A Parallel Map Overlay Algorithm for Vector Data

By
Andras Frankel

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

September 23, 2004

© Copyright
2004, Andras Frankel

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-97452-9

Our file Notre référence

ISBN: 0-612-97452-9

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

PAGINATION ERROR.

ERREUR DE PAGINATION.

TEXT COMPLETE.

LE TEXTE EST COMPLET.

Abstract

Map overlay is one of the fundamental operations of Geographic Information Systems. Given its processor intensive nature, important benefit can be achieved by parallelizing it over several processors. The unequal distribution of data, however, makes load balancing difficult to achieve. In our research, we have used a non-uniform partitioning technique to distribute hierarchical data between the processors. We have achieved parallel efficiencies between 40 to 96% using up to 36 processors.

Moreover, we have developed a floating-point filter to solve numerical problems that affects the line intersection algorithm, a crucial component of the map overlay operation. The speed and robustness of our filter greatly improves the map overlay operation.

Acknowledgments

I would like to thank my supervisors, Dr. Sack and Dr. Nussbaum for all their help to complete this thesis. Their insight and guidance was indispensable for this research.

I would like to thank Environment Canada for allowing me to take time off to work on my Master's Degree, and to the systems administrators' group at the Canadian Meteorological Center for taking over my duties while I was absent.

Nortel Networks supported my quest toward this degree with a graduate scholarship. The research was also supported by NSERC, Sun Microsystems and Stantive Computing.

Finally, I owe numerous thanks to my family for their patience and support during the course of my studies.

Contents

Abstract	iii
Acknowledgments	iv
1 Introduction	1
1.1 Summary of thesis contributions	1
1.2 Structure of the thesis	2
2 Background	3
2.1 Geographic Information Systems (GIS)	3
2.1.1 GIS map	4
2.1.2 GIS operations	6
2.2 Parallel Processing	6
2.2.1 Parallel hardware models	7
2.2.2 Parallel measurements	9
2.2.3 Parallel libraries	11
3 Map Overlay	13
3.1 Problem Description	13
3.2 Difficulties	16
3.2.1 Sequential map overlay	16
3.2.2 Map overlay in parallel	17
3.3 Previous research	19
3.3.1 Non-spatial partition method	20

3.3.2	Uniform size partition methods	20
3.3.3	Variable size partition methods	21
3.4	Parallel domain partitioning method	23
3.4.1	Data structure for domain partitioning method	23
3.4.2	Algorithm to overlay two MFP-trees	27
4	Numerical Issues	31
4.1	Problem description	31
4.2	Difficulties	33
4.3	Previous work	35
4.3.1	Preprocessing of the input data	35
4.3.2	Epsilon tweaking	36
4.3.3	Interval arithmetic	37
4.3.4	Exact arithmetic	37
4.3.5	Adaptive evaluation (filters)	38
4.3.6	Geometric predicates	39
4.4	Description of a line segment intersection filter algorithm	40
4.4.1	Computing the error bound in floating-point computation	41
4.4.2	Plane sweep algorithm	41
4.4.3	Filtering for line intersection test	44
4.5	Discussion of the results	48
5	Polygonization	52
5.1	Problem description	52
5.2	Difficulties	56
5.3	Previous work	60
5.4	Our solution: crossover polygonization in MFP-tree	60
5.4.1	Algorithm to set up crossover polygons in a page	61
5.4.2	Mechanism to complete crossover polygons between pages	64
5.4.3	Implementation of the message passing algorithm	68
5.4.4	Neighbor exchange message passing	72
5.4.5	Numerical issues related to polygonization	73

6	Discussion of results	74
6.1	Testing methodology	74
6.1.1	Setup	74
6.1.2	Hardware profile	75
6.1.3	Input data classification	75
6.2	Results	77
6.2.1	Uniform data	78
6.2.2	Hierarchical data and symmetric maps	79
6.2.3	Hierarchical data and asymmetric maps	81
6.3	Summary of results	82
7	Conclusion and further research	87
7.1	Future work	88
	Bibliography	89

