned} (\{0,\ 1\} \cup \{1,\ 2\}) \cup \{3,\ 4\} &= \{0,\ 1\} \cup (\{3,\ 4\} \cup \{1,\ 2\}) \\ &= \{0,\ 1\} \cup \{1,\ 2\} \cup \{3,\ 4\} \\ &= \{0,\ 1,\ 2,\ 3,\ 4\} \end{aligned} \tag{2.7}$$

The commutative and associative properties of the union operation are exploited in both Algorithm 3 and Algorithm 6 in order to reduce the complexity of the computations required to ultimately convolve the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. In the next section, we will introduce another field of study in mathematics which is essential to the research of this thesis: linear algebra.

[3] The negative operators also exists as $\notin$, $\not\subseteq$, and $\not\supseteq$.

[4] which are the union, intersection, complement, and Cartesian product.

2.1.2. Linear Algebra

Nearly as fundamental as set theory, linear algebra is the branch of mathematics which studies the different methods and representations which may concern a line, including: sets of equations, matrices, transformations, vector spaces, norming functions, et cetera. Linear algebra is used in almost all scientific domains that require the use of mathematics, including geometry and scientific computing, and also provides the symbols and concepts for performing blocks of operations in parallel [40]. In this section, we will briefly introduce only the major concepts used in our research: position vectors, the subtraction of two vectors, and computing the L2-norm of a vector.

2.1.2.1. Position Vectors

The concept of position vectors is of paramount importance to nearly all of the calculations which must be performed by the the Fast One-Ring smoothing filter, because the research in this thesis is focused on the efficient convolution of the filter over a discrete manifold embedded in 3D-space, with each point being represented by a set of three Cartesian coordinates, equating points with position vectors.

As in 3D-data, all position vectors are represented by a set of Cartesian coordinates with cardinality matching that of the dimensionality of the Euclidean space in which it is embedded [44]. For example, in Section 2.2.1 we say that a point can also be called a position vector, so if we are given a point in $\mathbb{R}^3$, it is defined by the three Cartesian coordinates x, y, and z. "Cartesian" refers to the French scholar, René Descartes [33], and "Euclidean" refers to Euclid of Alexandria, the Ancient Greek mathematician [32].

Figure 2.1 provides three examples of why a point may be called a position vector. Any set of coordinates has an implied direction pointing away from the origin, which in $\mathbb{R}^3$ is located at $(0, 0, 0)$. In (a), the point $\mathbf{p}_a$ is in $\mathbb{R}^1$ and located at at coordinate (4), which is at a distance of 4 units away from the origin. In (b), the point $\mathbf{p}_b$ is in $\mathbb{R}^2$ and located at at coordinates (4, 4), which is at a distance of $4\sqrt{2}$ units away from the origin. In (c), the point $\mathbf{p}_c$ is in $\mathbb{R}^3$ and located at coordinates (4, 4, 4), which is at a distance of $4\sqrt{3}$ units away from the origin[5]. This concept extends to any Euclidean space $\mathbb{R}^n$.

2.1.2.2. Subtracting Two Vectors

Among the various binary operators with which one could operate on vectors, this thesis is primarily concerned with subtraction, because it will be used to not only calculate the distance between each pair of adjacent points in the mesh, but also to interpolate and extrapolate the function values during the calculations of the weighted mean function values described in detail in Sections 3.4 and 3.5.

In order to use the subtraction operator on two vectors in the same Euclidean space, one must simply subtract each component element-wise. [43] For example, given two points in $\mathbb{R}^3$, A and B, the difference is calculated as

$$A - B = (A_x - B_x,\ A_y - B_y,\ A_z - B_z) \tag{2.8}$$

This results in the single set of new coordinates, $\{C_x, C_y, C_z\}$, which both represents the point C located there, or the position vector pointed from the origin to there.

[5] The lengths in this example increase with each additional dimension: $4 < 4\sqrt{2} \approx 5.657 < 4\sqrt{3} \approx 6.928$.

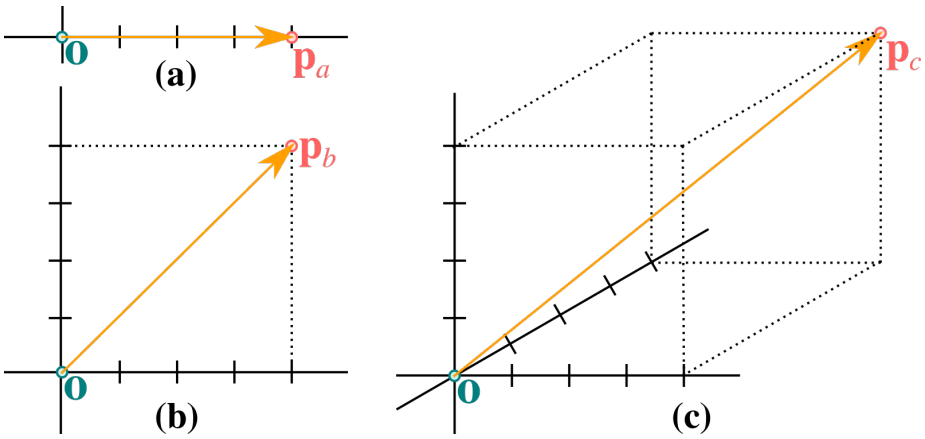


Figure 2.1. Three examples of position vectors shown as arrows in sand color pointed from the origin in teal color to the points in coral color: (a) $\mathbf{p}_a$ in $\mathbb{R}^1$ at coordinate (4) with length 4, (b) $\mathbf{p}_b$ in $\mathbb{R}^2$ at coordinates (4, 4) with length $4\sqrt{2}$, (c) $\mathbf{p}_c$ in $\mathbb{R}^3$ at coordinates (4, 4, 4) with length $4\sqrt{3}$. The origins are located at (0), (0, 0), and (0, 0, 0) respectively.

2.1.2.3. L2-norm

The L2-norm may be seen elsewhere in the literature abbreviated as L^2 or ℓ^2, or perhaps called the "Euclidean norm" or "Euclidean distance". It is so named because it is the ordinary, straight line distance between two points in Euclidean space, and in the case of a position vector $\mathcal{P}$ in $\mathbb{R}^3$, those two points are the origin at $(0, 0, 0)$ and $\mathbf{p}$ at the coordinates $\{\mathcal{P}_x, \mathcal{P}_y, \mathcal{P}_y\}$ [37].

The purpose for calculating the L2-norm of a vector is to determine the ordinary, unidimensional length of the vector, regardless of the dimensionality of Euclidean space in which it is embedded. The significance of this distinction can be seen in Figure 2.1; despite having all coordinates exclusively set to 4, the position vector on (a) is of length 4, in (b) it is of length $4\sqrt{2} \approx 5.657$, and in (c) it is of length $4\sqrt{3} \approx 6.928$.

To determine the length of a vector, one can use the following equation for calculating the L2-norm,

$$|\mathbf{p}| \;=\; \|\mathbf{p}\|_2 \;=\; \sqrt{x^2 + y^2 + z^2 + \ldots + n^2} \;:\; \mathbf{p} \in \mathbb{R}^n \tag{2.9}$$

In this thesis, the L2-norm is denoted using the abbreviated symbol $|\cdot|$. Please be aware of the similar notation for the cardinality of a set $|\mathcal{P}|$. While the matching symbols do represent somewhat similar concepts, cardinality is the count of elements in the set, but the L2-norm of a vector is calculated as in Equation 2.9. [26, p. 26]

2.1.3. Geometry

As vast and encompassing as the field of geometry is, in this section, we will only introduce the few concepts which are of paramount importance for the Fast One-Ring smoothing filter, including: geodesic discs for ensuring a consistent filter window size for the duration of each convolution in Section 3.1, the characteristics of a circle sector to be used in calculating the area and center of gravity in Section 3.3, as well as

interpolation and extrapolation for calculating the weighted mean function values in Section 3.5.

2.1.3.1. Geodesic Discs

Geodesy, the term from which geodesic discs get their name, is defined differently in many fields [7]. A geodesic, as defined in the original sense, was the shortest route between two points on the Earth's surface; so a straight line distance in curved space. In this thesis, we use the term geodesic disc to mean the surface area on a manifold, encompassed by a circle with a given radius, which itself is also embedded in $\mathbb{R}^3$, and reference to this disc with the symbol Ω.

The significance is that one may treat similarly all adjacent faces in a neighborhood of a non-planar mesh as if they were planar, greatly simplifying the required mathematics involved. It also becomes possible to ensure a consistent filter window size for the duration of each convolution of the Fast One-Ring smoothing filter, which is imperative in order for the output signal to be mapped correctly back onto the scalar field [21, p. 106-112], despite that given the nature of acquired 3D-data, as discussed in Section 2.2.6, each adjacent face likely exists on a different plane in $\mathbb{R}^3$ than its neighbor. Also, from the geodesic disc we derive the circle sectors which are described in detail in the next section.

2.1.3.2. Circle Sectors

A circular sector, or circle sector, is the portion of a disc enclosed by two radii and an arc. In general, the larger area is known as the major sector, however the Fast One-Ring smoothing filter is only concerned with the area of the other, smaller area, known as the minor sector. This is a natural way to partition a geodesic disc embedded in $\mathbb{R}^3$, where each sector can be delineated by the border of two triangular faces which likely exists on different planes. Going forward, the minor circle sector will be abbreviated as just "the circle sector", or even more simply, "the sector" , and will be denoted as $\mathbf{s}$, or $\mathbf{s}_i$ in reference to a specific circle sector, as described in Section 3.1.

As the sector can be described entirely by the central angle α and the circle's radius[6] $\ell_{a,b}$, the area of the sector can be given as

$$A = \pi \ell_{a,b}^2 \frac{\alpha}{2\pi} = \frac{\ell_{a,b}^2 \alpha}{2} \tag{2.10}$$

because of the fact that in radians, the ratio of a sector's central angle α to the complete angle of the circle 2π, is equal to the ratio of that sector's area to the area of the whole circle [35].

2.1.3.3. Center of Gravity

Along with area, the other important characteristic of a circle sector for the Fast One-Ring smoothing filter is the centroid, or center of gravity, which is used in both Algorithms 5 and 9 in order to calculate the weighted mean function values, by serving

[6] $\ell_{a,b}$ was chosen to represent the radius instead of the much more common r, because in 3D-data, edges are not stored, yet distances between points can be calculated as shown in Section 2.2.3.

as the location to which the function values are interpolated, the operation which is discussed in the next section.

The center of gravity is so named, because it is the point where the sector would balance on a pin[7]. The center of gravity **c** is calculated as the distance from the center point $\mathbf{p}_a$, along the line which bisects the central angle α, and is given by

$$\check{\ell} := \frac{4\,\overline{\ell_{\min}}\,\sin(\frac{\alpha_i}{2})}{3\,\alpha_i} \tag{2.11}$$

In the Figure 2.2, a circle sector is produced by the points $\mathbf{p}_a$, $\mathbf{p}_b$, and $\mathbf{p}_c$, has the central angle α which is bisected by a dotted line, and $\ell_{a,b}$ representing the length between points $\mathbf{p}_a$ and $\mathbf{p}_b$, which is equal radius of the circle. Also shown, is the center of gravity **c**, as well as $\check{\ell}$, the distance to **c** from the center point $\mathbf{p}_a$.

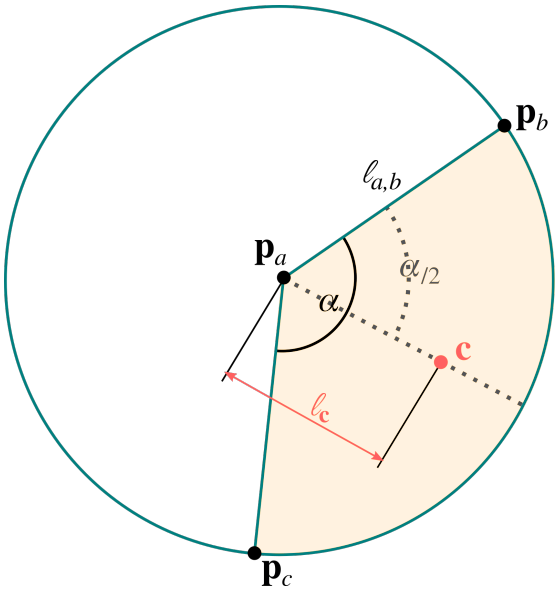


Figure 2.2. The circle sector produced by points $\mathbf{p}_a$, $\mathbf{p}_b$, and $\mathbf{p}_c$, with α as the central angle bisected by a dotted line, and $\ell_{a,b}$ as the length between points $\mathbf{p}_a$ and $\mathbf{p}_b$, which is equal to the radius of the circle. Also labeled is the center of gravity **c** as well as the distance from the center point $\mathbf{p}_a$ to **c**, drawn in a coral color as $\check{\ell}$

The line which bisects the central angle of a circle sector, while not used explicitly in calculations, is an important concept for much of the computations used by the Fast One-Ring smoothing filter, as will be seen in Chapter 4. Therefore, we make a special note here that the abbreviated, "bisecting line", indeed refers to this line which splits the circle sector and its central angle α into two equivalent halves.

2.1.3.4. Interpolation & Extrapolation

Simply stated, interpolation is a method of constructing new data points within the range of a discrete set of known data points, such as those points found in the discrete

[7] given that it had been bestowed a volume with uniform density.

manifolds of 3D-data. There exist a large variety of methods for interpolating different kinds of data [36], but in this thesis, we will only consider linear interpolation, which is simple to calculate[8] and can be extended for n-dimensions [36].

Given the two data points in $\mathbb{R}^2$, (x_a, y_a) and (x_b, y_b), we can calculate the y_c coordinate of a third point located along a straight line drawn between the first two points, while also having the coordinate x_c, as

$$y_c = y_a + (y_b - y_a)\frac{x_c - x_a}{x_b - x_a} \tag{2.12}$$

To elaborate the concept of interpolation, now consider the two points $(1, 2)$ and $(5, 1)$, then find the y coordinate of a point between them, located at the intersection of the line which contains the first two points and $x = 3$.

$$y_c = 2 + (1 - 2)\frac{3 - 1}{5 - 1} \tag{2.13}$$

$$= 2 + -1\frac{2}{4} \quad = \quad 1.5 \tag{2.14}$$

Therefore, we interpolate the value halfway between the two given points to be $(3,\ 1.5)$, which is halfway between as expected, and illustrated in Figure 2.3.

Conversely, extrapolation is a method for constructing new data points *outside* the range of a discrete set of known data points. The process of linear extrapolation is equivalent[9] to linear interpolation; namely, first calculating the line between two points, then calculating the y coordinate of a third point at a new value of x, the latter of which will be either greater than or less than both of the first two points.

As illustrated by Figure 2.3, the similarities and differences between the processes of interpolation and extrapolation become even more intuitive when seen graphically, where having chosen the two points $\mathbf{p}_a$ and $\mathbf{p}_b$, a third point $\mathbf{p}_c$ is interpolated between the two points, while the fourth point $\mathbf{p}_d$ is extrapolated outside their domain.

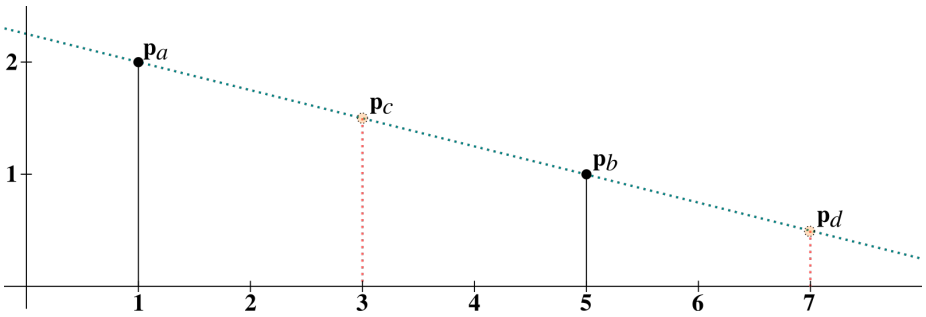


Figure 2.3. The point $\mathbf{p}_c$ is interpolated as the value between $\mathbf{p}_a$ and $\mathbf{p}_b$, while the fourth point $\mathbf{p}_d$ is extrapolated outside their domain

[8] however can become imprecise in relation to the square of the distance between the data points.

[9] The difference between interpolation and extrapolation is largely an academic one, owing primarily to the lack of boundaries and increasing error rates associated with extrapolation.

Later in this thesis, we will use both interpolation and extrapolation together in order to calculate the weighted mean function values at the center of gravity of circle sectors, so that the function values may be weighted fairly in regards to their distance to the other points in the one-ring neighborhood, as discussed in detail in Section 3.4.

2.1.4. Topology

Topology is the mathematical study of the properties of space that are preserved through deformations, twistings, and stretchings of objects[10]. Topology developed as a field of study from geometry and set theory through analysis of concepts such as space, dimension, and transformation. For example, a circle is topologically equivalent to an ellipse, because it can be deformed by stretching. [42]

Of all the major concepts studied in topology, this thesis is only concerned firstly with manifolds, as 3D-data is composed primarily of two-dimensional discrete manifolds embedded in three-dimensional space, and secondly, neighborhoods, which are the general concept from which one-ring neighborhoods are derived, thus being the basis upon which the Fast One-Ring smoothing filter is founded.

2.1.4.1. Manifolds

A manifold is a topological space that is locally, but possibly not globally, Euclidean; a concept that is central to many parts of geometry because it allows complicated structures, such as the triangle mesh, to be described and understood in terms of the simpler, local topological properties. Expressed another way, this means that a manifold is an nD-subset of an Euclidean space $\mathbb{R}^{>n}$ [26, p. 199]. For example, 1-dimensional manifolds in $\mathbb{R}^2$ include lines and circles, and 2-dimensional manifolds in $\mathbb{R}^3$, also called surfaces, can include commonly-known shapes such as the plane, the sphere, and the torus, but also of prime importance for this thesis, the triangle mesh.

The significance to our research is that each convolution of the Fast One-Ring smoothing filter is comprised of calculations of the weighted mean function values at the central point of a one-ring neighborhood, having weighted the function values in relation to the distance between each neighbor, and because those distances are uniformly dissimilar in acquired 3D-data [26, p. 29], maintaining a consistent filter window size is only accomplished by defining the geodesic disc, which exists on the manifold defined by the mesh. Therefore, as mentioned in Section 2.1.3.1 and discussed in more detail in Chapter 4, the weights may be calculated sector-wise, as if all sectors were on the same plane, greatly reducing the complexity of the entire procedure.

2.1.4.2. Neighborhoods

A neighborhood is also one of the basic concepts of topology. Intuitively speaking, a neighborhood of a point is a set of points containing that point, where one can move in any direction without leaving the set. While determining the neighborhood is trivial for uni-dimensional manifolds[11], and relatively simple for regular two-dimensional manifolds[12], determining the neighborhood becomes more complicated for irregular

[10] but not tearing or gluing.

[11] where each neighborhood only consists of a point, and the two points to either side of it on the line.

[12] such as with a 2D-image, whose pixels have at most four neighbors in orthogonal directions with a geometric distance of 1, as well as four neighbors at the diagonal directions with a geometric distance of $\sqrt{2}$.

surfaces like those found in 3D-data. With those surfaces, one must determine the set of neighbors using the associations that define the triangular faces; the details of which are covered in detail in Section 2.2.5, once the other necessary topics regarding 3D-data have already been discussed.

Figure 2.4 illustrates the differences among one-ring neighborhoods in different kinds of manifolds: (a) is a regular square mesh as with pixels of a digital image, (b) is a regular triangle mesh, as in a hexagonal tessellation, and (c) shows two very different neighborhoods in a single, irregular triangle mesh typical of acquired 3D-data. In particular, notice in (c) the completely arbitrary shape and size of the one-ring neighborhoods found in the irregular triangle mesh; even the number of neighbors varies widely. From this observation, came much of the motivation behind the Fast One-Ring smoothing filter.

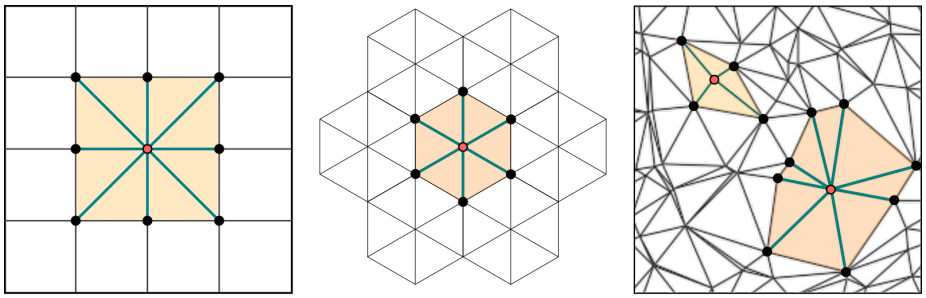


Figure 2.4. One-ring neighborhoods in (a) a regular square mesh, as in pixels of a digital image (b) a regular triangle mesh, as in a hexagonal tessellation (c) an irregular triangle mesh, typical of acquired 3D-data.

2.2. 3D-data

The data upon which one convolves the Fast One-Ring smoothing filter is called 3D-data [26, p. 29]. As described in Section 2.2.4, 3D-data consists primarily of a single mesh $\mathcal{M}$, which is composed of $\mathcal{P}$, a set of points, and the set $\mathcal{T}$, consisting of triangular faces, each to be covered in Sections 2.2.1 and 2.2.2 respectively. As discussed in Section 2.2.6, 3D-data also comes in two distinct flavors depending on its origin: acquired or synthetic. The data can also include a texture map and other various types of information stored as scalar or vector fields, which is elaborated on in 2.2.7.

2.2.1. Points

A point **p** is the most primitive element of 3D-data. "Point" is the abbreviated form of "measuring point", and is also known in other fields of study as a vertex, or a position vector $\mathbb{R}^3$ [44]. A point is defined by the 3-dimensional Cartesian coordinates x, y, and z, and in 3D-data, points are generally unique and not required to be in any particular order. In this thesis, a point is addressed using several different subscripts, depending on the context.

In this thesis, when referring to a point in $\mathcal{P}$, v is used as the globally unique index; the

index with which we can define the set

$$\mathcal{P} := \{\, \mathbf{p}_v \mid v \in \mathbb{N},\ \text{and } 1 \leq v \leq v_{max} \,\} \tag{2.15}$$

where v_{max} is the maximum index[13] of points in the data, and is equivalent to the cardinality of the set of points $|\mathcal{P}|$.

Otherwise, when referencing to a point within a particular face or neighborhood, the three corners can be referenced indirectly[14] as $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, and $\mathbf{p}_{i+2}$, or directly as $\mathbf{p}_0$[13], $\mathbf{p}_1$, $\mathbf{p}_2$, etcetera. [26, p. 25]

2.2.2. Faces

Faces are another primitive element of 3D-data. As we are working exclusively with triangular meshes [26, p. 26], we define a face $\mathbf{t}$ by the set of three distinct points, which we will index in clockwise order[15] [27, p. 4].

$$\mathbf{t} := \{\, \mathbf{p}_a,\ \mathbf{p}_b,\ \mathbf{p}_c \} = \{\, \mathbf{p}_1,\ \mathbf{p}_2,\ \mathbf{p}_3 \} \tag{2.16}$$

With the introduction of faces comes the concept of adjacency. A face $\mathbf{t}_i$ is said to be adjacent to another face $\mathbf{t}_j$ if, and only if, they share together the same subset of two points. Similarly, a point $\mathbf{p}_a$ is said to be adjacent to another point $\mathbf{p}_b$ if, and only if, the set $\{\mathbf{p}_a, \mathbf{p}_b\}$ is a subset of at least one face $\mathbf{t} \in \mathcal{T}$. A synonym for adjacent is "neighboring", and this thesis will use both interchangeably.

Figure 2.5 shows two adjacent triangular faces, $\mathbf{t}_1$ and $\mathbf{t}_2$, and the relationship between their points, $\mathbf{p}_a$, $\mathbf{p}_b$, and $\mathbf{p}_c$; their edge lengths, ℓ_a, ℓ_b, and ℓ_c; and their clockwise orientation. The concept illustrated in this figure is fundamental to understanding the structure of all 3D-data, likewise also the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, and yet may not be immediately intuitive without proper context, therefore we reference this figure several times throughout the rest of this thesis.

Each distinct triangular face is addressed using the global index k, and when taken together, comprise the set

$$\mathcal{T} := \{\, \mathbf{t}_k \mid k \in \mathbb{N},\ \text{and } 1 \leq k \leq k_{max} \,\} \tag{2.17}$$

[13] Beginning indices with 1 is significant because of the discordant conventions between addressing the first element of a data structure with 0 in programming languages, such as C++ and python, versus addressing first elements with 1, as is the standard for mathematical literature. Because this thesis should indeed be considered mathematical literature, we will always start indices at 1, and only use index 0 for special cases. For example, $\mathbf{p}_0$ is used in Chapter 4 to represent the center point of the one-ring neighborhood.

[14] or even more generally, as $\mathbf{p}_a$, $\mathbf{p}_b$, and $\mathbf{p}_c$ in the case of Equation 2.18, or in the case of a nested loop: as $\mathbf{p}_j$ and $\mathbf{p}_{j+1}$ in Algorithms 5 and 9.

[15] It is worth mentioning here that many software packages, such as the GigaMesh Framework [26, p. 89], may expect counter-clockwise ordering of indexes. This is significant because the ordering provides an orientation by which visualization software can apply texture and/or lighting. The only mathematical significance of the ordering is the sign of the area of a face, and totally inconsequential when the absolute value is expected, as shown by [16, p. 2]. This is yet another example of the difference between the conventions of mathematical literature versus those of computer science. And again, because this thesis should indeed be considered mathematical literature, we will continue to follow the conventions of mathematical literature.

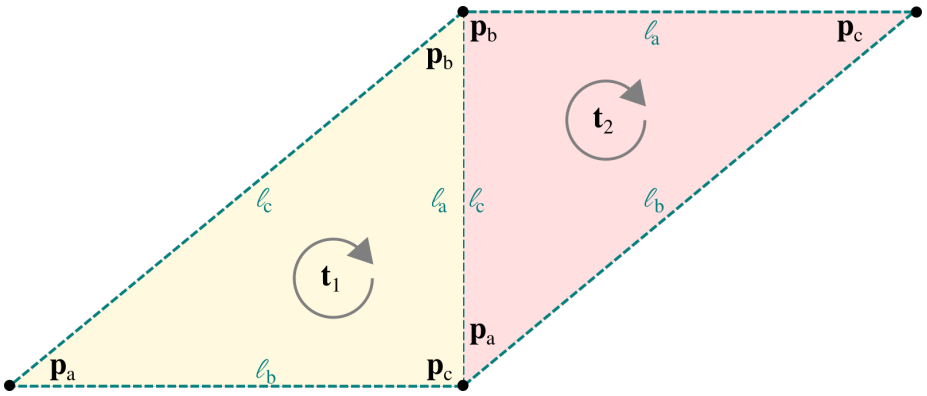


Figure 2.5. Two adjacent triangular faces, $\mathbf{t}_1$ and $\mathbf{t}_2$, showing the relationship between their points, $\mathbf{p}_a$, $\mathbf{p}_b$, and $\mathbf{p}_c$; their edge lengths, ℓ_a, ℓ_b, and ℓ_c; and their clockwise orientation.

where k_{max} is the maximum index of faces in the data, and is equivalent to the cardinality of the set of faces $|\mathcal{T}|$.