List of Tables

4.1	Running time of our filtering algorithm versus LEDA's general purpose filtering algorithm when using integer based coordinates.	49
4.2	Running time of our filtering algorithm versus LEDA's general purpose filtering a algorithm when using float based coordinates.	50
4.3	The running time of filter and rational operations of overlaying maps with integer coordinates	51
6.1	Input test cases	76
6.2	Running time of r50k assuming best threshold	79
6.3	Running time of r200k assuming best threshold	79
6.4	Running time of r600k assuming best threshold	80
6.5	Running time of Madagascar30k assuming best threshold	81
6.6	Running time of Madagascar100k assuming best threshold	82
6.7	Running time of r250k assuming best threshold	83
6.8	Running time of r500k assuming best threshold	84

List of Figures

2.1	Examples of raster and vector data representation	4
2.2	Schema of MIMD hardware models	9
2.3	Four common interconnection topologies	10
3.1a	Input map 1.	14
3.1b	Input map 2.	14
3.1c	Input map 1 and 2 are superimposed.	14
3.1d	Map resulting from the overlay of map 1 and 2.	14
3.1	Example of a polygon-on-polygon map overlay.	14
3.2	Buffer zones around a point, line and polygon.	16
3.3	MFP domain partitioning for a 2 by 3 processor torus	24
3.4	The corresponding MFP tree	24
3.5	The corresponding Morton ordering of the pages	24
3.6	Possible cases when overlaying two MFP-trees.	29
3.1	Algorithm to merge MFP-trees	30
4.1	The three angle bisectors intersect in a single point	33
4.2	Three angle bisectors intersect at three distinct points	33
4.3	Incorrect Delaunay triangulation due to numerical instability.	34
4.4	Correct Delaunay triangulation.	34
4.5	Example where the transitivity property is not preserved for epsilon tweaking method	36
4.6	Example of a plane sweep algorithm at point e	44
4.7	The data structure of a plane sweep algorithm at point e	45

4.8	Segment of null size	46
4.9	Intersection points are close to each other	46
4.10	Segments are potentially overlapping	46
4.11	Uncertain intersection point	46
4.12	Example of a degenerate case: do S_1 and S_2 intersect?	47
4.13	Example of a degenerate case: do S_3 and S_4 overlap?	47
4.14	Running time LEDA versus our filter algorithm with integer coordinates	49
4.15	Running time LEDA versus our filter algorithm with floating-point coordinates	49
4.16	Time it takes to execute 100,000 floating-point filter and rational op- erations	50
4.17	Rate of applicability of filter with input of integer coordinates	51
4.18	Rate of applicability of filter with input of fractional coordinates . . .	51
5.1a	Polygon set with boundary before polygonization begins	53
5.1b	Polygon after polygonizing node A along segment AE	53
5.1c	Polygon after polygonizing node A along segment AF	53
5.1d	Polygon after polygonizing node D along segment DC	53
5.1	Polygonization example	53
5.2	Polygon 2 is completely surrounded by polygon 1	55
5.3	Page 1's view of arc 1 is different from page 2	55
5.4	A corner polygon	55
5.5	A polygon spanning three pages	55
5.6	Polygons are disjoint inside page	56
5.7	Polygons are joined outside the page.	56
5.1	Algorithm to create borders	57
5.2	Algorithm to polygonize a node under arc.	58
5.3	Algorithm to polygonize around a node.	58
5.4	Algorithm to polygonize a page.	59
5.12	Crossover polygonization: normal case	62
5.13	Crossover polygonization: enveloping case	62

5.14	Crossover polygonization: enveloping case II	62
5.15	Crossover polygonization: segment cutting through page	62
5.16	Crossover polygonization: No crossover arcs	63
5.17	Example of polygonizing crossover arcs of a mix of cutting edge and enveloping cases	64
5.5	Algorithm to to set up crossover polygons in a page.	65
5.19	Examples of level asynchronous message passing mechanism	69
5.20	Examples of crossover polygons that cross and do not cross node's boundary	70
5.21	Examples of crossover polygons in Border-first message passing mech- anism.	71
6.1	A 2 by 2 subdivision of a map.	76
6.2	A 3 by 3 subdivision of a map	76
6.3	Running time of r50k	78
6.4	Running time of r200k	78
6.5	Running time of r600k	78
6.6	Speedup for uniform data distribution	80
6.7	Parallel efficiency for uniform data distribution	80
6.8	Madagascar30k: example of a hierarchical and symmetric overlay. . .	81
6.9	Running time of Madagascar30k	82
6.10	Running time of Madagascar100k	82
6.11	Speedup for hierarchical data distribution with symmetric maps . . .	83
6.12	Parallel efficiency for hierarchical data distribution with symmetric maps	83
6.13	Example of a hierarchical and asymmetric overlay: overlaying R250k.dat (right) with R100k.dat (left)	84
6.14	Running time of r250k	85
6.15	Running time of r500k	85
6.16	Speedup for hierarchical data distribution with asymmetric maps . .	85
6.17	Parallel efficiency for hierarchical data distribution with asymmetric maps	86