2.2.3. Edge Lengths

As illustrated in Figure 2.5, each triangular face is implicitly composed of three edges. Despite the fact that an edge is not typically[16] a primitive element of 3D-data, we will endeavor to define the length of an edge ℓ, because edge lengths are of particular significance for both the design and implementation of the Fast One-Ring smoothing filter.

What is also particularly interesting about Figure 2.5, is that when edge lengths are defined by the points of specific faces, each non-border edge length, illustrated in the figure as ℓ_a in $\mathbf{t}_1$, and ℓ_c in $\mathbf{t}_2$, will be labeled twice, despite representing the same distance.

When in the context of a particular face $\mathbf{t}_k$, we will use a single index to define the length

$$\ell_a := |\mathbf{p}_b - \mathbf{p}_c| \quad : \quad \{\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c\} = \mathbf{t}_k \tag{2.18}$$

and double indices when an edge length is referenced in relation to a specific point $\mathbf{p}_v$ and its neighbor $\mathbf{p}_i$

$$\ell_{v,i} := |\mathbf{p}_i - \mathbf{p}_v| \tag{2.19}$$

Please be aware of the similar notation for the cardinality of a set $|\mathcal{P}|$, to that for the calculation of the length of the edge $|\mathbf{p}_i - \mathbf{p}_v|$, as defined in Equation 2.19. While cardinality is simply the count of elements in the set, the length is calculated as the L2-norm of the referenced vector [26, p. 26].

[16] Other data structures, such as the Winged-Edge [12, p. 1], may use edges as a primitive element.

$$\begin{aligned} |\mathbf{p}_i - \mathbf{p}_v| &= \|\mathbf{p}_i - \mathbf{p}_v\|_2 \\ &= \sqrt{(x_i - x_v)^2 + (y_i - y_v)^2 + (y_i - y_v)^2} \end{aligned} \tag{2.20}$$

2.2.4. Discrete Meshes

In 3D-data, a mesh $\mathcal{M}$ is the digital representation of a discrete manifold embedded in $\mathbb{R}^3$, and is typically[17] two-dimensional, non-planar and comprised of non-regular, triangular faces composed of connected points. [26, p. 25] A mesh is the set defined as

$$\mathcal{M} := \{\mathcal{P}, \mathcal{T}\} \tag{2.21}$$

with the set $\mathcal{P}$ consisting of points, and the set $\mathcal{T}$ consisting of triangular faces.

Many 3D-scanners produce point clouds, which as the name suggests, are comprised solely of a set of points $\mathcal{P}$, and do not provide the set $\mathcal{T}$. However, it is possible, and necessary for the production of a mesh, to perform a point set triangulation in order to connect $\mathcal{P}$ into a set of triangle faces, enabling one to combine the two sets into a mesh [26, p. 26]. For example, a portion of our experiments use a random triangulation disc generator, as presented in Section 6.1.4, which performs the well-known Delaunay triangulation [17] in order to produce a mesh from a randomly generated point cloud.

2.2.5. One-Ring Neighborhoods

It was mentioned in Section 2.1.4.2, that it is a non-trivial task to determine the neighborhood of a point $\mathbf{p}_v$, which exists in an irregular triangle mesh embedded in three dimensions. Having now examined the definitions of points, faces, and meshes, we can now formalize the one-ring neighborhood in 3D-data as

$$\mathcal{N}_v := \{\ \mathbf{p}_i\ :\ \{\mathbf{p}_i, \mathbf{p}_v\} \subseteq \mathbf{t} \quad \forall \mathbf{t} \in \mathcal{T}\ \} \tag{2.22}$$

In plain English, this means that the neighborhood $\mathcal{N}_v$ is defined as the set of points $\mathbf{p}_i$ such that both $\mathbf{p}_i$ and $\mathbf{p}_v$ are two points of a triangular face $\mathbf{t}$, for all faces in the mesh. Therefore, the adjacency of points is defined by the co-membership within a face $\mathbf{t}$, from the set of faces $\mathcal{T}$.

When taken together, all the neighborhoods $\mathcal{N}_v$ comprise a set of sets of neighbors; the family of sets

$$\mathcal{N} := \{\mathcal{N}_v, \mathcal{N}_{v+1}, \ldots, \mathcal{N}_{v_max}\} \tag{2.23}$$

This definition shares the indices v and v_{max} with the definition for a set of points, Equation 2.15, to highlight the fact that neighborhoods are defined per point. Therefore, there must be a correlation in the cardinality between the two sets.

[17] except in the case of specifically designed synthetic data, such as those presented in Section 6.1.

Conversely, the cardinality of each individual neighborhood $|\mathcal{N}_v|$ will likely vary among other neighborhoods in $\mathcal{N}$. However, the value must always be ≥ 2, and though there is no upper limit, $|\mathcal{N}_v|$ is typically ≤ 12. Furthermore, the neighboring points are always indexed in a clockwise direction for the same reasons as discussed in Section 2.2.2.

As illustrated in Figure 2.4, the set of black points are the neighbors $\mathbf{p}_i$ which belong to the one-ring neighborhood[18] $\mathcal{N}_v$ centered on point $\mathbf{p}_v$, which is drawn in coral color. The moniker originates from graph theory where each adjacent point is said to have a relative distance of one within the graph of the mesh.

Our solution for maintaining a constant filter window size while convolving the Fast One-Ring smoothing filter over neighborhoods of varying shapes and sizes is to determine the global minimum edge length, then use that as the radius of the geodesic disc centered at the central point of each neighborhood. Then when calculating the weighted mean function values of each circle sector, using the interpolated function value from the neighboring points at the center of gravity. This process is explained in greater detail later in this thesis, and is the topic of Chapter 4.

2.2.6. Acquired versus Synthetic 3D-data

The corpus of all 3D-data exists in two flavors, acquired and synthetic data, with each being handily classifiable by the fashion in which it was generated, and the characteristics innate to those techniques.

Acquired 3D-data is typically captured as a point cloud utilizing various methods, such as: LiDAR (Light Detection and Ranging), Structured Light, or Structure from Motion [26, p. 19]. Then the data is exported from software packages accompanying the 3D-scanners as either just a point cloud comprised of a simple set of points $\mathcal{P}$, or optionally as a triangle mesh $\mathcal{M}$, described by one or more scalar-fields. Acquired 3D-data also often consists upwards of a million points and as many as twice that number of faces[19]. These exported meshes [26, p. 25] uniformly contain noise and may exhibit other complexities for analysis, such as: non-manifold points, multiple borders and holes in the surface, inverted face orientation, non-manifold edges, and agglutination or degenerate faces [26, p. 28-32].

Conversely, synthetic 3D-data is artfully-crafted to avoid the complexities exhibited by acquired data. When modeling 3-dimensional objects, synthetic 3D-data can require significantly less memory for storage, as simplifications can be made for large regular surfaces. For example, even the largest flat, rectangular surface can be modeled with only four points and two faces, whereas the acquired data methods require that the Nyquist–Shannon sampling theorem [21, p. 249-250] be obeyed for the smallest detectable feature throughout the entire surface [26, p. 19] [27, p. 3].

As presented in Section 6.1, for a portion of our experiments, we created another kind of synthetic 3D-data which does not model a 3D-object. Instead, these synthetic-mesh

[18] A one-ring neighborhood can be extended to become a two-ring neighborhood by taking the union of the neighborhood $\mathcal{N}_v$ with the neighborhood of each one-ring neighbor $\mathbf{p}_i$, which adds to the neighborhood points with the relative distance of two within the graph. This iterative concept can be repeated k times and the neighborhood is then referred to as k-ring. Unfortunately, regardless of the fact that the relative distance within the graph can not be assumed equal to the geometric distance nor geodesic distance, it is still used throughout the literature, especially in the field of Computer Graphics [26, p. 29].

[19] Appendix A presents an interesting trend concerning how the ratio of face counts to point counts approaches to two for increasing mesh sizes, which is somehow related to Euler's polyhedron formula.

generators produce different types of tessellations on arbitrarily large, planar surfaces, accompanied by a configurable scalar-field of function values.

2.2.7. Function Values

In addition to the three Cartesian coordinates, points which comprise a mesh may also contain other relevant data in the form of scalar fields, or vector fields, when combined together into multi-dimensional data. This information, which is stored at each point **p**, often includes data regarding RGB color, material type, reflectivity, transparency, quality, confidence, or the resulting function values from an analytical filter such as the Multi-Scale Integral Invariants filter (MSII) [26, p. 21]. A scalar field can also be extended to include data important to other fields of study, such as infrared or ultraviolet light, temperature, rainfall, population, crime-rates, etc; essentially, any kind of data that may be measured at a point. As the Fast One-Ring smoothing filter will currently only process a single field at a time, we can define a scalar field simply as the set of function values

$$\mathcal{F} := \{\, f_v \mid v \in \mathbb{N}, \text{ and } 1 \leq v \leq v_{max} \,\} \tag{2.24}$$

This definition shares the indices v and v_{max} with Equation 2.15, the definition for a set of points, because it is indeed a fact that the cardinality of $|\mathcal{F}|$ must be equal to $|\mathcal{P}|$

$$|\mathcal{F}| \stackrel{!}{=} |\mathcal{P}| \tag{2.25}$$

g because function values are only stored alongside the Cartesian coordinates, one-to-one with points, due to 3D-data existing as a discrete manifold. The significance of which is another motivating factor behind the research conducted in this thesis.

2.3. Parallel Processing

In this section, limited topics regarding the design and analysis of algorithms for parallel processing are briefly discussed, including: the nature of serial computation and threads; SIMD as an architecture of concurrency; GPGPUs as a tool for parallel processing; the identification of control and data dependencies in an algorithm; various topics regarding program correctness, including: the identification of critical sections, the utilization of mutexes as a locking mechanism, and the costliness of explicit thread synchronization; and finally the metrics available for evaluating a parallel algorithm.

2.3.1. Serial Computation & Threads

In a very general sense, when a computer executes a serial program, it proceeds one instruction at a time, sequentially in the order that it is written in the source code, and as required, reads from and writes to memory, which must then be reserved for it. Individually, these processes are called “threads of execution”, or just “threads”, and a single processing core can only process a single such thread at a time[20].

[20]This is not counting the limited application of parallel processing by exploiting the width of modern register arrays, in order to concurrently compute instructions at half or less precision.

Modern processors, however, can perform fast context switching [30], which means loading the memory and state of a new thread, among a multitude of self-contained threads, and executing it so rapidly, as to give the illusion of parallel processing; similar to the commonly-known phenomenon where individual frames of film, when viewed at a high enough frequency, create the illusion of moving pictures.

Fast context switching has its limitations, as even at the modern frequency of 4GHz [9] per core, a serial algorithm can take dozens of hours, or even days, to finish computing, when iterating many times over a large collections of data, as seen in Section 6.3.1, the portion of our experiments comparing the timing of the serial version and the parallel variant of the Fast One-Ring smoothing filter algorithm.

Alternatively, a system which exhibits some kind of architecture for real concurrency may execute programs which can then "spawn" new threads, which are given their own unique context, and thus may be run simultaneously on individual processing cores; enabling real parallel processing. One such architecture is discussed in the next section.

2.3.2. SIMD - A Concurrency Architecture

All modern computers implement some form of parallelism, be it multi-core or multi-threaded processors, arrays of stream processors as found in GPUs, or the networks comprising supercomputers and data centers, and can therefore be classified by Flynn's Taxonomy of the different architectures of concurrency [19].

Of the four classifications[21] originally defined by Flynn, this thesis is primarily concerned with Single Instruction, Multiple Data (SIMD)[22] systems. As the name implies, the SIMD classification is characterized by systems which are able to take a single instruction, otherwise known as a kernel [4, p. 8], and execute it simultaneously on multiple elements from a pool of data. In general, this is described as "processing in parallel".

Figure 2.6 illustrates a simplified impression of a system which employs SIMD architecture. The array of processors each receive the same kernel from a pool of instructions, then load a different portion of the data pool in order to process to combination in parallel.

The motivation to implement software which utilizes the architecture of an SIMD system is the speedup to be gained, which may be quantified as discussed in Section 2.3.6, and stems from what is known as loop-level parallelism, which is the modification of instructions which would otherwise be executed serially in a loop, to instead be computed simultaneously, in parallel, incurring only minor synchronization overhead. For example: any operation performed in linear algebra, such as the subtraction of two high-dimensional arrays, is thus made much more scalable in regards to the time required for the computation to complete, as vectors of larger and larger sizes are considered.

With the proper programming strategy, using the proper frameworks, the model for parallel processing on SIMD architecture can be extended to far more complex instructions, and indeed, a major portion of the research presented in this thesis is focused

[21] {SISD, SIMD, MISD, and MIMD}

[22] or SIMT as is branded by NVIDIA [4, p. 70-72].

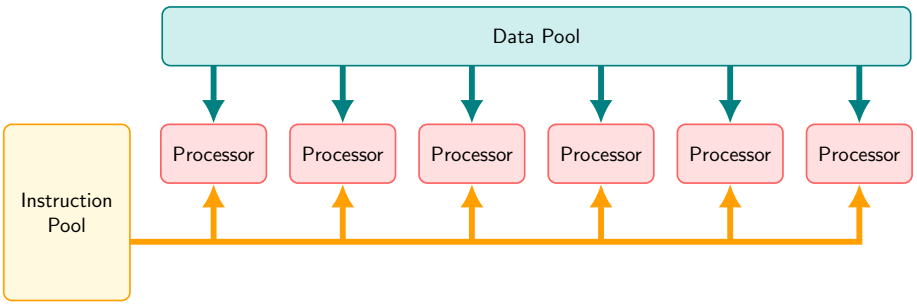


Figure 2.6. A simplified impression of a system characterized as SIMD by Flynn's Taxonomy of concurrent architecture [19]. The array of processors each receive the same kernel from a pool of instructions, then load a different portion of the data pool in order to process the combination in parallel.

on implementing the software to convolve the complex algorithm for the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, by simultaneously executing its various kernels of instructions on every point and one-ring neighborhood in a mesh, in order to fully utilize the parallel architecture of an SIMD system; specifically, commercially-available GPGPUs, which are described in the next section.

2.3.3. GPGPU - General Use, Graphics Processing Unit

General Purpose, Graphics Processing Unit (GPGPU) was coined by Mark Harris, the founder of GPGPU.org [8], and as the name implies, describes GPUs, which was originally developed to exclusively perform graphics processing, but can now be utilized for general purposes by using the frameworks which are publicly available, for example OpenCL [2] and CUDA [4].

Figure 2.7 illustrates the principle difference between a CPU, and a GPU. Because graphics rendering is mostly comprised of compute-intensive, but highly-parallel procedures, GPUs have been designed with less focus on data flow control and more transistors devoted to data processing[23]. This diverges from the design of modern CPUs, which has devoted more chip space to processing at higher frequencies, while using large, multi-leveled caches of memory for the optimization of context switching [30].

2.3.4. Control & Data Dependencies

The concepts of control and data dependencies are not new ones, and can be dated back to the invention of multi-pass compilers. The field of study regarding the details is called dependence analysis and has far-reaching consequences from business management, to economics, as well as software optimization. In general, the basic concept behind both control and data dependencies is that some procedure A is said to be dependent on another procedure B, if A requires B to be executed first.

Figure 2.8 shows the ubiquitous if-then control structure as a simple example of a control dependency. It reads, "if B is true, then do A". In this case A is control dependent on B, because B must execute before it can be determined whether or not to execute A.

[23] mimicking the design of early supercomputers [25, p.7-9]

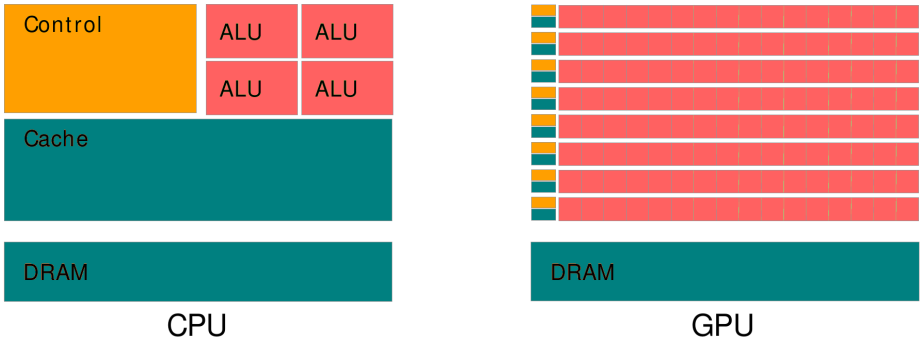


Figure 2.7. An illustration of the principle differences between a CPU and a GPU. CPUs have devoted more chip space to processing at higher frequencies, with more flow control and memory caching, while GPUs have been designed with more transistors devoted to data processing.

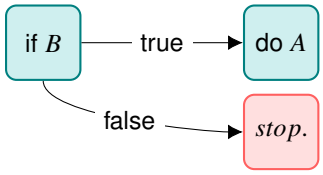


Figure 2.8. A simple example of control dependency using the if-then control structure, with each distinct operation colored in teal, and control dependency marked with an arrow.

There exist four distinct kinds of data dependence. Given that a certain value of data is stored in memory at a definite location and requires a finite amount of time to be read into context in order to be utilized by an operation, the four distinctions of data dependence are:

- "true dependence", which is exhibited when one operation writes to a location in memory which is then later read from by another operation;
- "anti dependence", when one operation reads from a memory location later written to by a second operation;
- "output dependence", meaning that two operations are to write a data value to the same location in memory;
- and "input dependence", when two operations must read a value of data from the same location in memory.

In this thesis, we will generally not make distinctions between the various types, and will simply refer to any instance thereof as just "data dependence".

Figure 2.9 illustrates the less-than-simple example of data dependencies inherent to calculating the L2-norm of a position vector in $\mathbb{R}^3$; an operation which will be revisited often in this thesis, and whose definition was already presented in Equation 2.9. There are six individual procedures involved with calculating the L2-norm: three squarings, two additions, and one square root.

The three squaring procedures are totally independent of each other and can be performed in any order, indeed even concurrently, in relation to one another. In contrast,

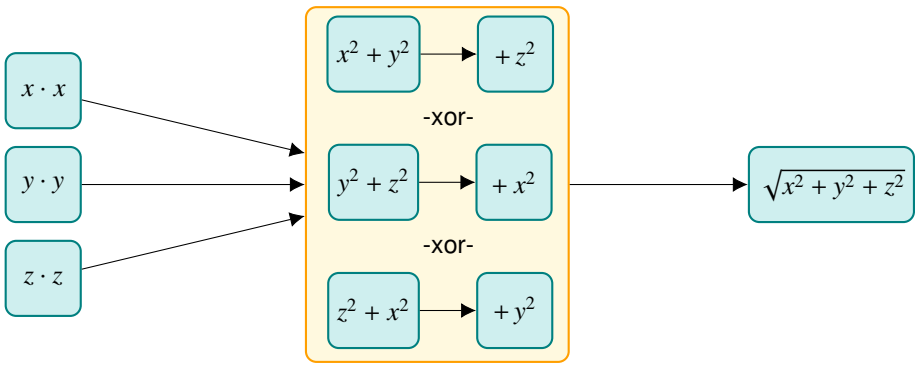


Figure 2.9. The data dependencies inherent to the calculation of the L2-norm of a position vector in $\mathbb{R}^3$, with each distinct operation colored in teal, and data dependencies marked with arrows. The centered block in sand color represents an exclusive-or situation, abbreviated as xor, where one, and only one pair of operations shall execute.

the two additions are totally dependent on their addends, because while the order in which the first two of the three addends are used is inconsequential due to the commutative property of addition, the first addition is always dependent on the completion of two squarings, and the second addition is dependent on not only the sum from the first addition, but also the product of the third squaring. Finally, because the square root procedure is dependent on the execution of both additions, it inherits their dependencies on the squaring procedures, and therefore must wait until the very end before executing last.

A proper understanding of all dependencies becomes even more crucial when designing software which processes data in parallel, because it is no longer implicitly-known when a particular operation has completed, so that the dependent operation may begin. Managing those complications is the topic of the next section.

2.3.5. Program Correctness

Program correctness, which may otherwise be referred to as functional correctness, is defined by the field of theoretical computer science as existing only when, for each input given, a program produces the expected output. In this section, we discuss the dangers inherent to asynchronous parallel processing, mentioning the concepts of critical sections, the mutex locking mechanism, and the explicit synchronization of threads.

2.3.5.1. Critical Sections

One major consequence of processing in parallel is that, due to the independence enjoyed by individual threads, the sequence in which operations are performed can no longer always be guaranteed. That is not intrinsically a problem, however, it becomes a critical issue with the introduction of data dependence, which requires that information be shared among threads by reading from and writing to volatile memory, which is memory that may be changed by any thread without warning.

Sections of code containing the instructions that utilize volatile memory are aptly named “critical section”, and special care must be taken in their regard, in order to

ensure program correctness, despite not being able to determine in what order the operations may occur.

For an illustrative example, consider Algorithm 1, the very simple parallel program to compute the sum of a set of integers. Given the set {1, 2, 3, 4}, the function *parallelSum* instantiates the value s as zero, in order to later accumulate the sum calculated in each thread, then spawns two threads, which each execute the kernel *accumulateSums* with a unique subset of the work to be done. Each kernel then simultaneously adds the two integers, naively reads the current value of s, adds the sum to that value, then stores the result back into s.

Algorithm 1: Simple parallel algorithm, exhibiting an unprotected critical section

Input : the set of integers {1, 2, 3, 4}
Output: the sum s

```
1 Function parallelSum({1, 2, 3, 4})
2     s ← 0
3     ~sum(s, 1, 2)
4     ~sum(s, 3, 4)

5 Kernel accumulateSums(s, a, b)
6     s′ ← a + b
7     s ← s + s′                                   /* Danger! */
```

Unfortunately, the final result of the execution of function *parallelSum* can not be accurately predicted, because the order of operations between each thread are not guaranteed due to what is known as a "raceccondition" in the algorithm. Closer evaluation of line 7 reveals the danger of the raceccondition inherent to this critical section.

The high level code in line 7 will be translated by a compiler into low level assembly instructions [25, p. 25] written in Algorithm 2, which are then executed in parallel by both threads.

Algorithm 2: A low level translation of the critical section in Algorithm 1

```
1 R1 ← s
2 R2 ← s′
3 R3 ← ADD R1 R2
4 s ← R3
```

Figure 2.10 illustrates how, because the value s is in volatile memory which is shared by both threads, and because each thread remains independent, performing their instructions sequentially at their own speeds, the total order of operations can not be guaranteed and will fall within a range of limited permutations. The consequence being, that only one third of the possible outcomes are actually correct, producing the sum of the four integers as would be expected.

In order to manage the complications inherent to the racecconditions engendered by the presence of critical sections in code processed in parallel, it becomes necessary to

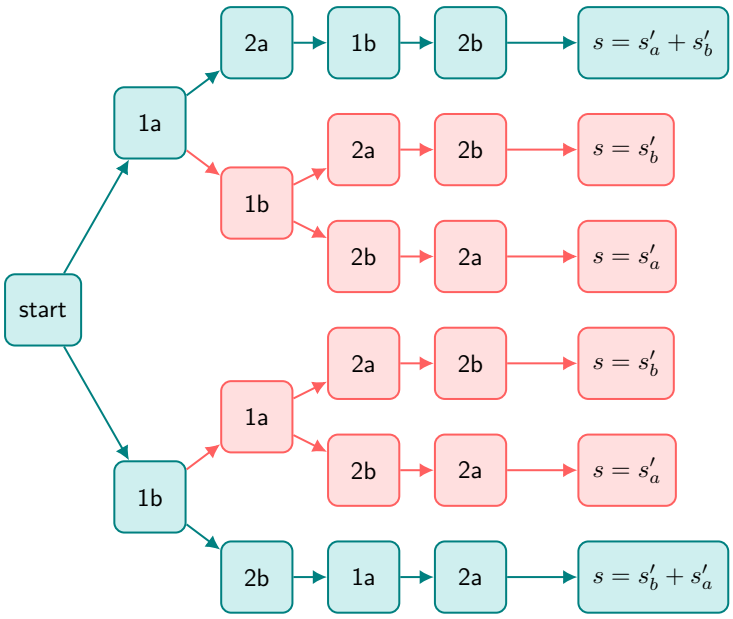


Figure 2.10. the possible permutations of operations as outcome of accumulating a sum with parallel threads without mutual exclusion. The two teal color paths are the only produce correct results, while the four coral colored paths produce only a partial sum.

introduce some type of locking mechanism with which one can control the flow of execution with regards to data dependence. The one and only type of locking mechanism used in this thesis, the mutex, is discussed in the next section.

2.3.5.2. Mutexes

The term “mutex” is an abbreviation for “mutual exclusion”, which is used to describe the situation where the presence of a single entity may exclude the presence of all others. For parallel processing, that translates to allowing only one single thread to enter a critical section of code at a time. A mutex is typically implemented as a shared value which any thread wanting to enter a critical section in code must first find unlocked, before locking the mutex itself and performing the operations therein. All other threads wanting to also enter the critical section will find the mutex locked, and therefore must “block”, waiting until the mutex is again unlocked, before being able to proceed.

While it is true that employing a mutex can be expensive to both compute time and memory because blocking intrinsically means the cessation of computation, however, when used efficiently, the speedup gained by exploiting the parallelism of an algorithm becomes worth the cost of the added synchronization overhead. [25, p.20] An example of this is thoroughly explored in the description of Algorithm 9, and the effects of this overhead can be seen in the results of the timing experiments presented in Sections 6.3.2 and 6.3.3, later in this thesis.

2.3.5.3. Explicit Thread Synchronization

Occasionally, in order to maintain the correctness of the overall program due to some dependence within the code, an algorithm is forced to block, waiting for all threads to complete their calculations, before finally continuing with its computations. This procedure is called, "explicit synchronization" [**?**, p. 34] and is expressed in the parallel algorithms presented in this thesis by a call to the subroutine *synchronizeThreads*.

Explicitly synchronizing threads can be very expensive, costing the overall efficiency of an algorithm greatly; easily reaching the level of billions of compute cycles. Therefore, it is absolutely imperative that one minimizes its inclusion in the design of any parallel algorithm, especially within loops, avoiding it altogether whenever possible, and limiting its use to only the sections of an algorithm which signify the culmination of a large volume of parallel work.

In this section, we discussed the dangers to program correctness that are inherent to parallel processing, including the concepts of critical sections which arise when individual threads must share information via volatile memory, the mutex locking mechanism which can be used to protect critical sections in order to preserve program correctness, and the explicit synchronization of threads which can be very expensive to the overall efficiency of an algorithm. In the next section, we will explore a few of the metrics that can be used to evaluate and analyze parallel algorithms, for use in comparison and possible further optimization.

2.3.6. Evaluation and Analysis of Parallel Algorithms

When engaging in the design of parallel algorithms, it is useful to define metrics which can be used to quantitatively describe the differences engendered by the modifications made to the basic serial algorithm. Such metrics could then influence not only choices in the design, but also environments under which the algorithm should be run. For example, in answering the question regarding the scalability of the algorithm and the worthiness of incurring the costs of increasing the count of processors in the pursuit of faster computation times [25, p. 330].

Timing is essential to every metric used in the evaluation of parallel algorithms. Given the inputs, $\hat{n}$ as the number of operations to be performed by the algorithm, and ρ as the count of processors available in the system, one can define the two essential timings as:

$$T_s(\hat{n}) = \text{The optimal sequential execution time} \tag{2.26}$$

$$T_\rho(\hat{n}, \rho) = \text{Parallel runtime} \tag{2.27}$$

which can then be used to define the three essential metrics:

$$\text{Speedup:} \quad S(\hat{n}, \rho) = \frac{T_s(\hat{n})}{T_\rho(\hat{n}, \rho)} \tag{2.28}$$

$$\text{Costs :} \quad C(\hat{n}, \rho) = \rho \cdot T_\rho(\hat{n}, \rho) \tag{2.29}$$

$$\text{Efficiency :} \quad E(\hat{n}, \rho) = \frac{T_s(\hat{n})}{C(\hat{n}, \rho)} = \frac{S(\hat{n}, \rho)}{\rho} \tag{2.30}$$

By applying these identities, we can determine that for all problem sizes, the speedup obtained is always less than the number of processors used, because otherwise, $T_s(\hat{n})$ could not be the optimal sequential execution time. Also, efficiency will always be less than one thus we can say that $E \cdot \rho$ processors contribute to the solution of the total procedure, while $(1 - E) \cdot \rho$ do not [25, p. 335].

Also important is the notion of an algorithm's "degree of parallelism", which is the maximum number of operations that can be executed in parallel, a principle central to Amdahl's law [10] which is defined as, when given a constant problem size $\hat{n}_{fixed}$, and an algorithm's degree of parallelism q,

$$\lim_{\rho \to \infty} S(\hat{n}_{fixed}, \rho) = 1/q \tag{2.31}$$

This leads to the conclusion that one can not simply add more processors in order to gain appreciable speedup, but instead that relies heavily on processor counts which grow in relation to problem sizes, and the degree of parallelism, which itself relies on the underlying nature of the serial algorithm and the artful design of the parallel algorithm.

2.4. Summary

This chapter briefly covered many topics across many fields of study, opting for breadth over depth in order to focus only on the specific topics which have a direct influence on research presented in the rest of this thesis.

Section 2.1 covered the elementary, theoretical basis for the mathematics utilized in the design of the Fast One-Ring smoothing filter, including various topics in set theory, linear algebra, geometry, and topology. We delved into: expressing definitions, special sets, cardinality, as well as selected membership, relational, and binary operators; position vectors, the subtraction of vectors, and computing the L2-norm; geodesic discs, circle sectors, centroids and the center of gravity, and interpolation and extrapolation; as well as manifolds and neighborhoods.

Next, Section 2.2 defined several symbols which will be used throughout the rest of this thesis. It discussed in relative detail, the primitives of 3D-data: points, faces, and edge lengths, as well as the nature of discrete meshes, one-ring neighborhoods, the differences between acquired and synthetic 3D-data, and the treatment of scalar fields as function values.

Then in Section 2.3, limited topics regarding the design and analysis of algorithms for parallel processing were briefly presented. These included: the nature of serial computation and threads; SIMD as an architecture of concurrency; GPGPUs as tools for parallel processing; the identification of control and data dependencies in an algorithm; various topics regarding program correctness, including: the identification of critical sections, the utilization of mutexes as a locking mechanism, and the costliness of explicit thread synchronization; and finally the metrics available for evaluating a parallel algorithm.

In the next three sections, we will present the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. This begins with the mathematical foundations upon which the the filter was designed, then we present the three parts of the serial algorithm for convolving the filter over a discrete manifold, and we conclude with an analysis of the serial algorithm, and the presentation of a parallel variant of each of its parts.

Chapter 3

Fast One-Ring Smoothing: Mathematical Foundation

In this and the next two chapters, we present an updated version of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, based on the ideas initially proposed by H. Mara and S. Krömker at the EUROGRAPHICS Workshop on Graphics and Cultural Heritage (2017) [27, s. 3.2]. Since publication, further research has been conducted by the authors, and it was determined that improvements to the weighting methods were possible. Therefore, modifications to the algorithm were implemented directly within the GigaMesh framework [29]. This chapter presents the mathematical grounding for the as-yet-unpublished version of the Fast One-Ring smoothing filter, as it exists now in GigaMesh. We have incorperated more accurate weighting methods based on the interpolation and extrapolation of function values to the center of gravity of each circular sector comprising the geodesic disc, centered upon each point in a triangle mesh of 3D-data.

The procedure for designing any convolutional filter must begin by defining the size of the filter window, in a way such that the ouptut signal can be mapped correctly back onto the scalar field [21, p. 106-112], and when designing a filter for acquired 3D-data, that entails the complex tasks of calculating each edge length and determining the global minimum edge length for the entire triangular mesh, as covered in Section 3.1 [27]. Next, because one-ring neighborhoods in acquired 3D-data uniformly have irregular shapes and sizes, the typical characteristics of which can can be seen clearly in Figure 2.4, each function value must be scaled appropriately in regards to the distance from it to all of its neighboring points, as discussed in detail in Sections 3.2, 3.3, and 3.4. Then, as presented in Section 3.5, the next steps for the Fast One-Ring smoothing filter involve the computation of the weighted mean function values for each circle sector of the geodesic disc, then the weighted mean function value for the entire disc, and finally, the convolutions of the filter over the entire mesh.

3.1. The Shortest Edge Length

The entire procedure for the Fast One-Ring smoothing filter for scalar fields on discrete manifolds begins by first calculating every edge length between every pair of adjacent

points in the mesh, then determining the global minimum of those lengths. To accomplish that goal, we start by choosing from the set of points $\mathcal{P}$, a point $\mathbf{p}_v$, which defines the one-ring neighborhood $\mathcal{N}_v$. This neighborhood is comprised of points $\mathbf{p}_i$, adjacent to the center point, given the local index zero; as in $\mathbf{p}_0$. The shortest edge length in the neighborhood of $\mathbf{p}_v$ can therefore be calculated as

$$\ell_{\min}(\mathbf{p}_0) := \min_{\forall \mathbf{p}_i \in \mathcal{N}_v} |\mathbf{p}_i - \mathbf{p}_0| \tag{3.1}$$

which, when utilized as a radius, defines the geodesic disc Ω_v centered on the point $\mathbf{p}_v$.

Figure 3.1 shows a typical configuration of a one-ring neighborhood $\mathcal{N}_v$, characterized by (a) the six irregular faces $\mathbf{t}_i$, which are defined by the six neighboring points $\mathbf{p}_i$, which are all adjacent to the center point $\mathbf{p}_0$, the shortest edge length $\ell_{\min}$, calculated here as the L2-norm of the difference between points $\mathbf{p}_5$ and $\mathbf{p}_0$, and the outline of the geodesic disc Ω_v, as well as (b) the complete geodesic disc Ω, composed of all six of the circular sectors $\mathbf{s}_i$ which it circumscribes.

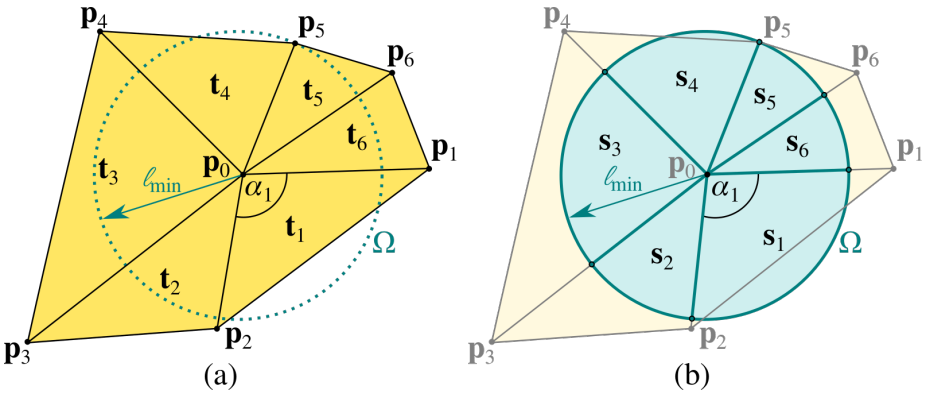


Figure 3.1. A typical configuration of one-ring neighborhood $\mathcal{N}_v$, described by (a) the six irregular triangular faces $\mathbf{t}_i$ in sand color, which are defined by the six neighboring points $\mathbf{p}_i$, which are all adjacent to the center point $\mathbf{p}_0$; the smallest edge length $\ell_{\min} = \ell_{\min}(\mathbf{p}_5) = |\mathbf{p}_5 - \mathbf{p}_0|$, illustrated with a teal arrow as the radius of the geodesic disc Ω_v, which is drawn as a teal colored dotted circl; and α_1 as the central angle of $\mathbf{t}_1$ (b) the complete geodesic disc Ω composed of all six of the circular sectors $\mathbf{s}_i$ which it circumscribes.

Furthermore, to ensure that the filter window size remains static for the duration of each convolution, we define the global minimum edge length as

$$\overline{\ell_{\min}} := \min \{\ell_{\min}(\mathbf{p}_0) \mid \mathbf{p}_0 \in \mathcal{M}\} \tag{3.2}$$

and then use $\overline{\ell_{\min}}$, the shortest edge length between any pair of adjacent points in the triangle mesh $\mathcal{M}$, in lieu of the local minimum edge length $\ell_{\min}(\mathbf{p}_i)$, as the radius of the geodesic disc for all neighborhoods in all the following equations presented in this chapter.

3.2. Interior Angles

Having determined the global minimum edge length $\overline{\ell_{\min}}$ in the previous section, we can now consider the next steps in calculating the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. Because we intend to use the area of each circle sector $\mathbf{s}_i$ comprising the geodesic disc Ω_v as weights upon the scalar field $\mathcal{F}$ for averaging the function values, first per sector, and then per neighborhood, we must initially calculate for use in those later computations, the central angles α_i of each triangular face $\mathbf{t}_i$ in the neighborhood $\mathcal{N}_v$. These central angles may be obtained by applying the law of cosines [38], which would yield

$$\alpha_i := cos^{-1}\left(\frac{|\mathbf{p}_0 - \mathbf{p}_i|^2 + |\mathbf{p}_0 - \mathbf{p}_{i+1}|^2 - |\mathbf{p}_i - \mathbf{p}_{i+1}|^2}{2 \cdot |\mathbf{p}_0 - \mathbf{p}_i| \cdot |\mathbf{p}_0 - \mathbf{p}_{i+1}|}\right) \tag{3.3}$$

or more compactly:

$$\alpha := cos^{-1}\left(\frac{\ell_c^2 + \ell_b^2 - \ell_a^2}{2 \cdot \ell_c \cdot \ell_b}\right) \tag{3.4}$$

Also, as it is our intention to interpolate and extrapolate the function values in relation to the distances between them and each of their adjacent points in the current neighborhood, we will be required to perform some calculations one side of the sector-bisecting line at a time, as will be discussed in detail in Section 3.5, later in this chapter. For now, having computed the central angles α, we can calculate the other interior angles β, in relation to $\overline{\ell_{\min}}$ and opposite the bisecting line. To do so, we use the line tangent to the circle sector at the bisecting line to create two proxy right triangles, which together comprise the larger isosceles triangle, which circumscribes the sector $\mathbf{s}_i$. Next, we can apply the third angle theorem[24] [34] with $\alpha/2$, to calculate β as defined in the formula

$$\beta := \left(\frac{\pi}{2} - \frac{\alpha}{2}\right) = \frac{(\pi - \alpha)}{2} \tag{3.5}$$

Figure 3.2 (a) extends Figure 3.1 by focusing on the circle sector $\mathbf{s}_1$, showing the triangular face $\mathbf{t}_1$, angles α and $\alpha/2$, the bisecting line, the two angles β with the tangent line, proxy right triangles defined by the points $\mathbf{p}'_1$ and $\mathbf{p}'_2$ used in their calculation, and the center of gravity, which is discussed in detail in the next section.

3.3. Area & Center of Gravity

The next step towards the goal of computing the weighted mean function values of the circle sectors $\mathbf{s}_i$, is to calculate for each sector the area A, and the distance $\check{\ell}$ from the center point $\mathbf{p}_0$ along the bisecting line to the center of gravity $\mathbf{c}$. Because a circle sector may be defined entirely by its radius and central angle [35], having now calculated the radius as $\overline{\ell_{\min}}$ in Section 3.1 and the central angle α in Section 3.2, the area of a sector can be calculated using the formula

$$A := \frac{\left(\overline{\ell_{\min}}\right)^2 \alpha}{2} \tag{3.6}$$

[24] otherwise known as the Angle-Angle-Angle Theorem, abbreviated as AAA [34].

Similarly, $\check{\ell}$, the distance from the center point $\mathbf{p}_0$, along the bisecting line to the center of gravity $\mathbf{c}$, can be calculated directly using the formula

$$\check{\ell} := \frac{4\,\overline{\ell_{\min}}\,\sin(\frac{\alpha}{2})}{3\,\alpha} \tag{3.7}$$

Figure 3.2 (a) extends Figure 3.1 by enhancing the circle sector $\mathbf{s}_1$ to illustrate the center of gravity $\mathbf{c}$ and $\check{\ell}$, the distance along the bisecting line from $\mathbf{p}_0$ to $\mathbf{c}$. In general, while holding the radius constant, the closer to $\pi/2$ the central angle α becomes, the longer the distance $\check{\ell}$ will be.

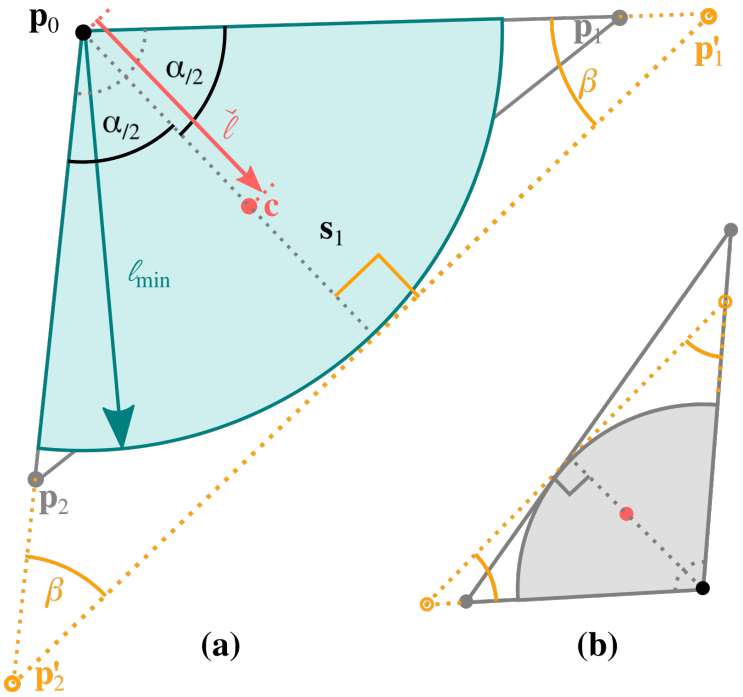


Figure 3.2. An enhanced view of Figure 3.1, focusing on (a) the circle sector $\mathbf{s}_1$, showing the triangular face $\mathbf{t}_1$ and points $\mathbf{p}_1$ and $\mathbf{p}_2$ in gray, with α and the bisecting line in gray dots. In coral color is the center of gravity $\mathbf{c}$ and $\check{\ell}$, the distance to it from $\mathbf{p}_0$ along the bisecting line. Shown in sand color are the angles β, with the proxy right triangles defined by the points $\mathbf{p}'_1$ and $\mathbf{p}'_2$ used in their calculation. (b) is an example of a face and sector requiring both interpolation and extrapolation of its function values

3.4. Interpolation & Extrapolation

With the goal of computing the weighted mean function values of the circle sectors $\mathbf{s}_i$ by interpolating and extrapolating the function values from their points towards the center of gravity of each sector $\mathbf{c}$, in Section 3.1 we calculated the global minimum edge length $\overline{\ell_{\min}}$, then in Section 3.2 we calculated the interior angle β; both of which are constant between the two halves of the circle sector. With that information, we can obtain the sector-wise constant ratio ζ, which will be used to scale the function

values at the points $\mathbf{p}_i$, to the interpolated or extrapolated values at the corners of the circumscribing isosceles triangle $\mathbf{p}'_i$, by applying the law of sines [39] and ascribing

$$\zeta := \frac{\overline{\ell_{\min}}}{\sin(\beta)} = \frac{|\mathbf{p}'_i - \mathbf{p}_0|}{\sin(\pi/2)} = |\mathbf{p}'_i - \mathbf{p}_0| \tag{3.8}$$

For the next few steps in the process of interpolating and extrapolating the function values, we must begin calculating for each half of the circular sector individually, because the points $\mathbf{p}_j$ and $\mathbf{p}_{j+1}$ are likely[25] at different distances from the center point $\mathbf{p}_0$. Therefore, while the index i will remain the index of the circle sector $\mathbf{s}_i$, we will now use the index j to denote the side of the bisecting line, defined by its point $\mathbf{p}_j$ or $\mathbf{p}_{j+1}$.

Next, by dividing the sector-wise constant ratio ζ by the distances to the original points ℓ_j and ℓ_{j+1}, we can determine the scalar values with which we can multiply the function values f_j and f_{j+1} in order to interpolate or extrapolate them to the corners of the circumscribing isosceles triangle at points $\mathbf{p}'_j$ and $\mathbf{p}'_{j+1}$. Therefore, these two scalar values are defined as

$$\tilde{\ell}_j := \frac{\zeta}{\ell_j} = \frac{\zeta}{|\mathbf{p}_j - \mathbf{p}_0|} \tag{3.9}$$

$$\tilde{\ell}_{j+1} := \frac{\zeta}{\ell_{j+1}} = \frac{\zeta}{|\mathbf{p}_{j+1} - \mathbf{p}_0|} \tag{3.10}$$

Then, by taking the function values[26] from the scalar field $\mathcal{F}$, and summing the products of the multiplications between the function values f_j, and f_{j+1} and their scalar values $\tilde{\ell}_j$ and $\tilde{\ell}_{j+1}$, plus the product of the multiplication between the function value at the central point f_0 and the complement of each side's scalar value, we can now interpolate and extrapolate the function values at $\mathbf{p}'_j$ and $\mathbf{p}'_{j+1}$ as

$$f'_j := f_0(1 - \tilde{\ell}_j) + f_j \tilde{\ell}_j \tag{3.11}$$

$$f'_{j+1} := f_0(1 - \tilde{\ell}_{j+1}) + f_{j+1}\tilde{\ell}_{j+1} \tag{3.12}$$

Figure 3.3 illustrates a 3D-projected, enhanced view of the circle sector $\mathbf{s}_1$, continuing with the example introduced in Figure 3.1. Drawn above the points $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ are the three function values f_0, f_1, and f_2 from the scalar field $\mathcal{F}$, with the heights indicative of each function value's magnitude. Also shown are the extrapolated function values f'_1, and f'_2 above the corners of the isosceles triangle $\mathbf{p}'_1$, and $\mathbf{p}'_2$, with the dotted lines illustrating the vectors on which the extrapolated values were calculated. Furthermore, this illustrates the process of interpolating the extrapolated function values towards each other, and then again towards the center of gravity, which is the topic covered in the next section.

[25] One need only to look at Figures 3.1 or 3.2 for an example of why that may be.

[26] The Fast One-Ring smoothing filter is agnostic to the meaning of information represented by the data stored as function values in scalar fields; therefore, it can similarly convolve any such data. However, because the filter was designed to only convolve scalar fields, any multi-dimensional data, such as RGB color, must be processed individually as independent scalar fields.

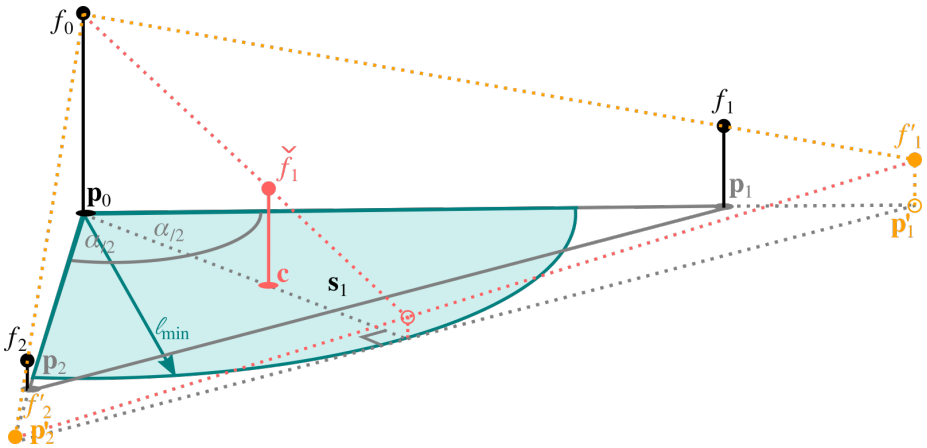


Figure 3.3. A 3D-projected, enhanced view of the circle sector $\mathbf{s}_1$, continuing with the example introduced in Figure 3.1. Drawn in black above the points $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ are the three function values f_0, f_1, and f_2 from the scalar field $\mathcal{F}$, with the heights indicative of each function value's magnitude. Drawn in sand color are the extrapolated function values f'_1, and f'_2 above the corners of the isosceles triangle $\mathbf{p}'_1$, and $\mathbf{p}'_2$, as well as the dotted lines illustrating the vectors on which the extrapolated values were calculated. Drawn in coral color at its interpolated value above the center of gravity $\mathbf{c}$ is the weighted mean function value $\check{f}_1$, as well as the dotted lines illustrating the interpolation from the extrapolated function values f'_1 and f'_2.

3.5. Weighted Means

Now that we have, in Section 3.3, calculated the distance to the center of gravity of each circle sector as $\check{\ell}$, and then in Section 3.4, computed the two function values interpolated or extrapolated to the corners of the isosceles triangle circumscribing the circle sector, f'_j and f'_{j+1}, we are poised to begin the computation of the weighted mean function values, first for each circle sector $\mathbf{s}_i$, then for the entire geodesic disc Ω_v, and finally iteratively convolving the Fast One-Ring smoothing filter for scalar fields on discrete manifolds over the entire mesh to generate the smoothed, scalar field of weighted mean function values $\mathcal{F}'$.

The next step then, is to calculate the weighted mean function value at the center of gravity $\mathbf{c}$, of a circle sector $\mathbf{s}_i$. That entails first interpolating the extrapolated function values towards each other, to the point where the tangent line intersects the arc of the circle sector, which is exactly the distance of $\overline{\ell_{\min}}$ away from $\mathbf{p}_0$. It is also always exactly half way between the two proxy points $\mathbf{p}'_j$ and $\mathbf{p}'_{j+1}$, so that interpolation always coincides with calculating the mean of the two function values. Therefore, by taking the equation for interpolation as was defined in Section 2.1.3.4, and used in Algorithms 3.11, and 3.12, then applying $\overline{\ell_{\min}}$ as the domain, and the difference between the original function value at the central point $\mathbf{p}_0$ and the mean of the interpolated and extrapolated function values as the range, we obtain the formula

$$\check{f} := f_0\,(1 - \check{\ell}) + \frac{(f'_j + f'_{j+1})\,\check{\ell}}{2} \tag{3.13}$$

which is used to calculate the weighted mean function value at the center of gravity $\check{f}$, and therefore represents the weighted mean function value over the entire circle sector.

Figure 3.3 illustrates a 3D-projected, enhanced view of the circle sector $\mathbf{s}_1$, continuing with the example introduced in Figure 3.1. The weighted mean function value $\check{f}_1$, which represents the entire circle sector, is drawn at its interpolated value above the center of gravity $\mathbf{c}$. The figure also shows the dotted lines illustrating the interpolation from the extrapolated function values f_1' and f_2', as well as the details pertaining to their own computations, as discussed in detail in Section 3.4.

Finally, we can multiply each weighted mean function value located at each circle sector's center of gravity $\check{f}_i$, by each sector's area A_i, to obtain the volume of function value over the entire circle sector. Then by dividing the sum of all those volumes by the total area of the geodesic disc Ω_v, we can compute

$$f_v' := \frac{\sum A_i \check{f}_i}{\sum A_i} \quad \forall i \in \{1, \ldots, |\mathcal{N}_v|\} \tag{3.14}$$

the one-ring weighted mean[27] function value for $\mathbf{p}_v$, which is the center point $\mathbf{p}_0$ of the neighborhood $\mathcal{N}_v$.

Figure 3.4 illustrates a 3D-projected view of the geodesic disc Ω, as introduced in Figure 3.1, with volumes of interpolated function value over each circle sector, which shall all be summed together, then divided by the total area of the geodesic disc in order to obtain the the one-ring weighted mean function value f_v' at point $\mathbf{p}_v$.

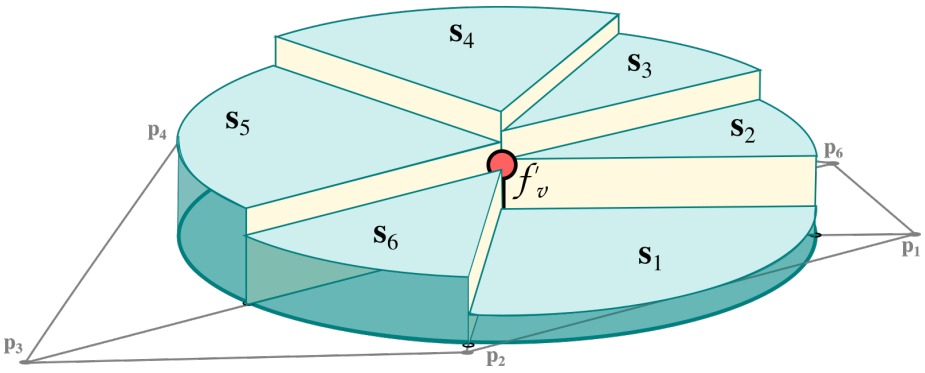


Figure 3.4. A 3D-projected view of the geodesic disc Ω, as introduced in Figure 3.1, with volumes of interpolated function values over each circle sector $\mathbf{s}_i$, which shall all be summed together, then divided by each sector's area A in order to obtain the the one-ring weighted mean function value f_v' at $\mathbf{p}_v$.