Chapter 1

Introduction

1.1 Summary of thesis contributions

The intent of the research described in this thesis is to efficiently and robustly parallelize one of the most basic operations in Geographic Information Systems (GIS): the map overlay paradigm. The research has two main scopes: to implement a robust and efficient solution to numerical problems which is an obstacle to the map overlay operation, and to experimentally evaluate the effectiveness of the Multi-dimensional Fixed Partition (MFP) tree for parallelizing the map overlay algorithm.

The goal of the thesis itself is to document what has been accomplished in the research, namely the effectiveness and applicability of a new filtering algorithm to solve numerical problems and the MFP tree to the parallelization of the map overlay problem. Moreover, the thesis adds theoretical background to the implementation, and explains the design decisions that have been made in the development of the map overlay application. However, it is not the intent of this thesis to serve as a design document to the application. Algorithms and data structures is presented only when it is deemed necessary for the comprehension of the underlying theory.

1.2 Structure of the thesis

- Chapter 2: Background. This chapter gives an overview of Geographic Information Systems and parallel programming in general.
- Chapter 3: Parallel map overlay. This chapter discusses the map overlay problem. It surveys previous approaches to parallelize this paradigm, and lists the problems that have been encountered. It finally presents MFP-trees, our choice for the parallel domain partition.
- Chapter 4: Numerical issues. This chapter explores the numerical problems arising in the implementation of computational geometry algorithms, including the line intersection algorithm, a major component of the map overlay problem. A survey of existing solutions are presented, and a method we have specifically designed for the line intersection algorithm is shown.
- Chapter 5: Polygonization. This chapter deals with polygonization, which is the re-formation of the polygons after the overlay. This chapter deals with the important issue of intercommunication between the processors.
- Chapter 6: Discussion of the results. The aim of the chapter is to present the experimental results of the parallel map overlay implementation.
- Chapter 7: Conclusion and further research.

Chapter 2

Background

This chapter gives an overview of subjects related to this thesis, such as Geographic Information Systems and parallel programming.

2.1 Geographic Information Systems (GIS)

A Geographic Information System is a collection of spatial data and a set of operations that manipulate the data. Typically authors restrict GIS to a computer based system. In their definition, GIS is an ensemble of computer hardware and software designed to perform any geographic operation on spatial data [9]. Others argue that GIS encompasses manual operations and non-digital devices, such as a paper based map [10]. For the purpose of this thesis, we will use the definition that restricts GIS to a digital based system. Regardless of the definition, the operations supported by GIS are essentially the same: store, modify, combine, query and display spatial information.

Data in a geographic information system is made up of geographic or spatial data that gives the geographic location of an object, and attribute or non-spatial data that qualifies the geographic object. An example of non-spatial data includes the amount of precipitation or elevation at a given point.

2.1.1 GIS map

Spatial data can be stored as raster data or vector data models. (See Figure 2.1.) In a raster data model, data is arranged in a uniform grid array, and the attribute is assigned to every element in the grid. In this representation, the precision of the location of an object is limited by the granularity of the grid. In a vector data model, geographic objects are stored at an arbitrary position. The geographic objects are made up of geometric primitives, such as points and line segments, which can be combined into more complex objects. Vector data is more accurate and it is more efficient in terms of storage space, but operations on it tend to be more complex [10].