All that remains for the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, is to convolve the filter over the entire set of points comprising the mesh $\mathcal{P}$, then to collect the computed one-ring weighted mean function values in the set

[27] The Fast One-Ring smoothing filter can be modified to use the median operation, instead of the mean, by using all the equations except Equation 3.14 and then sorting the results of Equation 3.13. The details of which can be found in the original publication [27, s. 3.2], but as it was not implemented in GPGPU for this thesis, we exclude the details here.

$$\mathcal{F}' := \{\, f'_v \mid v \in \mathbb{N}, \text{ and } 1 \leq v \leq |\mathcal{P}| \,\} \tag{3.15}$$

which can be either be used in lieu of $\mathcal{F}$ in subsequent convolutions, or simply taken as the final product of all the processes described in this chapter.

As with any smoothing filter, one can convolve the Fast One-Ring smoothing filter for any number of iterations, thus producing a smoothing effect with increasing intensity in relation to the number of convolutions. But while there is no upper limit to the number of convolutions one may apply, the function values will all approach the global mean average function value [21, p. 299-331].

3.6. Summary

In this chapter we presented the mathematical grounding for the as-yet-unpublished, updated version of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, which since its original publication [27, s. 3.2], now utilizes the entire area of each sector $\mathbf{s}$ of the geodesic disc Ω centered at each point $\mathbf{p}_v$ in the mesh $\mathcal{M}$, in order to calculate the weighted average of all the function values in its one-ring neighborhood.

First, we illustrated in detail how one can calculate the global minimum edge length $\overline{\ell_{\min}}$, and for each circle sector in the one-ring neighborhood $\mathcal{N}$, the interior angles α and β, the area A, and the distance $\check{\ell}$ from the center point to the center of gravity $\mathbf{c}$. Next, we provided the equations for interpolating and extrapolating the three function values f_0, f_j, and f_{j+1} to the center of gravity, using the sector-wise constant ratio ζ in order to obtain the weighted mean function value $\tilde{f}$ for each sector. Finally, we computed the weighted mean function value f'_v at the central point $\mathbf{p}_v$, which is collected into the set $\mathcal{F}'$. Convolving this filter at each point in the mesh, with the scalar field of function values $\mathcal{F}$, for any number of iterations, thus produces a smoothing effect with increasing intensity in relation to the number of iterations.

In the next chapter, we will develop the serial algorithms required for implementing the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, to be utilized as the foundation for the parallel algorithms described in detail in Chapter 5.

Chapter 4

Fast One-Ring Smoothing: Serial Algorithms

In the previous chapter we presented the mathematical grounding for an improved version of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, as it is currently implemented within the GigaMesh framework. Now in this chapter, we will use mathematical pseudo-code to combine all of the equations from Chapter 3, Equations 3.1 through 3.14, into a serial algorithm, with the goal of preparing the basis from which we will subsequently design the parallel algorithm presented in Chapter 6.

4.1. An Algorithm in Three Parts

The serial algorithm defined in this chapter requires that the membership of all one-ring neighborhoods must first be discovered, before one is ever able to begin convolving the filter, because the total efficiency of the algorithm relies on iterating over the family of sets of neighborhoods $\mathcal{N}$, as will be made clear in Sections 4.3, and 4.4, while the underlying data structure of 3D-data typically only stores a mesh $\mathcal{M}$ as the family of sets comprised of the two sets $\mathcal{P}$ of points, and $\mathcal{T}$ of triangular faces, remaining completely ignorant of the adjacency between faces or even the concept of neighborhoods.

Also critical to the total efficiency of the filter, is the pre-calculation of the edge lengths between all pairs of adjacent points, whose magnitudes never change regardless of the number of convolutions being applied. This importance stems from the way all edge lengths are used in computation during the loop for convolving the filter, which we will refer to as "the principle loop" going forward, at least five times per convolution, and ten times per convolution for the majority of points, which are non-border pairs whose relationships are duplicated in adjacent neighborhoods.

Therefore, in order to maximize the efficiency of the Fast One-Ring smoothing filter, it is crucial to split the algorithm into three distinct parts: first building neighborhoods, then calculating edge lengths, and finally convolving the filter, all of this while simultaneously storing the results of the first two parts so that they may be used during the iterative convolutions of the third part; the result being a massive increase in efficiency by greatly reducing the number of operations-per-convolution required by the filter.

4.2. Building Neighborhoods

Before convolving the Fast One-Ring smoothing filter, one must initially discover all the points in the mesh $\mathbf{p}_i$ which comprise each neighborhood $\mathcal{N}_v$, then store those connections in the family of sets $\mathcal{N}$. That is the purpose of Algorithm 3. Despite the fact that building $\mathcal{N}$ outside of the principle loop adds an additional $6 \cdot |\mathcal{T}|$ operations to the total, Algorithm 4 becomes able to exploit the explicit connections stored in $\mathcal{N}$, in order to vastly reduce its complexity from $|\mathcal{P}|^{|\mathcal{T}|}$ down to only $|\mathcal{P}|^{\bar{n}}$, where $\bar{n}$ is the average size of all neighborhoods, which with acquired 3D-data, will typically evaluate to approximately 6. Also, as we will see in Section 4.4, when τ is the user-defined number of convolutions to perform, the complexity of the principle loop can be meaningfully reduced to only $\tau^{|\mathcal{P}|^{2 \cdot \bar{n}}}$, down from the $\tau^{|\mathcal{P}|^{(2 \cdot |\mathcal{T}|)}}$ that would have been necessary, had the neighborhoods not already been discovered and the procedure been otherwise required to discover the members of $\mathcal{N}_v$ in each convolution.

Figure 4.1 describes a very simple mesh consisting of just two faces and four points, similar to that which is illustrated in Figure 2.5. The arrows represent the union operation between a point and a neighborhood, and are colored to match the face from whence the point had come. It should be noted that the two pairs of arrows pointing from $\mathbf{p}_2$ to $\mathcal{N}_3$, and $\mathbf{p}_3$ to $\mathcal{N}_2$, indicate that both of these union operations occur twice, one originating from each faces, but because of the uniqueness property of sets, the duplicated operations are wholly inconsequential to the the final membership of either neighborhood.

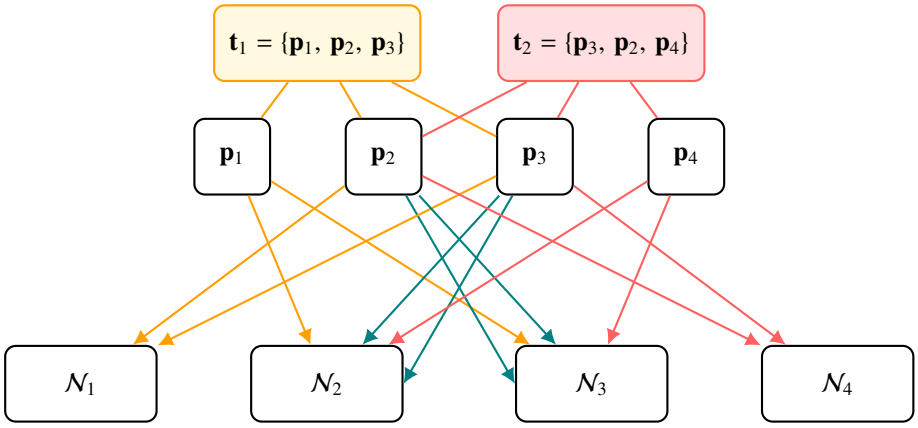


Figure 4.1. A very simple mesh consisting of just two faces and four points, similar to that which is illustrated in Figure 2.5. The face $\mathbf{t}_1$ is in sand color, and $\mathbf{t}_2$ is in coral color. The arrows represent the union operation between a point and a neighborhood are colored to match the face from whence the point had come. The two pairs of arrows, pointing from $\mathbf{p}_2$ to $\mathcal{N}_3$ and $\mathbf{p}_3$ to $\mathcal{N}_2$, are teal colored to highlight the fact that these union operations occur twice.

Also notice in Figure 4.1, that the union operation is performed exactly twice per point per face, which is once each between the neighborhood of the center point and a neighboring point, for a total of six times per face. For example, the triangular face $\mathbf{t}_2$ contains the point $\mathbf{p}_4$, therefore the union operation is performed on $\mathcal{N}_4$ at least two times, once for each of the other members of point $\mathbf{t}_2$, $\mathbf{p}_2$, and $\mathbf{p}_3$. This realization will influence the design of the parallel variant of the algorithm, as described in Section 5.1.1.

In Algorithm 3, the function *serialBuildNeighborhoods* describes iterating over every triangular face $\mathbf{t}$ in $\mathcal{T}$. Then for each face's three corner points, the union operation is perfomed between the neighborhood centered at the point, and the other two points which are adjacent to the central point. After processing every face, the result becomes a fully-populated family of sets $\mathcal{N}$, storing the connections between every neighbor of every neighborhood in the mesh $\mathcal{M}$.

Algorithm 3: Serial algorithm for building the family of sets $\mathcal{N}$, from all discovered members of each neighborhood in the mesh

Input : the set of all triangular faces $\mathcal{T}$
Output: the family of sets of discovered neighborhoods $\mathcal{N}$

```
1 Function serialBuildNeighborhoods(T)
2     for t ∈ T :                                   /* t = {p_a, p_b, p_c} */
3         union(N, p_a, p_b, p_c)
4         union(N, p_b, p_a, p_c)
5         union(N, p_c, p_a, p_b)

6 Function union(N, a, b, c)
7     N_a ← N_a ∪ {b, c}
```

The function *union* is defined separately in Algorithm 3 for two reasons. The first reason is that when separated, it becomes very clear that the union operation behaves similarly for all three permutations[28] of corners, and indeed, it is only the order of the indices which changes. The second reason is that this signature will match more closely that of its parallel variant, to be defined in Algorithm 6, which will require the union operation to remain separate.

4.3. Calculating Edge Lengths

Having now built in the previous section the family of sets of neighborhoods $\mathcal{N}$, we can advance to the next step, Algorithm 8, which iterates over each pair of neighbors comprising $\mathcal{N}$, with the goal of building a set of pre-calculated edge lengths $\mathcal{E}$, as well as determining the global minimum edge length $\overline{\ell_{\min}}$; both being essential parameters of Algorithm 5.

As shown in Equation 2.19, the calculation of an edge's length requires taking the L2-norm of the difference between two points, which itself involves using the square root operation. In modern software, the square root operation is performed by computing "Newton's Iteration", or "Newton's method", which is essentially multiple iterations of the so-called, "recurrence equation" [41]. The impact for the Fast One-Ring smoothing filter for scalar fields on discrete manifolds is that the computation of a square root typically takes many more compute cycles than any other binary or unary operation, thus taking more time to complete overall. In fact, because of the slowness of the square root operation, computing the L2-norm in order to calculate an edge's length is

[28] In general, a set of three numbers has six permutations, however, here only the first index is important, and the order of the second two parameters are arbitrary due to the commutative property of the union operation, as shown in Equation 2.7, resulting in only 3 distinct possibilities.

empirically the most costly operation performed by the filter. Therefore, it is imperative that we pay special attention to avoid unnecessary instructions to calculate an edge's length. For that reason, we define the symbol ℓ_* to represent the calculation of an edge's length using "Newton's Iteration", so that we can draw focus to its importance while remaining concise.

In Algorithm 4, the function *serialCalculateEdgeLengths* iterates over a set of nested loops which considers each neighbor $\mathbf{p}_i$ of each neighborhood $\mathcal{N}_v$, in order to calculate the edge lengths between the center point $\mathbf{p}_v$ and its neighboring points $\mathbf{p}_i$. The result is then stored in the set $\mathcal{E}$ using the pair of indices v, i to reflect the structure of the family of sets of neighborhoods $\mathcal{N}$. Finally, in each iteration, the value of the global minimum edge length $\overline{\ell_{\min}}$ is updated to become the minimum between $\mathcal{E}_{v,i}$ and the current $\overline{\ell_{\min}}$, ensuring that at the conclusion of the procedure, no other edge length in $\mathcal{E}$ will be shorter than $\overline{\ell_{\min}}$.

Algorithm 4: Serial algorithm for calculating all the edge lengths between each pair of adjacent points in the mesh

Input : the set of all points $\mathcal{P}$,
the family of sets of discovered neighborhoods $\mathcal{N}$
Output: the set of pre-calculated edge lengths $\mathcal{E}$,
the global minimum edge length $\overline{\ell_{\min}}$

```
1 Function serialCalculateEdgeLengths(P, N)
2     for p_v ∈ P :
3         for p_i ∈ N_v :
4             E_{v,i} ← |p_i − p_v|                    /* This is ℓ_* as in Eq: 3.1 */
5             ℓ_min ← min{ℓ_min, E_{v,i}}              /* Eq: 3.2 */
```

Given the substantial impact of computing ℓ_*, and the enormous number[29] of times an edge length is required in the computation of the weighted mean function values per convolution of the filter, pre-calculating the set of all edge lengths $\mathcal{E}$ outside of the principle loop becomes critical to the overall efficiency of the algorithm, despite the fact that it requires ℓ_* to be calculated and stored $|\mathcal{P}|^{\bar{n}}$ times.

This pre-calculation is of paramount importance for two reasons. The first reason is that without pre-calcuating every edge length, it would be otherwise impossible to calculate the global minimum edge length $\overline{\ell_{\min}}$, which is used $|\mathcal{P}|^{4\,\bar{n}}$ times in every convolution of the filter. The second reason is that by recording the results of every ℓ_* calculation in the set $\mathcal{E}$, we are then able to completely exclude any further calculations of ℓ_* from the principle loop, and as can be seen in Algorithm 5, that reduces the total count of ℓ_* calculations performed by the filter from the $\tau^{|\mathcal{P}|^{3\,\bar{n}}}$ had the procedure been required to calculate an edge length each time it was used during computation, down to only the initial $1 \cdot |\mathcal{P}|^{\bar{n}}$ pre-calculations, thus becoming completely independent of τ and significantly more efficient overall.

[29] "enormous" $\approx 2 \cdot \left(|\mathcal{P}|^{5\,\bar{n}}\right)$, because with increasing mesh densities, as the ratio of counts of faces to counts of points converges to two, the ratio of border to non-border edge lengths diminishes. This is described in detail in the Appendix A.

4.4. Convolving the Filter

Having discussed in Section 4.2 the discovery and subsequent construction of the set of every one-ring neighborhood $\mathcal{N}$, then in Section 4.3, the set of pre-calculations of every edge length $\mathcal{E}$, in this section, we present the third and final part of the serial algorithm for convolving the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, describing the remaining steps required to convolve the filter over a mesh.

In Algorithm 5, the function *convolveFilter* outlines how the convolutions of the Fast One-Ring smoothing filter are performed for a user-defined number of times τ, by convolving over each and every point $\mathbf{p}_v$ in the set $\mathcal{P}$. Then at each point, each neighboring point $\mathbf{p}_i$ in the one-ring neighborhood $\mathcal{N}_v$ is examined in order to calculate the weighted mean function value $\tilde{f}$, as described in detail in Sections 3.2 - 3.5, at the center of gravity of the circle sector defined by $\mathbf{p}_v$ and $\mathbf{p}_i$. Next, the weighted mean function value f'_v is evaluated as the average of all the weighted mean function values from each circle sector comprising the geodesic disc Ω_v, and is stored in set $\mathcal{F}'$, before the filter finally progresses to the next point in the mesh and its corresponding one-ring neighborhood. Afterwards, one may efficiently convolve the filter, for as many convolutions as are required to achieve the desired smoothing effect.

While the performance of Algorithm 5 is much improved by the neighborhood building and pre-calculations performed in Algorithms 3 and 4, its strictly-serial design prevents it from scaling in performance appropriately for the sizes of real-world, acquired 3D-data, performing especially sluggishly[31] for high numbers of convolutions on meshes with large amounts of triangulated points; the very targets for which the Fast One-Ring smoothing filter for scalar fields on discrete manifolds is primarily intended.

4.5. Summary

In this chapter, we presented the serial algorithms for implementing the improved version of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, as implemented within the GigaMesh framework. In Section 4.1, we discussed the motivation for separating the algorithm into three parts, citing the substantial improvements to the over all efficiency of the filter. In Section 4.2, the considerations behind building a family of sets of neighborhoods are weighed, before presenting the function *serialBuildNeighborhoods* in Algorithm 3. Then in Section 4.3, the significance of ℓ_* and the L2-norm calcuation is explained in context with modern implementations of the square root operation, before presenting function *serialCalculateEdgeLengths* in Algorithm 4. Finally, in Section 4.4, Algorithm 5 is presented with the function *serialConvolveFilter*, which implements the principle loop for convolving the filter over a mesh, for as many

[30] In the source code which had existed in GigaMesh, the scalar value $\overline{\ell_{\min}}$ had accidentally been removed from the implmentation of line 8, which constituted an error in the new weighting mechanism. However, as its absence equates to fixing the global minimum edge length to be equal to one, the result was an inaccurate calculation of length to the center of gravity in each sector, impacting the final weighted mean function value of every point in the convolution. An experiment was conduncted to examine the effect on the filter response cause by this error, and the conclusion was that while the filter response was indeed different, it was typically very similar to the correct value. Also, when convolving the filter until convergence, the resulting scaler field of function values output by the erroneous algorithm and the corrected version were exactly the same.

[31] Among our experiments, the maximum time for the serial algorithm to complete was obtained when convolving a mesh comprised of $\approx 9 \times 10^6$ points for 3×10^3 convolutions. This experiment required ≈1,108 minutes or ≈18.5 hours to complete.

Algorithm 5: Serial algorithm for convolving the Fast One-Ring smoothing filter for scalar fields on discrete manifolds

Input : the set of all points $\mathcal{P}$,
the family of sets of discovered neighborhoods $\mathcal{N}$,
the set of pre-calculated edge lengths $\mathcal{E}$,
the global minimum edge length $\overline{\ell_{\min}}$,
the set of function values $\mathcal{F}$,
the user-defined number of convolutions τ

Output: the set of one-ring weighted mean function values $\mathcal{F}'$

Function *serialConvolveFilter($\mathcal{P}$, $\mathcal{N}$, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathcal{F}$, τ)*
  for $t \leftarrow 1$ **to** τ **:**
    for $\mathbf{p}_v \in \mathcal{P}$ **:**
      for $\mathbf{p}_i \in \mathcal{N}_v$ **:**
        $\alpha \leftarrow cos^{-1}\left(\frac{\mathcal{E}_c^2 + \mathcal{E}_b^2 - \mathcal{E}_a^2}{2 \cdot \mathcal{E}_c \cdot \mathcal{E}_b}\right)$ /* Eq: 3.4 */
        $\beta \leftarrow (\pi - \alpha) \,/\, 2$ /* Eq: 3.5 */
        $A \leftarrow \left(\overline{\ell_{\min}}\right)^2 \cdot \alpha \,/\, 2$ /* Eq: 3.6 */
        $\check{\ell} \leftarrow (4 \cdot \overline{\ell_{\min}} \cdot \sin(\alpha \,/\, 2)) \,/\, 3\,\alpha$ /* Eq: 3.7 [30] */
        $\zeta \leftarrow \overline{\ell_{\min}} \,/\, \sin(\beta)$ /* Eq: 3.8 */
        for $j \in 1, 2$ **:**
          $\tilde{\ell}_j \leftarrow \zeta \,/\, \mathcal{E}_j$ /* Eq: 3.9, 3.10 */
          $f'_j \leftarrow f_0 \cdot (1 - \tilde{\ell}_j) + f_j \cdot \tilde{\ell}_j$ /* Eq: 3.11, 3.12 */
        $\check{f} \leftarrow f_0 \cdot (1 - \check{\ell}) + ((f'_1 + f'_2) \cdot \check{\ell}) \,/\, 2$ /* Eq: 3.13 */
        $\tilde{f}_v \leftarrow \tilde{f}_v + A \cdot \check{f}$ /* Eq: 3.14 */
        $\tilde{A}_v \leftarrow \tilde{A}_v + A$ /* Eq: 3.14 */
      $f'_v \leftarrow \tilde{f}_v \,/\, \tilde{A}_v$ /* Eq: 3.14 */
    $\mathcal{F}' \leftarrow \left\{f'_1, \ldots, f'_{|\mathcal{P}|}\right\}$
    $\mathcal{F} \leftarrow \mathcal{F}'$ /* smooth newest values every convolution */

convolutions as are required to achieve the desired smoothing effect.

Unfortunately, the Fast One-Ring smoothing filter for scalar fields on discrete manifolds is still entirely serial in design; therefore, in the next section, we will endeavor to explore this as-yet-unpublished algorithm in order to discover any manifestations of independent procedures worthy of exploiting with parallel processing, with the goal of improving the overall performance and scalability of the filter when it is implemented on a system capable of parallel computation.

Chapter 5

Fast One-Ring Smoothing: Parallel Algorithms

In the previous two chapters, we presented the mathematical grounding and serial algorithms for an improved version of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, as it is currently implemented within the GigaMesh framework, and while it has improved accuracy in regards to the method of weighting the mean function values, it is still entirely serial in design. This means that, unfortunately, its performance suffers greatly under the complexity of modern mesh sizes, which with the current high resolution scanners in use, can commonly be comprised of millions, or tens of millions, of points [27, 25,144] [14, 4].

In this chapter, we will explore the serial version of this as-yet-unpublished algorithm, in order to negotiate any instances of control or data dependency. Upon the discovery of manifestations of independent procedures worthy of exploiting with parallel processing, we will augment the design of the parallel variant with the goal of improving the performance of the filter when implemented on a SIMD system capable of parallel computation.

Following the pattern set by the serial algorithm in Chapter 4, the parallel algorithm for the Fast One-Ring smoothing filter for scalar fields on discrete manifolds also has three main parts. In contrast however, the parallel variant of each of the three serial parts are further split into kernels, in order to delegate blocks of instructions to separate threads of execution, while protecting critical sections, and avoiding data and control dependencies, so that the threads can be safely executed in parallel.

Each of the following three sections will focus on producing the parallel variant of one of the three parts of the serial algorithm which was presented in Chapter 4. This will be accomplished by first analyzing the serial algorithm through the lens of parallel programing, next defining the strategy behind the design of the parallel variant, and finally providing the pseudo code definitions to implement that part of the algorithm.

5.1. Build Neighborhoods in Parallel

The sole purpose of Algorithm 3, as discussed in detail in Section 4.2, is to explore every triangular face in the set $\mathcal{T}$ in order to generate a family of sets of neighborhoods $\mathcal{N}$, so that subsequent algorithms can recall and iterate over that set and negate the computational costs incurred by searching the entire mesh each time the membership of a neighborhood must be known. In addition to building $\mathcal{N}$, because of the nondeterministic nature of the sum of the cardinalities of each neighborhood of $\mathcal{N}$, the parallel variant of the *buildNeighborhoods* procedure must also provide a census, a total count of all neighbors in all neighborhoods, so that the next two parts of the parallel variant of the Fast One-Ring smoothing filter algorithm can accurately predict work loads, and evenly delegate portions of the work to independent threads of execution.

5.1.1. Analysis of Serial Algorithm 3: Build Neighborhoods

In this section, we analyze the Serial Algorithm for Building Neighborhoods as presented in Section 4.2, through the lens of parallel programing. The strategy we employ involves reading the serial algorithm line by line, building a dependency graph, and as opportunities for exploiting parallelism are discovered, divising a strategy for maximizing efficiency is devised.

Starting in line 1 of Algorithm 3, we read the declaration of the function *serialBuildNeighborhoods* requiring the entire set of triangular faces $\mathcal{T}$ as an input. Then already in line 2, we encounter a loop iterating over each face $\mathbf{t}$ in $\mathcal{T}$. In general, concurrency found in the structure of a loop may be completely exploited by a parallel implementation of the same procedure, but only in the absence of loop-carried dependence. So, in order to determine if the iterations of this loop may be computed in parallel, an analysis of every operation in the loop block is required.

Figure 5.1 illustrates the data dependencies inherent to each iteration of the loop over the faces $\mathbf{t}$ in $\mathcal{T}$, as found in lines 2 - 5 of Algorithm 3.

As denoted by the first three teal-colored lines, the first operations are to read the three points, $\mathbf{p}_a$, $\mathbf{p}_b$, and $\mathbf{p}_c$, from the triangular face $\mathbf{t}$, stored is the set $\mathcal{T}$ in static memory, which will never be modified by this, or any other procedure. Also, despite the fact that faces can be adjacent to one another, each face is distinctly defined in the set $\mathcal{T}$, therefore, each face $\mathbf{t}$ can be considered independent of each other face. Thus, this first instruction of the loop block is totally free from any control or data dependencies. The next line in the algorithm is a complex instruction, though, so it must be analyzed in parts.

The first part of the instruction on line 3 uses the point $\mathbf{p}_a$ as a reference to read the state of $\mathcal{N}_a$ from the set $\mathcal{N}$ in volatile memory, as denoted as teal and coral-colored lines respectively. $\mathcal{N}_a$ is stored in volatile memory, which may potentially be modified by other threads, because it is not known a priori which thread will discover the points that belong to that neighborhood, thus it must be accessible to any and all threads. Next, also drawn in coral color, are the two union operations performed in succession[32] between $\mathcal{N}_a$ and the point $\mathbf{p}_b$, then $\mathbf{p}_c$. While these two operations are themselves independent, being performed only on values currently immutable by other procedures, they do rely on having already read the status of $\mathcal{N}_a$, which indeed does have its own

[32] A possibility enabled by the commutative property of the union operation, as shown in Equation 2.7.

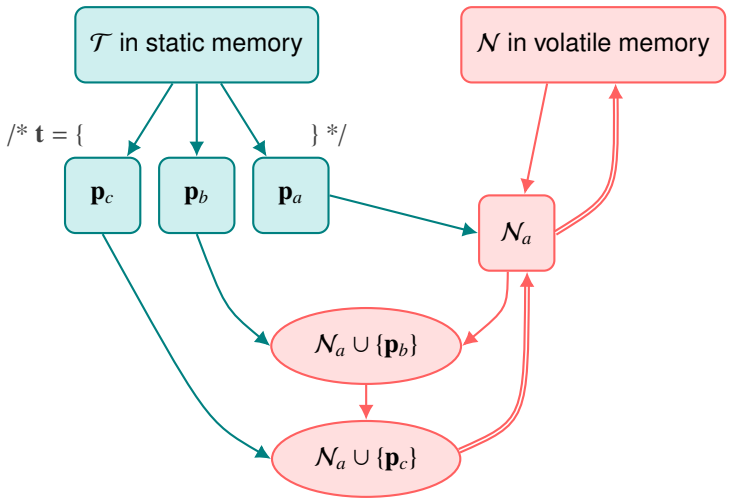


Figure 5.1. An illustration of the data dependencies found in Algorithm 3. Square nodes indicate a value stored in memory and oval nodes indicate operations. Single arrows indicate reading values and double arrows indicate writing values to memory. Static memory and the values and operations which are independent and free of data dependencies are in teal color. Volatile memory and the values stored there, which may be modified at anytime during a procedure, and the operations which rely on those volatile values, are in coral color.

data dependence, hence the indicative coloring. Finally, illustrated as a coral colored double arrow, the updated set $\mathcal{N}_a$ is stored back into $\mathcal{N}$, replacing the original set in volatile memory.

Both reading from and writing to $\mathcal{N}_a$ in volatile memory constitutes a critical section in the algorithm, which must be accounted for in the design of the parallel variant of this part of the algorithm. Furthermore, both dependencies lie in a single path of dependence, which means that the entire group of operations must be protected by a locking mechanism as described in Section 2.3.5.2.

Fortunately, because both dependencies revolve around the same set $\mathcal{N}_a$, instead of implementing a global mutex to protect the critical section in all threads computing weighted mean function values, it will be sufficient to design the algorithm with a set of mutexes, where each mutex only protects a neighborhood's portion of the total critical sections being processed in parallel.

The significance of this design choice comes from how typical of acquired 3D-data, the ratio of points in $\mathcal{P}$ to the average neighborhood size approaches $|\mathcal{P}|$, appreciable with even moderately sized meshes, so instead of all processors, which can count in the thousands, blocking in wait of entering the same critical section for all neighbors of all points, which can count in the tens of millions, each processor must now only monitor the mutex of the neighborhood for which it is working, which typically will include only about 6 neighbors.

Lines 4 and 5 behave similarly to line 3, except that they concern other neighborhoods, which in turn, need their own mutexes. Fortunately, as is modeled with a simple mesh in Figure 4.1, the union operation is performed exactly twice per point per face; that is, once each between the neighborhood of the center point and a neighboring point,

for a total of six times per face. By exploiting this pattern and executing the two union operations sequentially within a single mutex, one can mitigate exactly half of the possible collisions in the *buildNeighborhoods* procedure.

5.1.2. Parallel Variant of Build Neighborhoods

Algorithm 6 defines the structure and procedures required to implement a parallel variant of the function *serialBuildNeighborhoods*, while ensuring correctness by using a set of mutexes to guard critical sections. The main function *parallelBuildNeighborhoods* requires only the number of available processors ρ, the set of all triangular faces $\mathcal{T}$, and the cardinality of the set of points $|\mathcal{P}|$ to complete its task.

Initially, the portion of the total work load, hence forth referred to as "the stride", is calculated in line 8. Then each individual processor will be instructed to compute its own stride, equal to the problem size divided by the number of available processors. Here, the unit of work being only the three points of the triangular face $\mathbf{t}$.

Next, a set of mutexes are created in line 3, containing an equal amount of elements as the cardinality of points in $\mathcal{P}$, to be used individually for each neighborhood. Then begins the loop to spawn a "build" thread for a quarter of the available processors in the system, with the scalar four being a result of each build thread's need to spawn three additional "union" threads. Then all the working threads must synchronize first, before the family of sets of neighborhoods $\mathcal{N}$ can finally be realized.

In line 7, we read that along with access to the set of triangular faces $\mathcal{T}$ and set of mutexes $\mathfrak{M}$, each build thread in Algorithm 6 also requires an index Π, which it uses along with the stride *sigma*, to calculate the lower and upper boundaries, $\check{\sigma}$ and $\hat{\sigma}$, for the portion of the work load on which it should compute. Next, it iterates over each face $\mathbf{t}$ within those stride boundaries, and spawns three union threads; one for each of the three points comprising the corners of the current triangular face.

Line 14 shows that each union thread not only requires the family of sets of neighborhoods $\mathcal{N}$ and the set of mutexes $\mathfrak{M}$, but also the point which it will consider to be central, and the two points which are the central point's neighbors. With those inputs, each union thread then attempts to lock the mutex which shares the index of both the central point and its corresponding neighborhood, blocking if necessary, in order to safely perform the union operation between the currently known set of neighbors for the central point, and the set of its two newly discovered neighbors. Finally, it saves the updated neighborhood to volatile shared memory while still protected by the mutex, before unlocking the mutex.

5.1.3. Parallel Recursive Census Neighborhoods

The motivation behind this section and Algorithm 7 is that the next two parts of the parallel algorithm for the Fast One-Ring smoothing filter for scalar fields on discrete manifolds perform individual computations for each neighbor in the family of sets of neighborhoods $\mathcal{N}$, and so require a total count of all neighbors in all neighborhoods in order to accurately predict and evenly delegate portions of the work load to each independent thread of execution.

Additionally, because of the nondeterministic nature of the cardinality of neighborhoods in acquired 3D-data, therefore also the total sum of all cardinalities of neighbor-

Algorithm 6: Parallel algorithm for building the family of sets of all members of each neighborhood discovered in the mesh

Input : the number of available processors ρ,
the set of all triangular faces $\mathcal{T}$,
the cardinality of the set of points $|\mathcal{P}|$
Output: the family of sets of discovered neighborhoods $\mathcal{N}$

```
1  Function parallelBuildNeighborhoods(ρ, T, |P|)
2      σ ← |T| / ρ                                   /* assuming an integer quotient */
3      M ← {μ_1, ..., μ_|P|}
4      for Π ← 1 to ρ / 4 :
5          ~build(Π, σ, M, T)
6      synchronizeThreads()

7  Kernel build(Π, σ, M, T)
8      σ̌ ← (Π − 1) σ + 1                             /* works through its stride */
9      σ̂ ← Π σ
10     for t ∈ {t_σ̌, ..., t_σ̂} :                      /* t = {p_a, p_b, p_c} */
11         ~safeUnion(M, N, p_a, p_b, p_c)
12         ~safeUnion(M, N, p_b, p_a, p_c)
13         ~safeUnion(M, N, p_c, p_a, p_b)

14 Kernel safeUnion(M, N, a, b, c)
15     lock(μ_a)
16     N_a ← N_a ∪ {b, c}
17     unlock(μ_a)
```

hoods in $\mathcal{N}$, the parallel variant of the function *serialBuildNeighborhoods* must also provide a census of total membership, defined as

$$\hat{n} := \sum_{v=1}^{|\mathcal{N}|} |\mathcal{N}_v| \tag{5.1}$$

While a sum of all cardinalities could be accumulated by each thread in Algorithm 6, unlike in the serial version, it would very inefficient[33] due to the added overhead of collisions with the locking mechanism required to protect the shared value of the total sum. Alternatively, algorithm 7 only requires as inputs, the family of sets of neighborhoods $\mathcal{N}$, and the number of processors available in the system, but is highly parallel and performs at the quick rate of $O(log_2(n))$. Therefore, when using an SIMD system with multiple processors, calculating the census of the neighborhoods of $\mathcal{N}$ in isolation after the family of sets has been built is a much more efficient alternative than calculating the sum implicitly while building $\mathcal{N}$.

[33] It is possible that in some programming languages, the underlying data structure may automatically keep a record of the size of membership in $\mathcal{N}$. In that case, one would simply ignore Algorithm 7 and instead assign that generated value to $\hat{n}$.

As Algorithm 7 is a recursive algorithm, it is best described by dividing it into its three distinct parts. The first part, "the termination clause", only executes when the cardinality of the current subset of $\mathcal{N}$ is two or less[34], then finally returning the sum[35] of the members as $\hat{n}$. In all other cases, when the cardinality of the current subset of $\mathcal{N}$ is greater than two, the second part, "working towards the termination state", begins.

The second part of this recursive strategy describes summing in parallel the cardinality or value of each adjacent pair of members in the current subset of $\mathcal{N}$, then saving the sum of each addition in a new ordered subset of integers $\widetilde{\mathcal{N}}$, which will have half the cardinality of the current subset. To that end, the stride is calculated using one half of the cardinality of $\mathcal{N}$ as the total work load to be divided among all of the available processors. Next, a new thread is spawned for each processor in order to execute the kernel *parallelSum*, to process each portion of the work load in parallel.

The kernel *parallelSum* requires as inputs: a process index Π, the stride *sigma*, and the current subset of neighborhood cardinalities being processed[36]. First, it calculates the lower and upper boundaries, $\check{\sigma}$ and $\hat{\sigma}$, for the portion of the work load on which it should compute. Next, it iterates over every other index within the stride boundries in order to return the sum of the cardinalities of each even-odd pair of neighborhoods. While it may appear that this is an accumulation threatening a race-condition, because each sum is stored as its own unique value to be used in subsequent additions, albeit in the same set, there is no need to protect these operations with locking mechanisms, thus contributing greatly to the overall efficiency of the algorithm.

In the third and final part, "the recursive call", the function *parallelRecursiveCensus-Neighborhoods* must first wait for all the threads executing instances of *parallelSum* to synchronize before calling itself, using the new subset $\widetilde{\mathcal{N}}$ as a parameter instead of the original $\mathcal{N}$. In each iteration, every available processor is utilized and the cardinality of $\mathcal{N}$ is reduced by half, quickly approaching the termination clause, while requiring no locking mechanisms. Additionally, even without memory recycling, it only requires slightly less than twice the total memory required than storing $\mathcal{N}$ alone.

5.2. Calculating Edge Lengths in Parallel

The goal of this section is to produce the parallel variant of Algorithm 4, as presented in Chapter 4, which has the goal of building a set of pre-calculated edge lengths $\mathcal{E}$, as well as determining the global minimum edge length $\overline{\ell_{\min}}$; both essential parameters of Algorithm 5. First, we start by analyzing the serial algorithm through the lens of parallel programing, to be followed by defining the strategy behind the design of the parallel variant, and finally providing the pseudo code definitions to implement Algorithm 8.

[34] This conditional operator allows for unit subsets of only a single member, which is possible anytime one processes a set with a cardinality other than one of the positive powers of two, {2, 4, 8,. . . , 1048576, . . . }, so in a majority of case.

[35] This ignores the trivial case where the cardinality of the full family of sets $\mathcal{N}$ is less than three, thus never being processed by the *parallelSum* kernel. If that is required, only a simple modification, similar to the conditional statement found in line 13, must be made.

[36] or if this is the first call, the family of sets of neighborhoods $|\mathcal{N}|$.

[37] It is important to notice in line 13, the distinction between the family of sets of neighborhoods $\mathcal{N}$ in the first call of *parallelRecursiveCensusNeighborhoods*, and the set of integers accumulating in the subset $\widetilde{\mathcal{N}}$ in subsequent calls. Also, non-existent neighborhoods should be treated as having a cardinality of zero.

Algorithm 7: Parallel algorithm for recursively counting a census of all neighbors in all neighborhoods

Input : the number of available processors ρ,
the family of sets of neighborhoods $\mathcal{N}$
Output: the count of all neighbors in all neighborhoods $\hat{n}$

1 **Function** *parallelRecursiveCensusNeighborhoods(ρ, $\mathcal{N}$)*
2 **if** $|\mathcal{N}| \leq 2$ **then**
3 $\hat{n} \leftarrow \sum_{i=1}^{|\mathcal{N}|} \mathcal{N}_i$
  else
4 $\sigma \leftarrow |\mathcal{N}| \,/\, (2\,\rho)$
5 **for** $\Pi \in \{1, \ldots, \rho\}$ **:**
6 *~parallelSum(Π, σ, $\mathcal{N}$)*
7 *synchronizeThreads()*
8 *parallelRecursiveCensusNeighborhoods(ρ, $\widetilde{\mathcal{N}}$)*

9 **Kernel** *parallelSum(Π, σ, $\mathcal{N}$)*
10 $\check{\sigma} \leftarrow (\Pi - 1)\,\sigma + 1$
11 $\hat{\sigma} \leftarrow 2\,\Pi\,\sigma$
12 **for** $v \in \{\check{\sigma}, \check{\sigma}{+}2, \check{\sigma}{+}4, \ldots, \hat{\sigma}\}$ **:**
13 **if** $\mathcal{N}$ *is a family of sets* **then** /* only occurs in first call[37] */
14 $\widetilde{\mathcal{N}_v} \leftarrow |\mathcal{N}_v| + |\mathcal{N}_{v+1}|$
  else
15 $\widetilde{\mathcal{N}_v} \leftarrow \mathcal{N}_v + \mathcal{N}_{v+1}$

5.2.1. Analysis of Serial Algorithm 4: Calculate Edge Lengths

In this section, we analyze the Serial Algorithm for Calculating Edge Lengths as presented in Section 4.3, through the lens of parallel programing. The technique we employ involves reading the serial algorithm line by line, building a dependency graph, and as opportunities for exploiting parallelism are discovered, devising a strategy for maximizing efficiency.

Starting in line 1 of Algorithm 4, we read the declaration of the function *serialCalculateEdgeLengths* requiring the set of points $\mathcal{P}$, as well as the family of sets of neighborhoods $\mathcal{N}$ as inputs. Then, in line 2, we encounter a loop over each point $\mathbf{p}_v$ in $\mathcal{P}$. In order to determine if the iterations of the loop may instead be computed in parallel, an analysis of every operation in the loop block for loop-carried dependencies is required, which in this case, includes lines 3- 5.

The first operation in the loop block, line 3, starts another loop over each point $\mathbf{p}_i$ in the neighborhood $\mathcal{N}_v$. Because there are no operations in the first loop which starts on line 2, that are not also in the nested loop starting at line 3, we may consider them together as a single, multi-dimensional loop; one which iterates over the entirety of the unpredictable dimensions of the nondeterministic family of sets of neighborhoods $\mathcal{N}$. We will see this pattern again in Algorithm 9, and indeed, one can unroll both loops together in order to compute the loop block in parallel, by utilizing the total count of

neighbors and the average size of all neighborhoods, that is, barring any occurrences of loop-carried dependencies.

As can be clearly seen in Figure 2.4, the cardinality of each individual neighborhood cannot be predicted for irregular, triangle meshes, like those typical of acquired 3D-data. However, that becomes less important because it is possible to know the total census count of neighbors in all neighborhoods of $\mathcal{N}$ by actually counting the size of each neighborhood a posteriori, which is done in Algorithm 7. In the next chapter, we will use $\hat{n}$ to unroll this pair of loops and instruct a calculable number of threads to efficiently process the loop block in parallel.

Figure 5.2 illustrates the data dependencies inherent to each iteration of the pair of loops over the points $\mathbf{p}_v$ in $\mathcal{P}$, and then $\mathbf{p}_i$ in $\mathcal{N}_v$, as found in lines 2 - 5 of Algorithm 4.

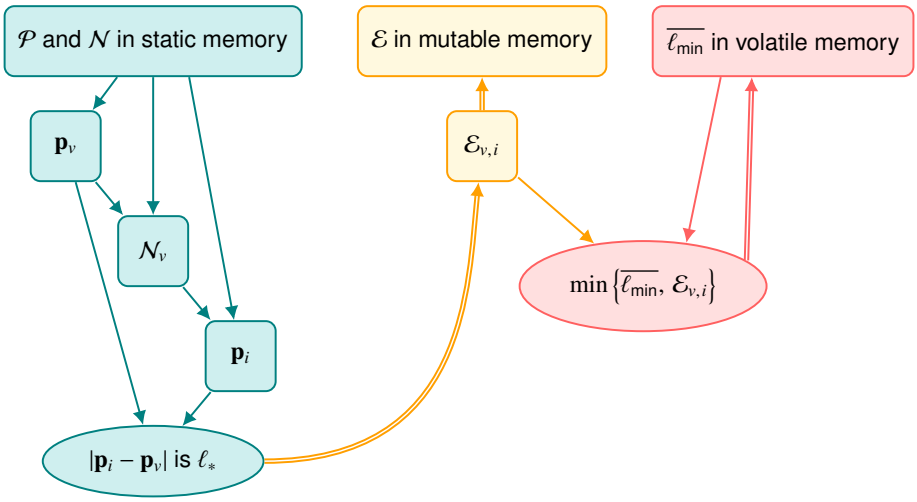


Figure 5.2. An illustration of the data dependencies found in Algorithm 4. Square nodes indicate a value stored in memory and oval nodes indicate operations. Single arrows indicate reading values and double arrows indicate writing values to memory. In teal color are static memory, and the values and operations which are independent and free of data dependencies. In sand color are mutable memory, the values stored there, and the operations which rely on them. These are independent, however, may be modified during processing. In coral color, are volatile memory, the values stored there, which may be modified at anytime during a procedure, and the operations which rely on those volatile values.

As denoted by a teal-colored lines, the first operation reads the current point $\mathbf{p}_v$ from $\mathcal{P}$ in static memory. Next, that point is referenced to read the corresponding neighborhood $\mathcal{N}_v$ from $\mathcal{N}$. Then the point $\mathbf{p}_i$ is determined and combined with $\mathbf{p}_v$ in order to compute ℓ_*, the distance between the two points.

The procedure to store the resulting independent scalar as $\mathcal{E}_{v,i}$, in its own unique place in mutable memory[38], is illustrated in sand color. Next, in coral color, the current global minimum edge length $\overline{\ell_{\min}}$ is read from volatile memory and is compared to $\mathcal{E}_{v,i}$ from mutable memory, shown in sand color, finally storing the minimum of the two values as an updated $\overline{\ell_{\min}}$ back into volatile memory.

[38]This requires special attention during implementation because operations involving reading and writing to mutable memory do not require the expensive protection of locking mechanisms, even though they may be modified concurrently.

Line 3 of Algorithm 4 is the ℓ_* operation, the most costly operation performed by the Fast One-Ring smoothing filter, due to the use of the square root operation $\sqrt{(\cdot)}$. Therefore, unlike with the union operation in Algorithm 6, it is of paramount importance that we avoid any unnecessary duplication of ℓ_*.

In the family of sets of neighborhoods, each pair of adjacent points are represented exactly twice, as illustrated by the simple example in Figure 2.5, being indexed once from both directions, and while calculating the length both times is exactly what we want to avoid, whether to actually store the length twice is but a design choice related to the optimization of memory vs speed.

The benefit of storing the set of adjacent points twice, once in both directions, it that it creates an implicit reverse lookup-table which is well documented for increasing the speed of computations, and further simplifies the complexity of indexing the values. If the average number of neighbors per point is six, then the number of edge lengths to be calculated and stored will be six times as large as the cardinality of $\mathcal{P}$. Conversely, in order to save nearly half[39] of that memory, one could store the value only once by implementing the control structures for detecting if an edge length has already been saved, then when retrieving the values, one could search for the edge length required at the cost of compute time. In the next section, we choose to implement the first, speedier method.

5.2.2. Parallel Variant of Calculate Edge Lengths

In this section, we apply the knowledge gained from the analysis of Algorithm 4 conducted in Section 5.2.1, in order to provide the pseudo code definitions with which one may implement a parallel variant of the function *serialCalculateEdgeLengths*.

The mathematical pseudo code for the parallel algorithm for calculating all the edge lengths between each pair of adjacent points in the mesh is presented in Algorithm 8. It requires the knowledge of and access to the number of available processors in the system ρ, the set of all points $\mathcal{P}$, the family of sets of neighborhoods $\mathcal{N}$, and the count of all neighbors in all neighborhoods $\hat{n}$, and produces as an output, the set of pre-calculated edge lengths $\mathcal{E}$, and the global minimum edge length $\overline{\ell_{\min}}$; both instrumental to the calculation of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. This algorithm has been further split into three kernels in order to maximize the efficiency of the overall process by facilitating the balance of work loaded onto each processor in parallel.

The initial function, *parallelCalculateEdgeLengths*, requires all of the inputs listed in the previous paragraph, then calculates the stride with the problem size equating to the count of all neighbors in all neighborhoods $\hat{n}$. Next, the average size of all neighborhoods in $\mathcal{N}$ is calculated as $\bar{n}$, and the single mutex μ is created to be used in protecting reads and writes to to $\overline{\ell_{\min}}$. Then, the loop iterating over a portion of the total number of processors, scaled by the average neighborhood size, is begun, calling in each iteration, the kernel *calculateLengths*. Because each thread will be expected to spawn additional threads[41] for each pair of adjacent points found in each neighborhood

[39] Scaling by half comes from the observation that as mesh density increases, the ratio of border to non-border edges dimensions, as discussed in greater detail in Appendix A.

[40] While it is true that we are attempting to avoid any unnecessary edge length calculations or mutex locks, the hidden message in this line is honestly just a happy accident.

[41] the greater portion of the total thread count, overall.

Algorithm 8: Parallel algorithm for calculating all the edge lengths between each pair of adjacent points in the mesh

Input : the number of available processors ρ,
the set of all points $\mathcal{P}$,
the family of sets of neighborhoods $\mathcal{N}$,
the count of all neighbors in all neighborhoods $\hat{n}$
Output: the set of pre-calculated edge lengths $\mathcal{E}$,
the global minimum edge length $\overline{\ell_{\min}}$

1 **Function** *parallelCalculateEdgeLengths(ρ, $\mathcal{P}$, $\mathcal{N}$, $\hat{n}$)*
2 $\quad\sigma \leftarrow \hat{n} \,/\, \rho$
3 $\quad\bar{n} \leftarrow \hat{n} \,/\, |\mathcal{P}|$
4 $\quad$**for** $\Pi \in \{1, \ldots, \lceil \rho/\bar{n} \rceil\}$ **:**
5 $\quad\quad$ *~calculateLengths(Π, σ, μ, $\mathcal{P}$, $\mathcal{N}$)*
6 $\quad$ *synchronizeThreads()*

7 **Kernel** *calculateLengths(Π, σ, $\mathcal{P}$, $\mathcal{N}$)*
8 $\quad\check{\sigma} \leftarrow (\Pi - 1)\,\sigma + 1$
9 $\quad\hat{\sigma} \leftarrow \Pi\,\sigma$
10 $\quad$**create**(μ)
11 $\quad$**for** $\mathbf{p}_v \in \{\mathbf{p}_{\check{\sigma}}, \ldots, \mathbf{p}_{\hat{\sigma}}\}$ **:**
12 $\quad\quad$**for** $\mathbf{p}_i \in \mathcal{N}_v$ **:**
13 $\quad\quad\quad$ *~safeEdgeLengthCalculation(μ, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathbf{p}_v$, $\mathbf{p}_i$)*

14 **Kernel** *safeEdgeLengthCalculation(μ, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathbf{p}_v$, $\mathbf{p}_i$)*
15 $\quad\mathcal{E}_{v,i} \leftarrow |\mathbf{p}_i - \mathbf{p}_v|$ /* ℓ_*, Eq: 3.1 */
16 $\quad$**if** $\mathcal{E}_{v,i} < \overline{\ell_{min}}$ **then** /* heuristic only[40] */
17 $\quad\quad$**lock**(μ)
18 $\quad\quad\overline{\ell_{\min}} \leftarrow \min\left\{\overline{\ell_{\min}}, \mathcal{E}_{v,i}\right\}$ /* Eq: 3.2 */
19 $\quad\quad$**unlock**(μ)

in its designated stride of the work load, the greater portion of processors are held in reserve for those spawned threads to be able to run in parallel. Notice the "ceiling" operator $\lceil\cdot\rceil$, which ensures that at least instance of *calculateLengths* will be executed. Finally, all the working threads must be synchronized before the set of pre-calculated edge lengths $\mathcal{E}$, or the final value of global minimum edge length $\overline{\ell_{\min}}$, may be used.

The kernel *calculateLengths* requires an index Π, which it uses along with the stride *sigma*, to calculate the lower and upper boundaries, $\check{\sigma}$ and $\hat{\sigma}$, for the stride of the total problem size which it should compute. Next, the function iterates over each point $\mathbf{p}_v$ within its stride boundaries, as well as each neighbor $\mathbf{p}_i$ in the current center point's neighborhood $\mathcal{N}_v$, spawning new threads in each iteration to execute instances of the *safeEdgeLengthCalculation* kernel.

The *safeEdgeLengthCalculation* kernel requires the set of edge lengths $\mathcal{E}$, which is being collaboratively populated by each other instance of the kernel, as well as the

single mutex *mu*, used to guard the final reading of and writing to the shared value of $\ell_{\min}$. Naturally, also required are the coordinates of the two points, between which the distance is being calculated. In line 15, the first line of the function, the L2-norm of the difference between the two points is already calculated. This is the ℓ_* operation. Also notice that this operation writes to shared memory outside of the protection of a mutex. This is only possible because no two threads will attempt to write to the same value in memory, as discussed in Section 5.2.1 and illustrated in Figure 5.2.

There are two race conditions in line 18 which must be avoided, however it would be incredibly inefficient to have all $(\bar{n} - \rho)$ / $\bar{n}$ threads attempt to lock the same, single mutex μ. The solution is the heuristic conditional statement in line 16, testing the worthiness of incurring the cost of attempting to lock the shared mutex guarding the reads and writes to $\ell_{\min}$. We describe this conditional statement heuristic, in order to call attention to the fact that it may give an inaccurate result due to race conditions between the threads operating in parallel.

Algorithm 8 produces as an output, the set of pre-calculated edge lengths $\mathcal{E}$, and the global minimum edge length $\ell_{\min}$; both instrumental to the calculation of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, by calculating in parallel, each pair of adjacent points, in each neighborhood, comprising the family of sets of neighborhoods $\mathcal{N}$. This algorithm has been further split into three kernels in order to maximize efficiency by facilitating the balance of work loaded onto each processor in parallel.

In the next section, we will see how the pre-calculations of Algorithms 6, 7, and 8 contribute, alongside the modifications to exploit the concurrency inherent to the filter, to the speed and scalability of the main procedure, defined in Algorithm 9.

5.3. Convolving the Filter in Parallel

Finally, in this section, we examine the serial Algorithm 5, which includes the principle loop to convolve the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. As a prerequisite, one must have successfully executed the previous three parallel algorithms, as detailed in Sections 5.1 and 5.2, in order to generate the input parameters which will be used in the parallel variant of this algorithm. First we will analyze the serial algorithm through the lens of parallel programing, to be followed by defining the strategy behind the design of the parallel variant, then finally providing the pseudo code definitions to implement Algorithm 8.

5.3.1. Analysis of Serial Algorithm 5: Convolve Filter

In this section, we analyze the Serial Algorithm for Convolving the Filter as presented in Section 4.4, through the lens of parallel programing. The technique we employ will match that of the previous three algorithms, which involves the reading of the serial algorithm line by line, building a dependency graph, and as opportunities for exploiting parallelism are discovered, devising a strategy to maximize efficiency using parallel processing.

Starting in line 1 of Algorithm 5, we read the declaration of the function *serialConvolveFilter* requiring as inputs: the set of points $\mathcal{P}$, the set of function values $\mathcal{F}$, the

user-defined number of convolutions to perform τ, as well as the information generated in the previous three algorithms, the family of sets of neighborhoods $\mathcal{N}$, the set of pre-calculated edge lengths $\mathcal{E}$, and the global minimum edge length $\overline{\ell_{\min}}$. Then, in the first line of the function *serialConvolveFilter*, we encounter a loop which runs for the user-defined number of convolutions τ. In order to determine if one can parallelize the entire loop, once must closely analyze each operation within the loop block; searching for loop-carried control or data dependencies.

So, skipping ahead to the last line of the loop block, line 18, we read that the set of function values $\mathcal{F}$ is replaced by the newly computed set of weighted mean function values $\mathcal{F}'$, before starting the next convolution. This models exactly the definition of a loop-carried data dependency, therefore it would be impossible to efficiently calculate this loop entirely[42] in parallel.