Figure 2.1: Examples of raster and vector data representation

A map is a representation of a physical area. We define a map as a planar straight-line graph (PSLG). A PSLG is a graph embedded in the plane with straight line edges so that no two edges intersect, except possibly at their endpoints. A PSLG subdivides the plane into regions. It is composed of 0 dimensional vertices, 1 dimensional edges, and 2 dimensional faces. Each face is a simple polygon except for the exterior face. For the purpose of this thesis we call a PSLG a map if each such face has a unique identifier and possibly attributes associated with them.

The main question is how to represent a physical area as a map used by GIS. First of all, the information has to be broken up into two kinds of data: spatial data and attribute data. Spatial data represents the location of an object, such as the location of a house in the city. In GIS, spatial data is usually encoded in two-dimensional

absolute coordinates. Attribute data represents the non-spatial information about an object.

In vector data model, an object can be spatially represented as a *point*, *line* or *area*. A point has no dimension, as it only represents the location of the object. Objects are represented as points if their areas have no importance, such as measuring the temperature at certain locations. In this case, there is no need to attach shape to a thermometer. An other case where point representation is useful is when the map is reduced to such a scale that certain objects' shape cannot be represented. An example is to represent the spatial location of a house on a city map. At certain resolution the shape of the house is reduced to a single point [11].

Lines represent one dimensional objects between two points that have no width. Boundaries, such as the Canada-US border can ideally be represented by a line since it has zero width. Other objects often represented as lines are roads or electrical wires. In this case, the objects loose their width due to scaling. Often, curve lines are approximated as a chain of straight line segments to simplify the operations upon them.

Areas represent two dimensional objects. Lakes, buildings or municipal zones are examples of two dimensional objects. Areas are typically represented as polygons. Road networks are ideally suited to be represented as polygons.

Beside polygons, the other object representation that induces area is Triangulated Irregular Network (TIN). A TIN map is composed exclusively of triangles. TINs are often used to store elevation map. The elevation is measured at a set of points, which are then triangulated, so that each triangle has a constant slope [1].

Attribute data represents the classification of an object. An example would be the classification of a house as a residential compound. In this case, the attribute is said to be *qualitative*. The attribute is *quantitative*, if attributes of different objects can be compared to each other. For example, the value of a house in dollar figure is a quantitative attribute. A quantitative attribute does not have to be numeric. An example of an *ordinal* attribute is when a real estate agent lists the value of a house as cheap, affordable, expensive or luxurious. In case of ordinal data, there is no numerical representation of the attributes, however objects can still be compared

to each other [11].

Often, the attribute data is not stored directly with spatial data, but it is stored separately in a data base.

2.1.2 GIS operations

Perhaps the most basic operation of GIS is the map projection. Its use is to display the information in a spatial format. This operation is analogous to the use of the paper based map. Related operations are: wayfinding, zooming in and out on parts of the map, computing distances between objects, and finding the shortest path between two locations on the map [12].

Another class of GIS operations consists of query operations. It can be a single query, such as locating the hotel called Travelodge on the Ottawa map, or a range query, such as listing all hotels located 2 km from the downtown core. A related operation is called buffering, which covers a distance of a given radius from a point, line or around a polygon [14].

Classification is to subdivide the objects in a GIS based on their attribute. An example is the classification of land area into cultivated, cultivatable and uncultivated regions based on the type of soil.

Finally, the last basic operation of GIS and the one of most relevance to this thesis is overlay. The overlay operation has two functions. First, it can serve as a visual comparison of the input maps. For example, overlaying a map of hotels with a street map to visually illustrate the location of the hotels. Second, it can be used to combine two maps to create a third one [15]. An example of this is to overlay the map of precipitation with a map of elevation to determine if there is an effect of elevation on precipitation. The overlay operation is explained in more detail in Chapter 3.

2.2 Parallel Processing

Parallel processing refers to the simultaneous use of several processors to run a program. The goal of parallel processing is to speed up the execution time of the program.

A number of complex scientific applications do not run fast enough even on the fastest single processor system. Examples include weather forecasting [4], air pollution modeling [5], fluid dynamics [6], biological applications [7] and simulation of a massive agent-based system [8].

There are several reasons why GIS applications benefit from the speedup due to parallelism. First one is when an application would take several hours or even days to run on a single CPU. Since map overlay is one of the most processor intensive operations in GIS, the overlay of maps composed of hundreds of thousands of segments could take a long time to finish [11]. Obviously, such running time would not be tolerated in an operational environment, such as business or government.