In the second and third lines of function *serialConvolveFilter*, we encounter two more loops iterating over each point in $\mathcal{P}$, and then each neighborhood $\mathcal{N}_v$ associated with those points. We have seen this pattern before in Section 5.2.1; together, these two loops iterate over the non-deterministic membership of the family of sets of neighborhoods $\mathcal{N}$, and indeed can be unrolled and computed in parallel by utilizing the total count of neighbors and the average size of all neighborhoods, barring any occurrences of loop-carried dependencies within the loop block.

Figure 5.3 illustrates the data dependencies inherent to each iteration of the pair of loops over the points $\mathbf{p}_v$ in $\mathcal{P}$, and then $\mathbf{p}_i$ in $\mathcal{N}_v$, as found in lines 1 - 18 of Algorithm 5.

The first several operations exclusively require values stored in static memory, a fact which exposes an opportunity for exploiting the independence of these operations by computing them in parallel. As denoted by the teal colored lines and nodes, the current point $\mathbf{p}_v$ is read from $\mathcal{P}$ in static memory. Next, that point is referenced to read the corresponding neighborhood $\mathcal{N}_v$ from $\mathcal{N}$ in static memory. Then the points $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ are determined, and combined with $\mathbf{p}_v$ in order to reference the precomputed edge lengths $\mathcal{E}_a$, $\mathcal{E}_b$, and $\mathcal{E}_c$, stored in $\mathcal{E}$, also in static memory. The three edge lengths are then used in the operation to calculate α, which is subsequently used to calculate β, then combined with $\overline{\ell_{\min}}$ from static memory to calculate A and $\check{\ell}$. Next, β combines with $\overline{\ell_{\min}}$ in an operation to calculate ζ, which is then used with the edge lengths $\mathcal{E}_b$ and $\mathcal{E}_c$ in order to compute $\tilde{\ell}_j$ and $\tilde{\ell}_{j+1}$.

Then in sand color to highlight the fact that the values stored in mutable memory may change between iterations, we see that the function value in $\mathcal{N}_v$, of each point already in consideration, is read from mutable memory as the values f_0, f_j, and f_{j+1}. Next, $\tilde{\ell}_j$ and $\tilde{\ell}_{j+1}$ combine with f_j, and f_{j+1} in two operations which separately calculate the values f'_j, and f'_{j+1}, which are then combined with $\check{\ell}$ and f_0 to eventually calculate $\check{f}$.

Next, drawn in coral color to highlight the fact that the correctness of the values stored in volatile memory cannot be guaranteed without the protection of a locking mechanism, the current values of $\tilde{f}_v$ and $\tilde{A}_v$ are read from volatile memory, so that $\tilde{f}_v$ may combine with A and $\check{f}$ in order to accumulate the value of $\tilde{f}_v$, while $\tilde{A}_v$ combines with A in order to accumulate the value of $\tilde{A}_v$. Once $\tilde{f}_v$ and $\tilde{A}_v$ have totally accumulated, and

[42] What may be possible, would be to pipeline calculations for selected points between convolutions, since each calculation only actually requires the new function values of its neighbors, not the entire set. However, given the effort required to compute a single convolution of the Fast One-Ring smoothing filter on typical acquired 3D-data, it is unlikely that any additional speedup will be realized using such a technique, computing with the GPGPUs commercially available today.

their threads have been synchronized, drawn in sand color, we see that $\tilde{f}_v$ and $\tilde{A}_v$ combine to finally calculate the value f'_v, which is stored in mutable memory to be utilized in subsequent computations.

Having now considered the dependency graph in Figure 5.3, we shall continue analyzing, line by line, the Serial Algorithm 5, returning to the beginning of the loop block of the inner most nested loop.

The Lines 5 - 13 only generate new terms which are unique to the current iteration, using only original values which are not changed by any other thread, therefore every operation in the block is a prime candidate for exploiting concurrency in the loop. Line 10 starts a new loop over just two values, and while it would be possible to compute both halves of the values the loop in parallel, at this point in the computation, we would expect all processors to already be in use calculating the other iterations, so dedicating the resources to spawn new threads here would likely net a loss in efficiency, so we will simply unroll the loop and compute the four new terms in serial, per thread.

Lines 14 and 15 both accumulate values in shared memory which will be accessed in parallel, on average $\bar{n}$ times, setting up race conditions and threatening the correctness of the algorithm. Because of the limited number of possible collisions, we will treat these two lines as a single critical section, protecting each pair with a shared mutex as we did for the paired union operations in Algorithm 6. However, in contrast, these mutexes will have a limited scope, not being shared globally, but only shared within a local neighborhood.

In line 16, the values $\tilde{f}_v$ and $\tilde{A}_v$, having been accumulated for each neighbor in $\mathcal{N}_V$, are used to calculate the weighted mean function value for the entire geodesic disc f'_v, but before their values are read, all threads working on that neighborhood must be synchronized to ensure the correct values are used. As thread synchronization can be very expensive to computation time, a fact elaborated on in Section 2.3.5.3, it is imperative that only those threads working on the neighborhood are synchronized, and not all threads, as is necessary in most other parts of this algorithm, including between the next two lines.

In lines 17 and 18, all the values of f'_v from each neighborhood in $\mathcal{N}$ are collected into the new set $\mathcal{F}'$, then either returned as the final result of the filter's convolutions, or used as the scalar field of function values in the next iteration of the filter. Storing the values in the new set may be done in parallel, because each value will remain unique in the set, however, before the new set may be used in either of those two ways, all threads of execution must have time to synchronize in order to ensure that all values have had time to populate the set, thus avoiding the use of incorrect values.

Now that we have analyzed, the serial Algorithm 5, and discovered opportunities where exploiting the loop-level concurrency will benefit the efficiency of the Fast One-Ring smoothing filter, let us not hesitate to create the parallel variant.

5.3.2. Parallel Variant of Convolve Filter

In this section, we apply the knowledge gained from the analysis of Algorithm 5 conducted in Section 5.3.1, in order to provide the pseudo code definitions with which one may implement a parallel variant of the function *serialConvolveFilter*.

Algorithm 9 is split into three parts, the main function *parallelConvolveFilter*, and

two kernels *safeAccumulateGeoDiscMean* and *safeAccumulateSectorMean*, in order to facilitate the delegation of portions of the total work load to separate threads of execution, while protecting critical sections, and avoiding data and control dependencies, so that the threads can be safely executed in parallel. The main function *parallelConvolveFilter* requires several inputs, as did its serial counterpart, which include: the set of points $\mathcal{P}$, the set of function values $\mathcal{F}$, the user-defined number of convolutions to perform τ, as well as the information generated in the previous three algorithms, the family of sets of neighborhoods $\mathcal{N}$, the set of pre-calculated edge lengths $\mathcal{E}$, and the global minimum edge length $\overline{\ell_{\min}}$. In addition, it also requires the number of available processors ρ, and the count of all neighbors in all neighborhoods $\hat{n}$, specifically for this parallel variant of the algorithm.

In line 2, the first line of the function, the loop which iterates for the user-defined number of times τ begins. Initially, the stride σ is calculated with the work load computed as one for each point defining a geodesic disc Ω, which is equal to the cardinality of points in the mesh $|\mathcal{P}|$, plus one for each neighboring point in every neighborhood $\hat{n}$. Next, another loop is begun, which iterates over a portion of the available processors in the system, as we did in Algorithm 8, reserving resources for the rest of the threads that must be spawned in order to compute in parallel, the weighted mean function value over an entire neighborhood f'_v. In each iteration of this loop, a new thread is spawned to execute in parallel the kernel *safeAccumulateGeoDiscMean*. At the conclusion of the loop block, all threads are synchronized before either the new scalar field of weighted mean function values $\mathcal{F}'$ is used as the function values upon which the next convolution of the filter is based, or the convolutions end and $\mathcal{F}'$ is returned as the final result.

The kernel *safeAccumulateGeoDiscMean*, whose definition begins in line 6, is called for each processor not reserved for sector-wise computations, being passing to it as parameters: a process index Π, the stride length σ, the new scalar field which is to be populated with calculated weighted mean function values from each neighborhood $\mathcal{F}'$, and the other general information defining and describing the mesh, $\mathcal{P}$, $\mathcal{N}$, $\mathcal{E}$, $\overline{\ell_{\min}}$, $\mathcal{F}$. This kernel begins by using its designated index Π, to compute the lower and upper boundaries, $\check{\sigma}$ and $\hat{\sigma}$, for the stride of the total problem size on which it should compute. Next, the kernel iterates over each point $\mathbf{p}_v$ within its stride boundaries, creating a new set of mutexes $\mathfrak{M}$, containing one μ per neighbor in currently neighborhood $\mathcal{N}_v$, before beginning another loop. This nested loop iterates over each neighbor $\mathbf{p}_i$ in the current neighborhood $\mathcal{N}_v$, and spawns new threads to execute instances of the *safeAccumulateSectorMean* kernel. After the loop over each point in $\mathcal{N}_v$ is complete, the threads which were used to calculate the weighted mean function values for each circle sector are synchronized, before their values are used in the calculation of f'_v, the weighted mean function value at point $\mathbf{p}_v$, which is collected in the set $\mathcal{F}'$. While $\mathcal{F}'$ is a set shared among all the threads, the assignment of f'_v does not require the protection of a locking mechanism against race conditions, because during any given convolution of the filter, only one single thread will ever compute and attempt to store the value of f'_v at the location $\mathcal{F}'_v$.

In line 16, a new thread is spawned to execute an instance of the function *safeAccumulateSectorMean* for each sector in the current neighborhood. The first nine lines of *safeAccumulateSectorMean* culminate in the computation of $\check{f}$, the weighted mean function value, for the current circle sector being processed. As seen in Figure 5.3, because each of these operations are free of data and control dependence, besides the dependence of the results of the other operations in this block, these nine instructions

may all be computed in serial by individual threads, without requiring the expensive protection of a locking mechanism.

Furthermore, because the goal of this block is to accumulate these weighted mean function values in line 30, as well as the area for each sector in line 31, so that both may be used in the calculation of the weighted mean over the entire geodesic disc in line 18, we chose to guard the shared memory values $\tilde{f}_v$ and $\tilde{A}_v$ using mutexes, risking on average a small number of collisions, only $\bar{n}$ per neighborhood. An alternative solution, which we choose not to implement here because it would cost $2 \cdot \hat{n}$ more floating point values worth of memory, is to save each intermediate value separately, then perform the averaging calculation on those values after having synchronized all the threads.

The final result is that the weighted mean function values $\tilde{f'}_v$ and areas $\tilde{A}_v$ of each circle sector are calculated in parallel, then those values are averaged, also in parallel, over the total area of the geodesic disc centered at each $\mathbf{p}_v$, and collected in the set $\mathcal{F}'$, completing a single convolution of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds.

5.4. Summary

In this chapter, we explored the serial version of this as-yet-unpublished algorithm, in order to negotiate any instances of control or data dependency, and were able to discover several manifestations of independent procedures worthy of exploiting with parallel processing, which when implemented on a system capable of parallel computation, significantly improves the performance of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. The parallel algorithm presented in this chapter has three main parts, each further split into kernels, to facilitate the simultaneous execution of blocks of instructions by threads in parallel. This algorithm was developed by first by analyzing the corresponding serial algorithm through the lens of parallel programing, defining the strategy behind the design of the parallel variant with the aid of data dependency diagrams, then finally providing the pseudo code definitions to implement each part of the algorithm.

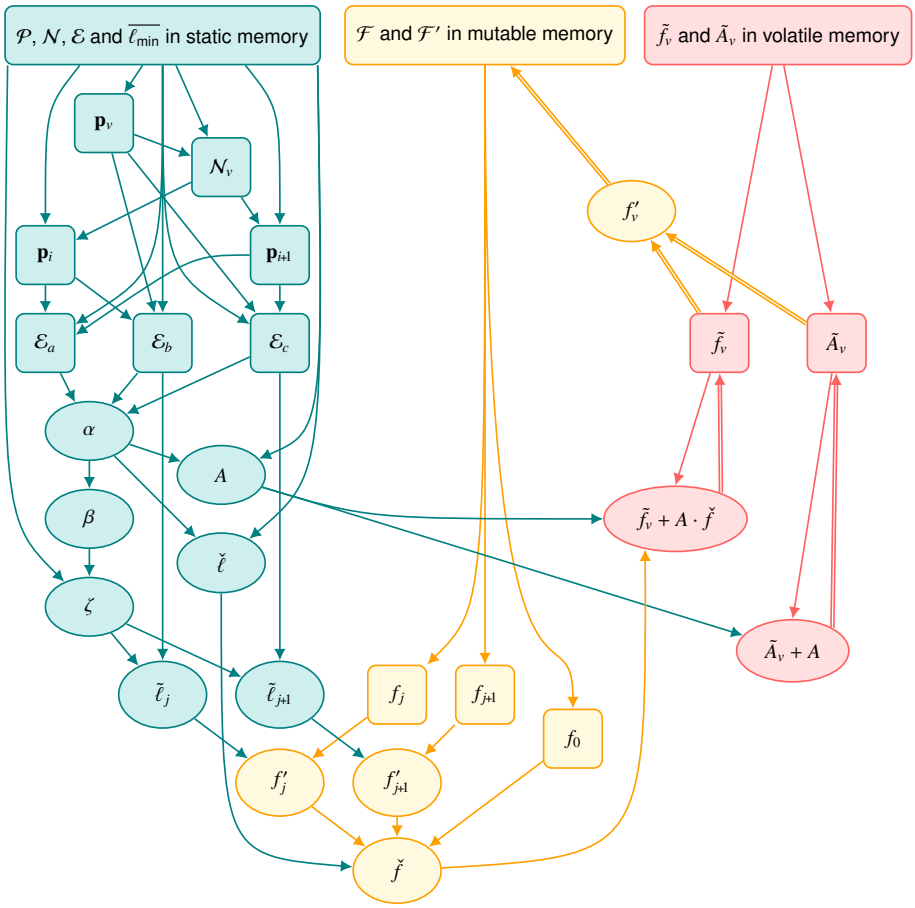


Figure 5.3. An illustration of the data dependencies found in Algorithm 5. Square nodes indicate a values stored in memory, oval nodes indicate operations. Single arrows indicate reading values, double arrows indicate writing values to memory. Drawn in teal color are static memory, and the values and operations which are independent and free of data dependencies. Drawn in sand color are mutable memory, the values stored there, and the operations which rely on them. These are independent, however may be modified during processing. Drawn in coral color, are volatile memory, the values stored there, which may be modified at anytime during a procedure, and the operations which rely on those volatile values.

Algorithm 9: Parallel algorithm for convolving the Fast One-Ring smoothing filter for scalar fields on discrete manifolds

Input : the number of available processors ρ,
the set of all points $\mathcal{P}$,
the family of sets of neighborhoods $\mathcal{N}$,
the count of all neighbors in all neighborhoods $\hat{n}$,
the set of pre-calculated edge lengths $\mathcal{E}$,
the global minimum edge length $\overline{\ell_{\min}}$,
the set of function values $\mathcal{F}$,
the user-defined number of convolutions τ

Output: the set of one-ring weighted mean function values $\mathcal{F}'$

Function *parallelConvolveFilter*(ρ, $\mathcal{P}$, $\mathcal{N}$, $\hat{n}$, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathcal{F}$, τ)
- **for** $t \leftarrow 1$ **to** τ **:**
  - $\sigma \leftarrow (|\mathcal{P}| + \hat{n}) \,/\, \rho$
  - $\bar{n} \leftarrow \tilde{n} \,/\, |\mathcal{P}|$
  - **for** $\Pi \in \{1, \ldots, \lceil \rho/\bar{n} \rceil\}$ **:**
    - ~*safeAccumulatesGeoDiscMean*(Π, σ, $\mathcal{P}$, $\mathcal{N}$, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathcal{F}$, $\mathcal{F}'$)
  - *synchronizeThreads*()
  - **if** $t < \tau$ **then** /* smooth newest values every convolution */
    - $\mathcal{F} \leftarrow \mathcal{F}'$
- **return** $\mathcal{F}'$

Kernel *safeAccumulatesGeoDiscMean*(Π, σ, $\mathcal{P}$, $\mathcal{N}$, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathcal{F}$, $\mathcal{F}'$)
- $\check{\sigma} \leftarrow (\Pi - 1)\,\sigma + 1$
- $\hat{\sigma} \leftarrow \Pi\,\sigma$
- **for** $\mathbf{p}_v \in \{\mathbf{p}_{\check{\sigma}}, \ldots, \mathbf{p}_{\hat{\sigma}}\}$ **:**
  - $\mathfrak{M} \leftarrow \{\mu_1, \ldots, \mu_{|\mathcal{N}_v|}\}$
  - **for** $\mathbf{p}_i \in \mathcal{N}_v$ **:**
    - ~*safeAccumulateSectorMean*($\mathfrak{M}$, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathbf{p}_v$, $\mathbf{p}_i$, $\mathcal{F}$, $\tilde{f}_v$, $\tilde{A}_v$)
  - *synchronizeThreads*($\{\check{\sigma}, \ldots, \hat{\sigma}\}$)
  - $\mathcal{F}'_v \leftarrow \tilde{f}_v \,/\, \tilde{A}_v$ /* as f'_v, Eq: 3.14 */

Kernel *safeAccumulateSectorMean*($\mathfrak{M}$, $\mathcal{E}$, $\overline{\ell_{min}}$, $\mathbf{p}_v$, $\mathbf{p}_i$, $\mathcal{F}$, $\tilde{f}_v$, $\tilde{A}_v$)
- $\alpha \leftarrow cos^{-1}\left(\frac{\mathcal{E}_c^2 + \mathcal{E}_b^2 - \mathcal{E}_a^2}{2 \cdot \mathcal{E}_c \cdot \mathcal{E}_b}\right)$ /* Eq: 3.4 */
- $\beta \leftarrow (\pi - \alpha) \,/\, 2$ /* Eq: 3.5 */
- $A \leftarrow \left(\overline{\ell_{\min}}\right)^2 \cdot \alpha \,/\, 2$ /* Eq: 3.6 */
- $\check{\ell} \leftarrow (4 \cdot \overline{\ell_{\min}} \cdot \sin(\alpha \,/\, 2)) \,/\, 3\,\alpha$ /* Eq: 3.7 */
- $\zeta \leftarrow \overline{\ell_{\min}} \,/\, \sin(\beta)$ /* Eq: 3.8 */
- **for** $j \in 1, 2$ **:**
  - $\tilde{\ell}_j \leftarrow \zeta \,/\, \mathcal{E}_j$ /* Eq: 3.9, 3.10 */
  - $f'_j \leftarrow f_0 \cdot (1 - \tilde{\ell}_j) + f_j \cdot \tilde{\ell}_j$ /* Eq: 3.11, 3.12 */
- $\check{f} \leftarrow f_0 \cdot (1 - \check{\ell}) + ((f'_1 + f'_2) \cdot \check{\ell}) \,/\, 2$ /* Eq: 3.13 */
- **lock**(μ)
- $\tilde{f}_v \leftarrow \tilde{f}_v + A \cdot \check{f}$ /* Eq: 3.14 */
- $\tilde{A}_v \leftarrow \tilde{A}_v + A$ /* Eq: 3.14 */
- **lock**(μ)

Chapter 6

Experiments & Evaluation

The goal of our experimentation was two-fold. First, it was our intuition that it would be useful to visualize the Fast One-Ring smoothing filter for scalar fields on discrete manifolds response when convolved over both regular and irregular synthetic 3D-data with the Dirac delta function applied as a scalar field, and when convolved over acquired 3D-data with the result of MSII applied as a scalar field. Secondly, the performance of the parallel algorithm was evaluated in regards to the speedup and efficiency obtained by convolving the Fast One-Ring smoothing filter over meshes of different sizes, using different hardware configurations.

In Section 6.1, we analyze the filter response when convolving the filter over four different configurations of synthetic data, with each image of a mesh visualized using the GigaMesh framework [28], with function values colored using the "Improved Hot" color ramp, then exported as a raster image in the Portable Network Graphic (PNG) format. Next, in Section 6.2 we analyze the same for three different examples of acquired 3D-data. Then, in Section 6.3, we establish the compute times for a multitude of pairs of experiments, involving (a) both the serial and parallel algorithms, (b) both acquired and synthetic 3D-data, and (c) four different configurations of hardware.

6.1. Synthetic 3D-data

In order to study the effect of each convolution of the Fast One-Ring smoothing filter on individual function values of a scalar field, we generated synthetic 3D-data arranged into four different configurations of triangle meshes, then applied the Dirac delta function as a scalar field. This involves applying a function value of one at the center point, and zero everywhere else, before iteratively convolving the Fast One-Ring smoothing filter multiple times over each mesh.

The four synthetic mesh configurations are: the bisected-square tessellation, the quadrisected-square tessellation, the hexagonal tessellation, and the random triangulated disc. These were generated using original C++ source code which was written specifically for the research presented in this thesis. However, as such tools may also be found useful by other researchers, it is our intention to make the source code for each generator as accessible as this thesis.

6.1.1. Bisected-Square Tessellations

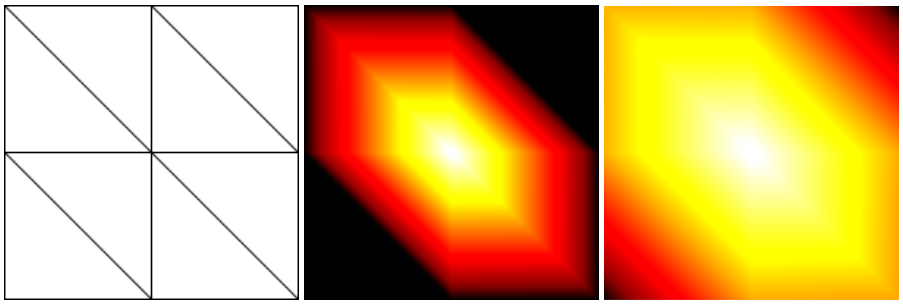
The synthetic mesh generator for bisected-square tessellations generates meshes characterized by rings of squares around a center point, with the northwest and southeast corners made adjacent, so as to bisect each square into two equally-sized isosceles, right triangles. This results in two dissimilar edge lengths adjacent to the center point.

The smallest, non-trivial mesh, generated with the parameter r equal to one, is composed of four squares around the center point. These are represented in total by only nine points and eight faces. However those numbers grow quickly with increasing parameter size r, according to the two equations

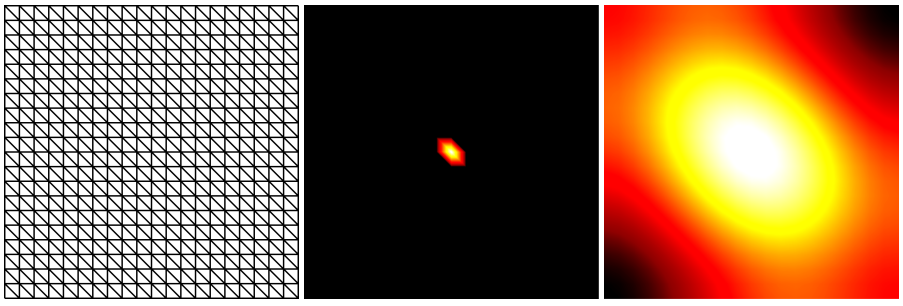
$$|\mathcal{P}| = 4r^2 + 4r + 1 \tag{6.1}$$

$$|\mathcal{T}| = 8r^2 \tag{6.2}$$

Figure 6.1 shows a comparison of two differently-sized, bisected-square tessellations, generated with parameters r set to 1 and 10. Shown in (a) and (d) are each in wireframe. Next in (b) and (e), each is colored by function value before convolving the filter. Then, in (c) $r = 1$ is shown after convolving the filter once. Finally in (f) $r = 10$ is shown after convolving the filter 100 times. Notice how the filter response appears to travel faster across the image, along the longer, diagonal edge lengths.



(a) $r = 1$, wireframe **(b)** $r = 1, c = 0$ **(c)** $r = 1, c = 1$



(d) $r = 10$, wireframe **(e)** $r = 10, c = 0$ **(f)** $r = 10, c = 100$

Figure 6.1. Comparison of two differently-sized, bisected-square tessellations, generated with parameters r set to 1 and 10. (a) $r = 1$ in wireframe, (b) $r = 1$ colored by function value before convolving the filter, (c) $r = 1$ colored by function value after convolving the filter once, (d) $r = 10$ in wireframe, (e) $r = 10$ colored by function value before convoving the filter, and (f) $r = 10$ colored by function value after convolving the filter 100 times.

6.1.2. Quadrisected-Square Tessellations

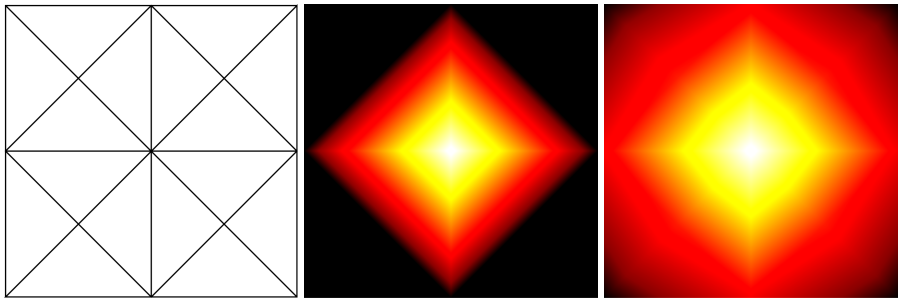
The synthetic mesh generator for quadrisected-square tessellations generates meshes characterized by rings of squares around a center point, with every corner made adjacent to the center point, quadrisecting each square into four equally-sized, isosceles, right triangles.

The smallest, non-trivial mesh, generated with the parameter r equal to one, is composed of four squares around the center point, represented in total by only thirteen points and sixteen faces. However those numbers grow quickly with increasing parameter size r, according to the two equations

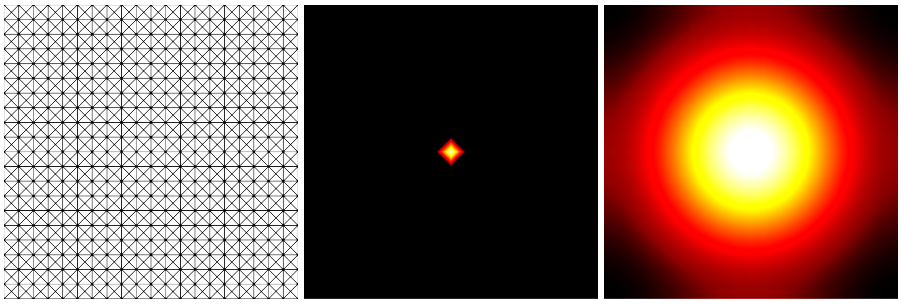
$$|\mathcal{P}| = 8r^2 + 4r + 1 \tag{6.3}$$

$$|\mathcal{T}| = 16r^2 \tag{6.4}$$

Figure 6.2 shows a comparison of two differently-sized, quadrisected-square tessellations, generated with parameters r set to 1 and 10. Shown in (a) and (d) are each in wireframe. Next in (b) and (e), each is colored by function value before convolving the filter. Then in (c) $r = 1$ is shown after convolving the filter once. Finally in (f) $r = 10$ is shown after convolving the filter 100 times. Notice the circular shape of the filter response in part (f).



(a) $r = 1$, wireframe **(b)** $r = 1, c = 0$ **(c)** $r = 1, c = 1$



(d) $r = 10$, wireframe **(e)** $r = 10, c = 0$ **(f)** $r = 10, c = 100$

Figure 6.2. Comparison of two differently-sized, quadirsected-square tessellations, generated with parameters r set to 1 and 10. (a) $r = 1$ in wireframe, (b) $r = 1$ colored by function value before convolving the filter, (c) $r = 1$ colored by function value after convolving the filter once, (d) $r = 10$ in wireframe, (e) $r = 10$ colored by function value before convoving the filter, and (f) $r = 10$ colored by function value after convolving the filter 100 times.

6.1.3. Hexagonal Tesselation

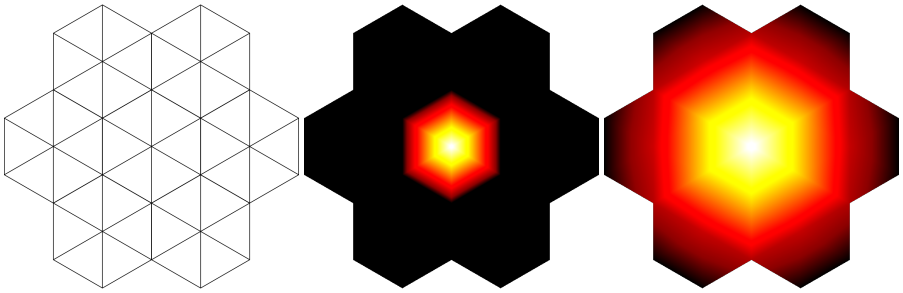
The synthetic mesh generator for hexagonal tessellations generates meshes characterized by hexagons around one central hexagon, with each corner of the hexagons made adjacent to its own center point, creating six equilateral triangles per hexagon, and ensuring that every edge length in the mesh is of equal size.

The smallest, non-trivial mesh, generated with the parameter r equal to zero, is composed of one hexagon around the center point, represented in total by only seven points and six faces. However those numbers grow quickly with increasing parameter size r, according to the two equations

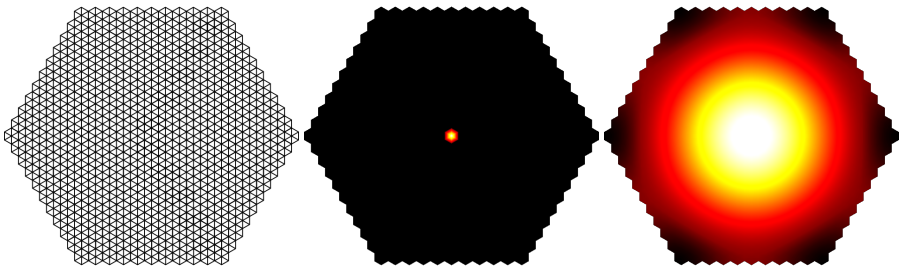
$$|\mathcal{P}| = 3r^2 + 3r + 7 + \sum_{i=1}^{r} 6(2i + 1) \tag{6.5}$$

$$|\mathcal{T}| = 18r^2 + 18r + 6 \tag{6.6}$$

Figure 6.1 shows a comparison of two differently-sized, hexagonal tessellations, generated with parameters r set to 1 and 10. Shown in (a) and (d) are each in wireframe. Next in (b) and (e), each are colored by function value before convolving the filter. Then, in (c) $r = 1$ is shown after convolving the filter twice. Finally in (f) $r = 10$ is shown after convolving the filter 200 times.



(a) $r = 1$, wireframe **(b)** $r = 1, c = 0$ **(c)** $r = 1, c = 2$



(d) $r = 10$, wireframe **(e)** $r = 10, c = 0$ **(f)** $r = 10, c = 200$

Figure 6.3. Comparison of two differently-sized, hexagonal tessellations, generated with parameters r set to 1 and 10. (a) $r = 1$ in wireframe, (b) $r = 1$ colored by function value before convolving the filter, (c) $r = 1$ colored by function value after convolving the filter once, (d) $r = 10$ in wireframe, (e) $r = 10$ colored by function value before convoving the filter, and (f) $r = 10$ colored by function value after convolving the filter 200 times.

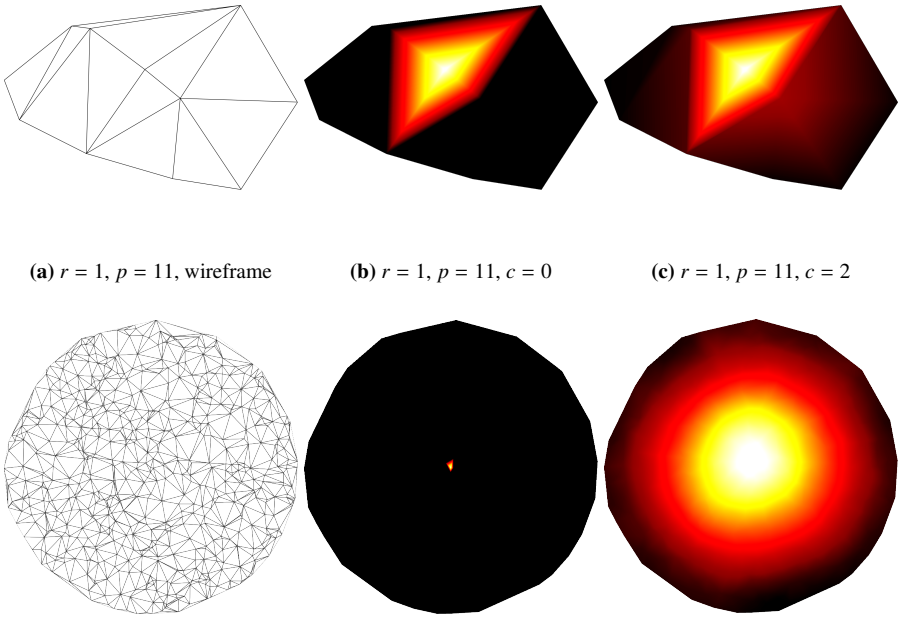
6.1.4. Random Triangulated Discs

The synthetic mesh generator for random triangulated discs generates meshes characterized by radially bounded, uniformly distributed, random points, and triangulated following the well-studied method originally presented by Delaunay [17]. Generating random triangulated discs requires two parameters: the radius of the disc r, and the number of points p. These parameters were chosen to ensure parity with the sizes and point density of the other synthetic meshes, as summarized in Table 6.1.

Radius r	**Points** p	**Faces**	**Radius** r	**Points** p	**Faces**
1	11	12	100	60,401	120,668
3	67	119	300	541,201	1,082,132
10	641	1,252	1,000	6,004,001	12,007,382
30	5,521	10,971	3,000	54,012,001	108,022,714

Table 6.1. Summary of the parameters used with the random triangulated disc generator

Figure 6.1 shows a comparison of two differently-sized, random triangulated discs, generated with parameters r set to 1 and 10, and parameters p set to 11 and 641. Shown in (a) and (d) are each in wireframe. Next in (b) and (e), each are colored by function value before convolving the filter. Then in (c) $r = 1$ is shown after convolving the filter twice. Finally in (f) $r = 10$ is shown after convolving the filter 10,000 times.



(a) $r = 1$, $p = 11$, wireframe **(b)** $r = 1$, $p = 11$, $c = 0$ **(c)** $r = 1$, $p = 11$, $c = 2$

(d) $r = 10$, $p = 641$, wireframe **(e)** $r = 10$, $p = 641$, $c = 0$ **(f)** $r = 10$, $p = 641$, $c = 10^4$

Figure 6.4. Comparison of two differently-sized, random triangulated discs, generated with parameters r set to 1 and 10, and parameters p set to 11 and 641. (a) $r = 1$, $p = 11$ in wireframe, (b) $r = 1$, $p = 11$ colored by function value before convolving the filter, (c) $r = 1$, $p = 11$ colored by function value after convolving the filter twice, (d) $r = 10$, $p = 641$ in wireframe, (e) $r = 10$, $p = 641$ colored by function value before convoving the filter, and (f) $r = 10$, $p = 641$ colored by function value after convolving the filter 10,000 times.

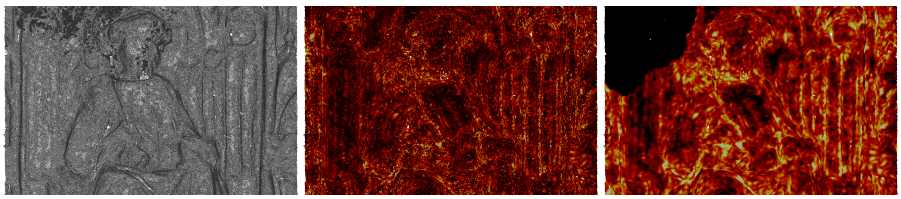
6.2. Acquired 3D-data

Each of the three acquired 3D-data examples used in our experiments were initially processed with the GigaMesh framework in order to generate a scalar field of function values using the Multi-Scale Integral Invariants filter, MSII [26]. The examples we use are the partial model of the University of Heidelberg seal, the flat surface from ILATO, and the Stanford Bunny. Also, we experience two kinds of difficulties, error propagation and diminishing effectiveness when convolving the filter. These highlight the complexity of working with acquired 3D-data.

6.2.1. The University of Heidelberg Seal

This acquired 3D-data is a partial model, comprised of 397,210 points and 789,406 faces, taken from the center of a 3D-scan of the seal of the University of Heidelberg, as established in 1386 and stored by the Univeristy of Heidelberg Archives. The original data was captured with a high resolution 3D-scanner and is stored in the heiDATA dataverse of the IWR Computer Graphics [22].

Figure 6.5 shows three views of the partial model taken from the University of Heidelberg seal: (a) in wireframe, (b) colored by function values generated by applying MSII before convolving the Fast One-Ring smoothing filter, and (d) colored by function value after convolving the Fast One-Ring smoothing filter 100 times.



(a) wireframe **(b)** $c = 0$ **(c)** $c = 100$

Figure 6.5. Three views of the Univeristy of Heidelberg seal (a) in wireframe, (b) colored by MSII value before convolving the filter, (c) colored by function value after convolving the filter 100 times.

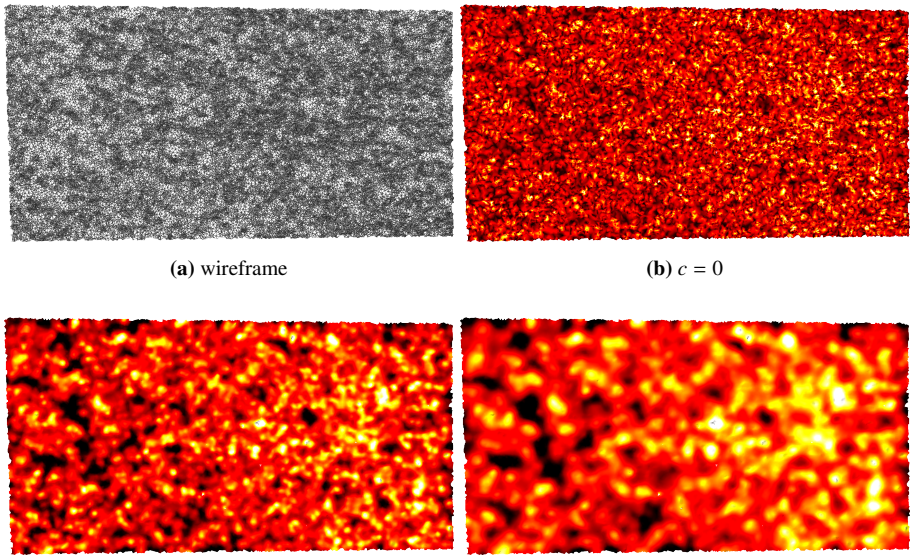
There exist several holes in this 3D-data, including the relatively large hole in the lap of the figure, just below the center of each image. However, as the Fast One-Ring smoothing filter convolves over every point, considering each neighborhood in isolation, the convexity of the entire mesh is inconsequential, and faces adjacent to holes are treated the same as faces adjacent to borders, where weighted mean function values are averaged over the sectors of the geodesic disc which do exist.

Also, notice the large black area in the top left corner of (c). Due to one of the many non-manifold irregularities of the original mesh, an error occurred when the filter attempted to divide a function value by a zero-sized area, and that error then propagated to every adjacent neighborhood in every subsequent iteration. Using the “Automatic Mesh Polishing” feature in GigaMesh [26, p. 29-32] [29, p.7], before convolving the filter, completely eliminates the problem. However further modifications to the Fast One-Ring smoothing filter algorithm may be necessary to ensure the overall stability of the filter.

6.2.2. A flat surface from ILATO

This acquired 3D-data is a partial model, comprised of 56,215 points and 111,311 faces, taken from the center of sample 1A of the set of industrial samples presented by the ILATO project , Improving Limited Angle computed Tomography by Optical data integration[14]. The original data was obtained by scanning a brushed-surface-finish aluminum block, with a Breuckmann scanners SmartScan 3D [13].

Figure 6.6 shows four views of the partial model taken from ILATO sample 1a: (a) in wireframe, (b) colored by function values generated by applying the MSII filter, before convolving the Fast One-Ring smoothing filter, (c) colored by function value after convolving the Fast One-Ring smoothing filter 1000 times, and then (d) again after convolving the filter 3000 times.



(a) wireframe **(b)** $c = 0$

(c) $c = 1000$ **(d)** $c = 3000$

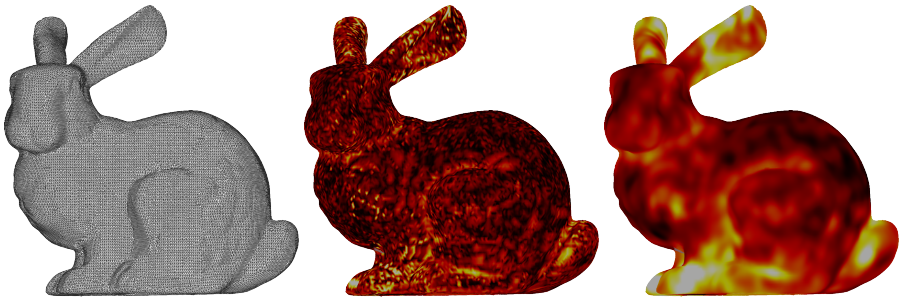
Figure 6.6. Four views of a flat surface (a) in wireframe, (b) colored by MSII value before convolving the filter, (c) colored by function value after convolving the filter 1,000 times, (c) and after 3,000 times.

As with the other acquired 3D-data, the function values were computed by applying MSII and use the same color ramp as the other models, however, because total variance over the entire model is very small, in the range of -0.018 and 0.018, practically representing the noise characteristic of the 3D-scanner and software used to generate the mesh. The range of function values are also compressed, so smoothing does not occur as rapidly as seen in other experiments, which is expected.

6.2.3. The Stanford Bunny

The Stanford Bunny was range scanned in 1994 using a Cyberware 3030MS optical triangulation scanner. The ten separate scans were then zippered together [31] to produce the 3D-data, which we obtained from The Stanford 3D Scanning Repository [23]. This mesh is comprised of 35,947 points and 69,451 faces.

Figure 6.7 shows three views of the Stanford Bunny: (a) in wireframe, (b) colored by function values generated by applying MSII, before convolving the Fast One-Ring smoothing filter, and (c) colored by function value after convolving the Fast One-Ring smoothing filter 100 times.



(a) wireframe **(b)** $c = 0$ **(c)** $c = 100$

Figure 6.7. Three views of the Stanford Bunny (a) in wireframe, (b) colored by MSII value before convolving the filter, and (c) colored by function value after convolving the filter 100 times.

In this section, we presented three examples of acquired 3D-data and experienced some of the difficulties inherent to processing and analyzing irregular triangle meshes. Without first cleaning the mesh, we experience error propagation with the university seal, and because of the relatively small variance in the function values of the flat surface, we experience a slow down in the effectiveness of the filter. In the next section we present and evaluate the results of the multitude of experiments conducted in order to establish the performance of the parallel variant of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds algorithm.

6.3. Convolving with a GPGPU

In this section, we present the collection of experiments performed with the goal of establishing the speedup and efficiency obtained when convolving the Fast One-Ring smoothing filter for scalar fields on discrete manifolds utilizing the SIMD concurrency architecture provided by GPGPUs. To that end, first we will present the optimal, sequential execution time, and the parallel compute time for the two variants of the the Fast One-Ring smoothing filter algorithm, as described in Section 6.3.1. Then we will analyze the speedup obtained for each pair of experiments in Section 6.3.2. Next we will analyze the efficiency of each pair of experiments in Section 6.3.3. Finally, we conclude with a discussion of the efficacy of our implementation of the parallel variant of the Fast One-Ring smoothing filter algorithm.

6.3.1. Compute Times

Before we are able to evaluate the speedup and efficiency of the parallel variant of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, as discussed in Section 2.3.6, we must first obtain two essential timings: the optimal sequential execution time T_s, and the parallel compute time $T_p(\hat{n}, \rho)$. In order to establish these values, we generated four versions each of the synthetic, bisected-square tessellation

and the synthetic, hexagonal tessellation, using the members of the set {1, 10, 100, 1000} as parameter r. Combined with the two acquired meshes, the university seal and the flat surface, these totaled ten meshes ranging in sizes from just nine points and eight faces, to 9×10^6 points and 1.8×10^7 faces.

Next, in order to obtain comparisons across hardware of differing capabilities, we designated two machines: a laptop M, and a desktop T. Each of these has both a CPU for serial computation, and a GPU for parallel processing. T operates on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, while M operates on an Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz. T has installed an NVIDIA Corporation GP104GL [Quadro P5000] GPU with 2,560 CUDA cores [5], while M has installed an NVIDIA Corporation GP107M [GeForce GTX 1050 Ti Mobile] with 768 CUDA cores [6].

Then, we systematically executed every combination[43] of hardware and mesh for eight increasing amounts of convolutions, {1, 3, 10, 30, 100, 300, 1000, 3000}. We restarted the count for each experiment in order to ensure that the resulting time was genuine.

For the serial algorithm, we used the implementation existing in the GigaMesh framework [28], for the parallel algorithm, a new implementation in CUDA extended C++ [**?**, p. 19-66] was developed and used. Verification of final resulting scalar field of weighted mean function values were conducted at the conclusion of each pair of experiments, and it was determined that zero difference existed between the output of the serial algorithm versus the parallel variant, even with the largest meshes for the most convolutions.

Figure 6.8 shows the timing results of every recorded experiment. The graph is drawn extra tall to ensure that the grid remains square, for an unbiased assessment of the correlations therein. Results from each combination of hardware and mesh are connected by lines.

Notice how the compute times increase linearly with both mesh size and number of iterations. Timing is very predictable when the total compute time is at least greater than 0.1 seconds, but is much less predictable for faster periods. Fortunately, the deviations from the trends only occur for short total timings, and do not appear to propagate per convolution for medium or long compute times. Therefore they are unlikely to be noticed by the typical user.

Figure 6.9 plots the total compute times of each experiment as the area of a circle[44], given different hardware and algorithm configurations, for each combination of mesh size, measured in point counts and number of convolutions of the Fast One-Ring smoothing filter.

In general, one can see that for all experiments, the total compute times increase with the size of the mesh, as well as the number of convolutions. As a result, an increase in both produces in an exponential increase in the compute time, which is as expected given the complexity of the algorithm as discussed in Chapter 3. This also demonstrates the positive effect of the parallel processing, as the growth of the compute times is much slower for the parallel variant than for the serial.

[43] Of the 320 combinations of experiments, we were only able to record data for 309 different combinations because of the memory limitations of the laptop M. However, given the strong trends already established, it is uncertain that any additional data would influence the conclusions we draw about the reported experiments.

[44] The areas of the circles were scaled by the cube root of the actual size. This is so that the smallest circle is still visible and the largest remains within the border of the plot, while the general growth trend is still recognizable.

In this section, we endeavored to establish the optimal sequential execution time T_s, and the parallel compute time $T_\rho(\hat{n}, \rho)$, given different hardware configurations and mesh sizes. In addition to the timing values, we also discovered a few trends characteristic of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. In the next section, we will continue our evaluation by calculating the speedup and efficiency obtained by processing the parallel variant of the filter algorithm using GPGPUs.

6.3.2. Speedup

Having established the optimal sequential execution time T_s, and the parallel compute time $T_\rho(\hat{n}, \rho)$ of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, we can now continue our evaluation by calculating, as discussed in detail in Section 2.3.6, the speedup obtained by convolving the filter over different sized meshes in parallel using GPGPUs.

Figure 6.10 is a plot of the speedup obtained by convolving the parallel variant of the Fast One-Ring smoothing filter over meshes of different sizes for increasing number of convolutions, using either the GPGPU of the laptop M with 768 Compute Unified Device Architecture (CUDA) cores or the GPGPU of desktop T with 2,560 CUDA cores.

One general trend which emerges is that, for both hardware configurations, more speedup can be gained when convolving the filter on meshes of larger sizes. This can be explained by the increase in the degree of parallelism, which comes with meshes with large counts of points, and is related to Amdahl's law, as discussed in Section 2.3.6.

Also, in general, but especially for larger meshes, convolving the filter using the GPGPU of the desktop T garners high speedup than the equivalent experiments using the laptop M hardware configuration. This is expected, because as the degree of parallelism increases with mesh size, the higher number of CUDA cores available to T allow it to process more computations in parallel than M. Though it never reaches the ratio of 2,560 to 768 that one might expect, as discussed in more detail in the next section, Section 6.3.3.

Another trend that develops is that for most configurations, speedup increases with increasing number of convolutions, seeming to converge to a certain number. More research must be done to determine why this occurs, but it is possible that it is related to low-level memory optimization by the operating system.

A third trend, which may not be easily seen in the graph, is that for the smallest-sized meshes, the speedup is less than one, indicating a slow down in compute time. In fact, the minimum speedup obtained in our experimentation was with the synthetic, bisected-square tesselation generated with parameter $r = 1$, consisting of only nine points, which has a speedup of $\approx$0.35 when convolving with the parallel filter on desktop T's GPGPU 3,000 times. That equates with performing three times as slow as the equivalent serial experiment, and is likely due to the overhead incurred by the necessary usage of locking mechanisms and explicit thread synchronization in the parallel variant of the algorithm.

Fortunately, the speedup obtained by convolving the parallel variant of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds using GPGPUs, is overall quite high. The maximum speedup obtained by any experiment was approximately 204.43, which equates to the parallel variant performing 200 times faster than the serial

algorithm. This speedup was obtained when convolving the parallel variant of the filter, for 3,000 convolutions, on the synthetic, quadrisected-square tesselation generated with parameter $r = 1000$, consisting of more than four million points, using desktop T's GPGPU.

Also notable, the acquired 3D-data enjoyed appreciable speedup, though considerably less than the synthetic meshes of equivalent sizes. The flat surface mesh obtained the maximum speedups of only ≈60.25 with the laptop M's GPGPU, and ≈102.74 when using desktop T's GPGPU, while the bisected-square tessellation generated with parameter $r = 100$ peaked at ≈103.67 with M, and ≈154.80 with T, despite being smaller. Likewise, the university seal obtained the maximum speedups of only ≈60.08 with the laptop M's GPGPU, and ≈113.10 when using desktop T's GPGPU, while the similarly-sized, quadrisected-square tessellation peaked at ≈115.62 with M, and ≈204.43 with T. It is unclear what caused these dependencies, but we speculate that it may be due to the wider variance in neighborhood size and function values present in 3D-data, when compared to synthetic 3D-data.

6.3.3. Efficiency

In this section, we evaluate the other important metric for analyzing the performance of a parallel algorithm, efficiency, as discussed in detail in Section 2.3.6. Having established the optimal sequential execution time T_s, and the parallel compute time $T_\rho(\hat{n}, \rho)$ of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds, efficiency is calculated as speedup divided by the number of processors used to obtain it.

Figure 6.11 is a plot of the efficiency obtained by convolving the parallel variant of the Fast One-Ring smoothing filter over meshes of different sizes for increasing numbers of convolutions, using either the GPGPU of the laptop M with 768 CUDA cores, or the GPGPU of desktop T with 2,560 CUDA cores.

The most noticeable trend in Figure 6.11 is that experiments run using the laptop M hardware configuration obtain higher efficiency than the equivalent experiments run using the desktop T hardware configuration. This trend is attributable to the design of the parallel algorithm, which is unable to scale effectively and utilize the increase in processor count provided by the GPGPU of T.

Secondly, the efficiency obtained when convolving the filter on acquired 3D-data is approximately only half of the efficiency obtained when convolving the filter on similarly-sized synthetic meshes. This is a direct consequence of the slower speedup for acquired 3D-data, as calculated in the previous section.

The maximum efficiency recorded is ≈0.151. This was obtained when convolving the Fast One-Ring smoothing filter 3,000 times on the bisected-square tessellation with four million points. This equates to only 15% of the total processor effort effectively contributing to the final values.

The minimum efficiency recorded is ≈0.000136, and similar to the minimum speedup, was obtained when convolving the Fast One-Ring smoothing filter 3,000 times on the bisected-square tessellation with only nine points. This equates to only .01% of the total processor effort effectively contributing to the final values.