Another occasion when a speed is crucial is when the program has to run under an operational time constraint. For example, an electrical utility company manages its power lines through a GIS application. In case of an emergency, they may have a minute to locate and cut the right wire. In this case, the one minute time constraint has to be respected by the application.

Moreover, parallelism allows developers to design better GIS models. A model may have several parameters, the running time of an operation being one of them. Using a parallel architecture, users can scale the model according to their time constraints.

2.2.1 Parallel hardware models

Flynn's taxonomy classifies the hardware used for parallel processing into two broad models [2]. The first one is the *single instruction multiple data* (SIMD) hardware model. It assumes that all the processors in the system execute the same instruction during the same cycle, albeit on a different data stream. The definition of this hardware model is too restrictive, it is used mainly for specialized applications, such as artificial vision. The second model is called *multiple instruction multiple data* (MIMD). It is a less restrictive model that allows the execution of different code on different data at each processor in the system. The MIMD model can be subdivided into four architectural models: parallel vector processors, symmetric multiprocessors, distributed shared memory machines and computer clusters. (See Figure 2.2.)

A parallel vector processor architecture is composed of several vector processors, interconnected by a high speed crossbar switch and accessing shared memory through that switch. A vector processor is capable of executing a single operation on an array of data instead of a single data per cycle. A vector processor is exceptionally powerful, and the communication between processors is very fast. However, this architecture supports just a small number (up to 32) of processors [2].

A symmetric multiprocessor (SMP) machine is composed of several processors, interconnected by a bus¹, which the processors use to access shared memory. It is called symmetric because the processors' clocks are synchronized with equal access to shared resources such as memory and I/O. This architecture is characterized by good load balancing and fast interprocessor communication but it is scalable typically to a custom number of processors. Moreover, it tends to have a high financial cost per processor.

A distributed shared memory machine (also called Shared Virtual Memory or non-uniform memory access - NUMA) is composed of several processors, each having its own memory and interconnected by a custom-designed network. The caches are made coherent to give the impression to the user of a shared-memory machine even though physically it is a distributed memory system. This system has a better scalability than an SMP but has a lower interconnection speed.

A computer cluster is an architecture made up of several single processor systems, called nodes, each having its own memory and local disks and interconnected by a commodity network. A cluster is characterized by low cost per processor and very good scalability. On the negative side, it has low interprocessor speed and load balancing is a challenge normally reserved for the programmer.

Note that systems actually built cannot always fit in one category. A node in a cluster can be a small SMP with typically two to four processors. Also, nodes in a cluster might share disk through a network or a switch.

In the case of distributed memory machines, a number of network topologies can be used to interconnect the processors, such as a ring, mesh, torus or bus. (See Figure 2.2.1.) In case of computer clusters, the network technology tends to be a commodity

¹By a *bus* we mean an interconnection channel that is synchronized with the CPU and memory

Figure 2.2: Schema of MIMD hardware models

network such as Ethernet or Fiber-Channel [2].

2.2.2 Parallel measurements

A straightforward way to measure the gain of running an algorithm in parallel is to calculate the *speedup*.

The speedup $S(p)$ is defined as

$$S(p) = \frac{T_s}{T(p)}$$

where T_s is the time to solve the problem on one processor using the best sequential algorithm and $T(p)$ is the time to execute a parallel version of a solution over p processors. In practice, one rarely writes a sequential program solely to test the parallel speedup against it. It is acceptable to substitute T_s with $T(1)$, the time to

Figure 2.3: Four common interconnection topologies

execute the parallel version of the program for T_s , assuming $T(1) \approx T_s$. The speedup thus can be written as:

$$S(p) = \frac{T(1)}{T(p)}$$

Since an algorithm is made up of a parallelizable part ($T_{parallel}$) and a non-parallelizable part ($T_{sequential}$), the speedup can be also written up as Amdahl's Law:

$$S(p) = \frac{T(1)}{T(p)} = \frac{T_{sequential} + T_{parallel}}{T_{sequential} + T_{parallel}/p}$$

If $S(p) > p$, the program is said to be *superlinear*. Superlinear speedup can occur for two reasons:

1. $T(1)$ is not a good approximation for T_s . The actual de-clustering of the data to be partitioned among the processors contributes more for the performance improvement than the the additional processors themselves. This is an indication that the sequential algorithm should be improved before parallelizing it.