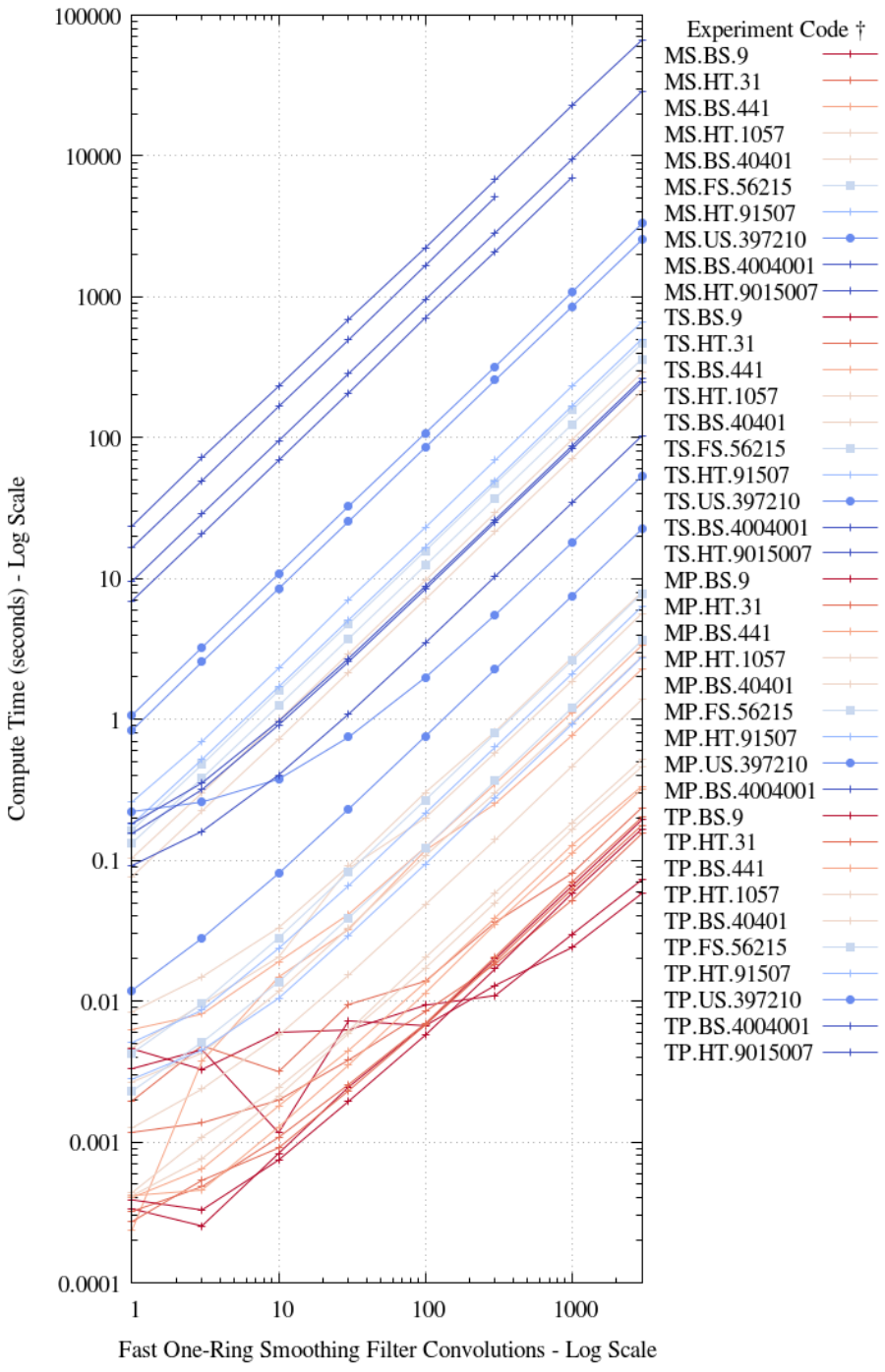


Figure 6.8. Compute times per experiment, for increasing numbers of convolutions of the Fast One-Ring smoothing filter, onto acquired and synthetic 3D-data of varying sizes. Combinations involving the smallest mesh sizes are colored red, the largest mesh sizes in blue, with all others scaled in between. The experiments involving acquired 3D-data are specially marked with squares for the flat surface mesh, and circles for the university seal.

† The experiment codes used correspond to: the system, T for the desktop or M for the laptop; the algorithm variant, P for parallel or S for serial; the mesh being processed, BS for bisected-square, HT for hexagonal tessellation, US for university seal, and FS for the flat surface; and finally, the count of points comprising the mesh.

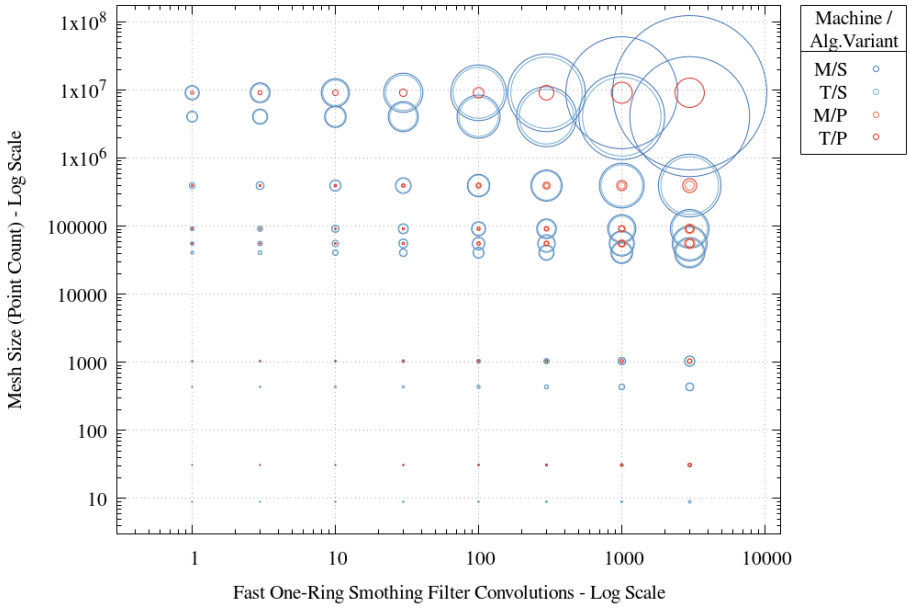


Figure 6.9. A plot of the total compute times of each experiment as the area of a circle, given different hardware configurations. Each is then plotted for each combination of mesh size measured in point counts, and the number of convolutions of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. Blue circles indicate the serial variant computed with a CPU, and a red circle indicates the parallel variant was computed on a GPGPU.

† See Figure6.8 for an explanation of the experiment codes.

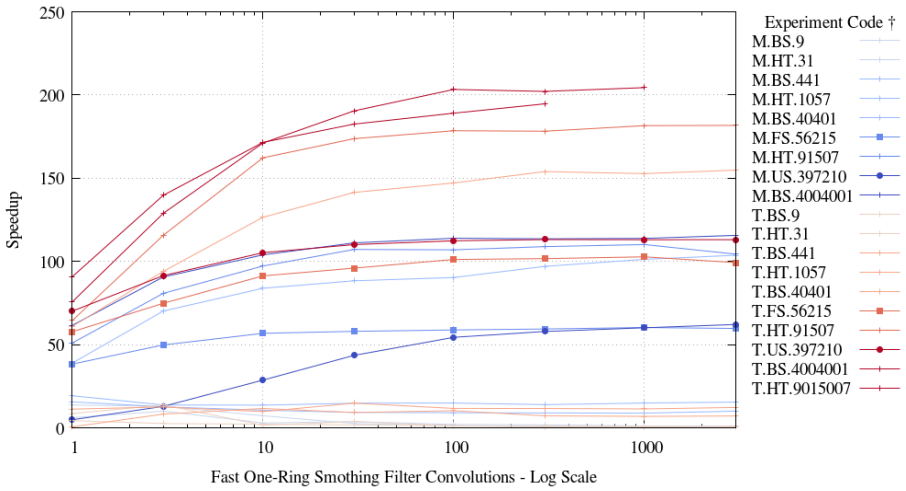


Figure 6.10. A graph of the speedup obtained by convolving the parallel variant of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds over meshes of different sizes. Lines in blue, increasing in saturation with the size of the mesh, are experiments run using the GPGPU of laptop M with 768 CUDA cores. Lines in red, increasing in saturation with the size of the mesh, are experiments run using the GPGPU of desktop T with 2,560 CUDA cores. The experiments involving acquired 3D-data are specially marked with squares for the flat surface mesh, and circles for the university seal.

† See Figure 6.8 for an explanation of the experiment codes.

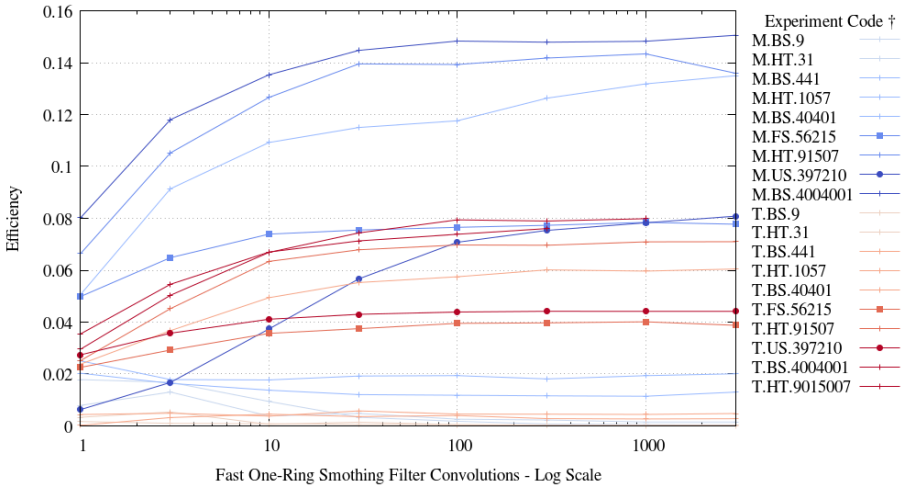


Figure 6.11. A graph of the efficiency obtained by convolving the parallel variant of the Fast One-Ring smoothing filter for scalar fields on discrete manifolds over meshes of different sizes. Lines in blue, increasing in saturation with the size of the mesh, are experiments run using the GPGPU of laptop M with 768 CUDA cores. Lines in red, increasing in saturation with the size of the mesh, are experiments run using the GPGPU of desktop T with 2,560 CUDA cores. The experiments involving acquired 3D-data are specially marked with squares for the flat surface mesh and circles for the university seal.

† See Figure 6.8 for an explanation of the experiment codes.

6.4. Summary

The goal of our experimentation was two-fold: analyzing the filter response on various kinds of 3D-data and evaluating the performance of the parallel variant of the Fast One-Ring smoothing filter algorithm. We began in Section 6.1 by analyzing the filter response when convolving the filter over four different configurations of synthetic 3D-data, each with the Dirac delta function applied as a scalar field.

Some anisotropic behavior was exhibited by the Fast One-Ring smoothing filter response, as it traveled faster along the connections of more distant adjacent points, while isotropic behavior was exhibited when the overall variance in the length of edges was reduced. The filter response also appeared to slow down when convolved over irregular meshes, which was more similar to acquired 3D-data.

Next, in Section 6.2 we analyzed the filter response when convolving over three different examples of acquired 3D-data. This highlighted the way an error in computation propagated across the image with each subsequent convolution, as well as the way the efficacy of the filter diminished, when convolving the filter over a scalar field with very low variance.

Then in Section 6.3, we established the compute times for a multitude of pairs of experiments. These involved both the serial sand parallel algorithm, both acquired and synthetic 3D-data, and four different configurations of hardware.

In the next chapter, we will combine everything that we have discovered through our research, to make conclusions about the Fast One-Ring smoothing filter for scalar fields on discrete manifolds in general, and its parallel variant utilizing GPGPUs.

Chapter 7

Conclusions and Outlook

Figure 6.1 exposed the anisotropic behavior of the filter response which was due information traveling faster along the connections of more-distant adjacent points. However, Figures 6.2 and 6.3 show the filter response exhibiting isotropic behavior as the density of the mesh increased, reducing the overall variance in the length of edges.

Our conclusion then becomes, that designing a filter with isotropic behavior, unlike with pixels of a digital image, will require considerations for not only the filter windows size for inputs, but also bounding the distance information may be written as an output, in order to prevent long, consecutive paths from propagating information faster that other paths.

Figures 6.4 showed how the filter response appears to slow down when convolved over an irregular mesh, and it still remains unclear why the discrepancy exists, as pure speculation, it could be related to low level optimization made at the compiler or processor level which are written to detect and exploit pattern in control and data structures.

In Figure 6.5 an error in computation was seen propagating across the image with each subsequent convolution. However, as that error stems from convolving the filter over "unclean" 3D-data, the solution lies not with the filter, but with first processing the data with another tool, such as the "Automatic Mesh Polishing" provided by the GigaMesh framework. However, graceful error handling would not be impossible to implement for an enhanced version of the the filter.

Then in Figure 6.6, we see that the filter's efficacy diminishes when convolving the filter over a scalar field with very low variance. While this is correct behavior, as an averaging operation should always output a value somewhere within the range of its inputs, this also highlights an opportunity to introduce new user-defined parameters for the filter, for example, to enable the control of convergence tolerance.

In each of the experiments with synthetic data, the filter response applied to the Dirac delta function did behave as a model of the diffusion process would which was expected. Also, when the Fast One-Ring smoothing filter was applied to MSII generated scalar fields on the examples of acquired 3D-data, the features did in fact become smoother. Therefore, it may be concluded that as a smoothing filter, the Fast One-Ring filter is successful.

Next, Figure 6.10 exposes just how powerful using a GPGPU to process a convolutional filter can be, even achieving speedups higher than 200%. As one of our goals was to realize an appreciable speedup, we can conclude that the Fast One-Ring smoothing filter is also successful are scaling better with the growing size of 3D-data.

Conversely, the parallel variant of the Fast One-Ring smoothing filter algorithm only ever reached as high as 15% efficiency. This indicates that vast improvements to the implementation of the algorithm are still yet to be made. An effective first approach would be to optimize how often threads are explicitly synchronized.

In conclusion, as a smoothing filter and at scaling for large mesh sizes, the Fast One-Ring smoothing filter for scalar fields on discrete manifolds is quite successful. However there are still several aspects of the filter remaining, which can be improved upon.

7.1. Future Research

Now that the filter has been implemented in CUDA enhanced C++, the next step could be to implement it using OpelCL in order to include other GPGPUs not manufactured by NVIDIA. Likewise, OpenMP could also be utilized in order to take advantage of under-utilized networks of computers by extending the parallel processing beyond just a single machine.

As mentioned in Section 5.2.1, there exists a design choice which must be made regarding whether or not to store edge lengths twice in order to speedup the algorithm. The decision to not store the reverse lookup table was never explored further, presenting an opportunity for future research.

Footnote 30 mentions how a typo in the original source code led to slightly different filter responses, but an equal convergence value. Therefore, it would be valuable to examine to necessity of calculating the global min edge length, and compare it to assigning the scalar to one instead.

Section 2.2.7 discusses the possibility of MSII processing feature vectors instead of only scalar fields, and we iterate here that future research into the processing of multi-dimensional function values would be valuable.

Currently, there exists a serial version of the Fast One-Ring smoothing filter which uses the median, instead of the mean as the operation for smoothing. A next step would be to also implement that filter with a parallel variant.

Finally, because a circle sector can be described entirely by its interior angle, further research may be able to determine if simply using the interior angles for weighting the function values, instead of also calculating the area of each circle sector, would not only be possible, but also have a positive effect on the performance of the algorithm.

Appendix A

Ratio Approaching Two

In addition to the synthetic meshes featured in Figures 6.1 - 6.4, many others sizes of each type of mesh were also generated to be used in the experiments designed to test compute times, which was discussed in detail in Section 6.3.1. While designing those experiments, an interesting trend uncovered itself as the ratio of the count of triangle faces $|\mathcal{T}|$ over the count of points $|\mathcal{P}|$ tended to approach two as the total count of points in a mesh increased.

$$\lim_{|\mathcal{P}|\to\infty} \frac{|\mathcal{T}|}{|\mathcal{P}|} = 2 \tag{A.1}$$

In fact, the largest two mesh sizes generated, the hexagonal tessellation and quadrisected-square, both with r set to 3,000 and more than 8.1×10^7 and 7.2×10^7 points each, exhibit ratios closer than 5×10^{-4} less than 2, but never at or above.

Figure A.1 illustrates the trend followed by every synthetic mesh we generated, and the three acquired meshes as well. The ratio of the count of faces $|\mathcal{T}|$, divided by the count of points $|\mathcal{P}|$, approaches two as the total count of points in the mesh increases.

Further research led us to the definition of Euler's polyhedron formula, which was proved by Cauchy as early as 1811, and says the counts of points plus the counts of faces minus the counts of edges, will always be two for convex polyhedrons.

$$|\mathcal{P}| + |\mathcal{F}| - |\mathcal{E}| = 2 \tag{A.2}$$

It is therefore our intuition that the trend of the ratio of faces to points approaching two, is related to the diminishing ratio of border-edges to non-border-edges with increasingly dense meshes. Of this, we say no more, except that it is interesting enough to warrant further research.

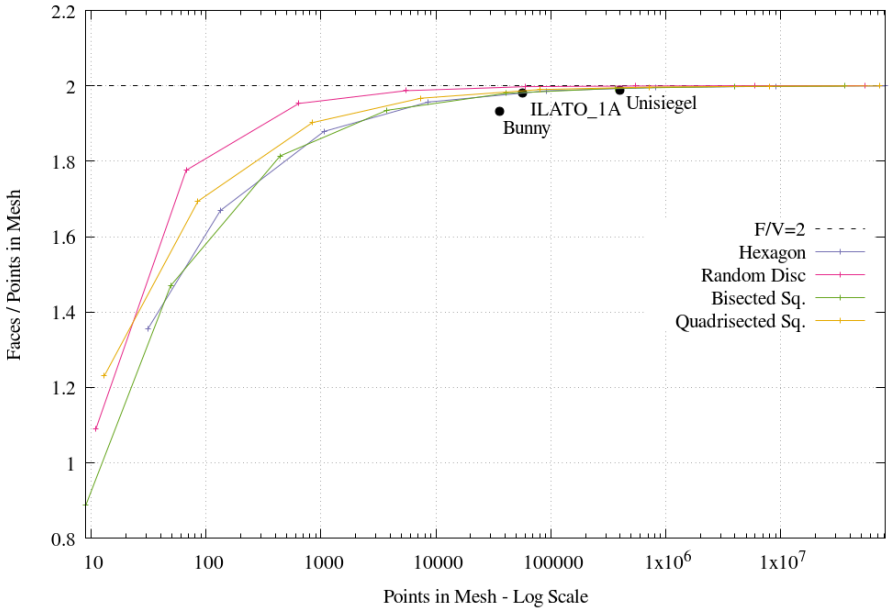


Figure A.1. Ratio of Faces to Points by Increasing Vertex Count

Glossary

adjacency see adjacent.. 36

adjacent synonym of neighboring. A face $\mathbf{t}_i$ is said to be adjacent to another face $\mathbf{t}_j$ if, and only if, they share together the same subset of two points. Similarly, a point $\mathbf{p}_a$ is said to be adjacent to another point $\mathbf{p}_b$ if, and only if, the set $\{\mathbf{p}_a, \mathbf{p}_b\}$ is a subset of at least one face $\mathbf{t} \in \mathcal{T}$. 14, 36, 78

Amdahl's law Given a constant problem size, the limit as the count of processors goes to infinity, the speedup obtained will only approach the inverse of the degree of parallelism exhibited by the algorithm, a problem which can be partially mitagated by only increasing the number of processors in realtion to the size of the problem. 26, 68

bisecting line the line which bisects the central angle of a circle sector. 10

census a total count of all neighbors in all neighborhoods. 43

critical section sections of code containing the instructions that read from or write to volatile memory. 22, 23, 25

degree of parallelism which is the maximum number of operations in an algorithm, which can be executed in parallel. 26, 68

Dirac delta function when applied as a scalar field, a function value of one at the center point, and zero everywhere else. 59, 73, 74

efficiency in regards to parallel processing, efficiency is the ratio of the speedup obtained by a parallel algorithm, divided by the count of processors used to obtain the parallel compute time. It will always be less than or equal to one, and represents the share of the maximal achieveable speedup obtained by the algorithm. 25, 26, 66, 69, 72

explicit synchronization in regards to parallel processing, the situation when all threads are forced to block until every thread is able to reach a designated state, is expensive to computation time. 25

kernel in regards to parallel processing in parallel, a set of instructions to be executed simultaneously on multiple processors using multiple elements from a pool of data. 19

law of cosines . 30

law of sines Let a, b, and c be the lengths of the legs of a triangle opposite angles A, B, and C. Then the law of sines states that $a/\sin A = b/\sin B = c/\sin C = 2R$, where R is the radius of the circumcircle [39]. 32

memory recycling the technique by which, during a convolution, values of a block of memory which will no longer be used until the next convolution, may be overwritten in order to optimize the total memory required to complete the computation. 47

mutable memory memory that may be changed by a specific thread, especially during parallel processing, not requiring a mutex. 49

mutex an abbreviation for "mutual exclusion", a locking mechanism which allows only one single threads to enter a critical section of code at a time. 24, 44

neighboring see adjacent.. 14

program correctness defined by the field of theoretical computer science as existing only when, for each input given, a program produces the expected output. 22

raceccondition in regards to parallel processing, a situation where two or more threads are instructed to perform operations on the same value in volatile memory, so that the program output is contingent on the non-deterministic order in which the operations are actually performed. 23

sector or "circle sector"; the minor sector of a circle defined by its radius and central angle. 9

speedup in regards to a parallel algorithm, speedup is the ratio of the compute time requried for the optimal serial algorithm to run, divided by the compute time required by the parallel algorithm. Speedup will always be less than the number of processors used. Also, as the goal of a parallel algorithm is to complete work faster than its serial variant, it is good for speedup to be a number much greater than one. 25, 26, 66, 68, 69, 72

the principle loop the loop for convolving the Fast One-Ring smoothing filter for scalar fields on discrete manifolds. 36, 39

the stride the size of a block of work intended for a single processor, equal to the problem size divided by the number of available processors. 45, 47

thread also known as "threads of execution", in regards to computer science, it is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system [24]. Threads may be spawned by other threads, creating a heiarchy.. 18

volatile memory memory that may be changed by any thread, especially during parallel processing, requiring thread synchronization before access to ensure program correctness. 22

Acronyms

CPU Central Processing Unit. 3, 20, 71

CUDA Compute Unified Device Architecture. 68, 69, 72

FCGL Forensic Computational Geometry Laboratory. 1

GPGPU General Purpose, Graphics Processing Unit. 2, 18, 20, 66, 68, 69, 71–73

GPU Graphics Processing Unit. 2, 20

ILATO Improving Limited Angle computed Tomography by Optical data integration. x, 65

IWR Interdisciplinary Center for Scientific Computing. 1

MSII the Multi-Scale Integral Invariants filter. 1, 18, 59, 64–66, 74, 75

PNG Portable Network Graphic. 59

SIMD Single Instruction, Multiple Data. 2, 3, 18, 19, 42, 46, 66

Bibliography

[1] *3D Face Recognition with Reconstructed Faces from a Collection of 2D Images: 23rd Iberoamerican Congress, CIARP 2018, Madrid, Spain, November 19-22, 2018, Proceedings*, pages 594–601. 01 2019.

[2] About the khronos group - the khronos group inc From khronos.org, 2019. Last visited on 26/02/2019.

[3] About us - openmp From openmp.org, 2019. Last visited on 26/02/2019.

[4] Accelerated computing — nvidia developer From NVIDIA.com, 2019. Last visited on 26/02/2019.

[5] Data sheet: Quadro p5000 From NVIDIA.com, 2019. Last visited on 26/02/2019.

[6] Geforce gtx 1050 graphics cards — nvidia geforce From NVIDIA.com, 2019. Last visited on 26/02/2019.

[7] Geodesy — science From Britannica.com, 2019. Last visited on 06/02/2019.

[8] Gpgpu.org - about page From GPGPU.org, 2019. Last visited on 26/02/2019.

[9] Passmark intel vs amd cpu benchmarks - high end From cpubenchmark.net, 2019. Last visited on 26/02/2019.

[10] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[11] Manuela Andreoni, Ernesto Londoño, and Lis Moriconi. Double blow to brazil museum: Neglect, then flames. *The New York Times*, Sep. 2018.

[12] Bruce G. Baumgart. A polyhedron representation for computer vision. In *AFIPS '75 Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 589–596. ACM New York, NY, USA, 2010.

[13] Andreas Beyer. *Data Fusion of Surface Meshes and Volumetric Representations*. PhD thesis, Heidelberg University, August 2016.

[14] Andreas Beyer, Hubert Mara, and Susanne Krömker. Ilato project: Fusion of optical surface models and volumetric ct data. 05 2014. arXiv:1404.6583.

[15] Bartosz Bogacz, Michael Gertz, and Hubert Mara. Character retrieval of vectorized cuneiform script. 08 2015.

[16] Bart Braden. The surveyor's area formula. *The College Mathematics Journal*, 17(4):326–337, 1986.

[17] B. Delaunay. Sur la sphère vide. a la mémoire de georges voronoï. *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na*, 6(1):793–800, 1934.

[18] Julie Digne and Carlo de Franchis. The Bilateral Filter for Point Clouds. *Image Processing On Line*, 7:278 – 287, 2017.

[19] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[20] P. R. Halmos. *Naive Set Theory*. Springer Science and Business Media, 2013.

[21] Bernd Jahne. *Digital Image Processing: Concepts, Algorithms, and Scientific Applications*. Springer-Verlag, Berlin, Heidelberg, 4th edition, 1997.

[22] Susanne Krömker and Hubert Mara. Seal of the university of heidelberg - siegel uah accnr 60/2012, 2015. Downloaded from `https://www.gigamesh.eu/pages/downloads/examples/Unisiegel_UAH_Ebay-Siegel_Uniarchiv_HE2066-60_010614_partial_ASCII.ply`.

[23] Stanford University Computer Graphics Laboratory. Stanford bunny, 1994. Downloaded from `http://graphics.stanford.edu/data/3Dscanrep/`.

[24] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sep. 1979.

[25] Stefan Lang. High performance parallel computing. Script from Lecture with same name, given during Summer Semester at Heidelberg University, October 2017.

[26] Hubert Mara. *MultiScale Integral Invariants for Robust Character Extraction from Irregular Polygon Mesh Data*. PhD thesis, Heidelberg University, Heidelberg, Germany, October 2012.

[27] Hubert Mara and Susanne Krömker. Visual computing for archaeological artifacts with integral invariant filters in 3d. In Tobias Schreck, Tim Weyrich, Robert Sablatnig, and Benjamin Stular, editors, *Eurographics Workshop on Graphics and Cultural Heritage*. The Eurographics Association, 2017.

[28] Hubert Mara, Susanne Krömker, Stefan Jakob, and Bernd Breuckmann. Gigamesh and gilgamesh 3d multiscale integral invariant cuneiform character extraction. In Alessandro Artusi, Morwena Joly, Genevieve Lucet, Denis Pitzalis, and Alejandro Ribes, editors, *VAST: International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*. The Eurographics Association, 2010.

[29] Hubert Mara, Bryan Wolfford, and Paul Bayer. *GigaMesh Short Manual*. FCGL (Forensic Computational Geometry Laboratory), INF 205, Heidelberg, Germany, unknown edition, October 2017. Downloaded from `https://www.gigamesh.eu/pages/tutorials/GM_Workflows_25.10.2017.pdf`.

[30] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts with Java*. John Wiley & Sons Software, 8th edition, 2010.

[31] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. volume 94, pages 311–318, 1994.

[32] Bartel Leendert van der Waerden and Christian Marinus Taisbak. Euclid - biography, contributions, & facts From Encyclopedia Britannica, 2019. Last visited on 06/02/2019.

[33] Richard A. Watson. Rene descartes - biography, philosophy, & facts. From Encyclopedia Britannica, 2019. Last visited on 06/02/2019.

[34] Eric W. Weisstein. Aaa theorem. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[35] Eric W. Weisstein. Circular sector. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[36] Eric W. Weisstein. Interpolation. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[37] Eric W. Weisstein. L2-norm. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[38] Eric W. Weisstein. Law of cosines. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[39] Eric W. Weisstein. Law of sines. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[40] Eric W. Weisstein. Linear algebra. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[41] Eric W. Weisstein. Newton's iteration. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[42] Eric W. Weisstein. Topology. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[43] Eric W. Weisstein. Vector addition. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.

[44] Eric W. Weisstein. Vector. From MathWorld—A Wolfram Web Resource. Last visited on 06/02/2019.