2. The increase in resources other than the CPU, such as main memory, contributes to the speedup. This can happen if the main memory was insufficient when running the program on a single processor and slower virtual memory was used. This is not a good measure for parallel speedup, since the CPU was not the main contributor to the speedup.

Another measure of the gain in parallelizing a program is the *parallel efficiency*.

$$E(p) = \frac{T(1)}{pT(p)}$$

Speedup shows the brute run time improvement, while parallel efficiency is used to show how scalable a program is on a given number of processors [3]. If it is necessary to run a program under certain time constraints then any resources can be used to achieve a the speedup necessary to respect those time constraints. However, when no such urgent time constraints exist, it is preferable to look at the parallel efficiency of a program to determine the cost effectiveness of utilizing more processors.

2.2.3 Parallel libraries

There are several libraries that allow programmers to parallelize their applications. One of the most commonly used is Message Passing Interface (MPI). MPI facilitates the exchange of data between processors. It supports sending and receiving data to and from a single or multiple processors in both synchronous and asynchronous ways. MPI defines synchronization primitives that act on all or a subset of the processors. It supports most built-in scalar and array data structures and allows the user to import any data types into MPI. MPI does not parallelize code nor does it perform load balancing between the processors [16].

Another parallel library is called Parallel Virtual Machine (PVM) [2]. Like MPI, it provides an intercommunication interface among the processes, which ideally reside on different processors. Unlike MPI, it provides its own process management and I/O functions instead of using the underlying operating system. As a result, it supports load balancing on heterogeneous processors. However, its messaging system is not as

flexible or powerful as MPI's.

Both MPI and PVM provide libraries for most common high level languages, such as C, C++ and FORTRAN.

Chapter 3

Map Overlay

This chapter discusses the sequential and parallel map overlay problem, its difficulties and how previous researchers have dealt with it. It analyses several parallel domain partition methods, including Multi-dimensional Fixed Partition trees (MFP-trees) which we decided to implement in our research.

3.1 Problem Description

Map overlay is one of the basic operations of GIS. Its objective is to combine information about two or more maps that are covering the same geographic area into a single map. How maps represent a physical area was explained in Section 2.1.1.

Map overlay of vector data can be defined as the combination into a single planar straight-line graph (PSLG) of two or more PSLGs having a common coordinate system. In particular, the intersection points between the line objects of the input maps need to be computed and the faces of the resulting map need to be identified in order for the resulting map to satisfy the definition of a PSLG. These steps are explained in greater detail in Section 3.2.1.

There are three main types of overlay: point-in-polygon, line-in-polygon and polygon-on-polygon [14]. An example of the point-in-polygon overlay is the overlay of a map showing the location of schools with a map of municipal districts to determine which school belongs to which district. Each school is represented by a

point and the boundaries of each district is traced by a polygon. An example of a line-in-polygon operation is to determine which municipalities does a road cut through, where lines represent roads and polygons represent the boundaries of municipalities.

Polygon-on-polygon overlay is a spatial operation in which two thematic maps represented by polygons are superimposed to create a new map with new polygons. (See Figure 3.1.) An overlay operations in GIS is analogous of placing maps drawn on a transparent medium on top of each other [11]. This operation is the generic overlay of two maps where polygons do not simply represent network, but they induce regions [1]. In this thesis, we concentrate on the polygon-on-polygon overlay operation, which is the most computation intensive, and it would benefit the most from parallelism.

Figure 3.1a: Input map 1.

Figure 3.1b: Input map 2.

Figure 3.1c: Input map 1 and 2 are superimposed.

Figure 3.1d: Map resulting from the overlay of map 1 and 2.

Figure 3.1: Example of a polygon-on-polygon map overlay.

Map overlay is used to combine the information from two maps. An example is

the overlay of the map of precipitation of the Rocky Mountains with that of elevation to find a correlation between precipitation and elevation. The map overlay is often combined with a boolean operation, such as to find the correlation between precipitation and elevation at 2000 meters or higher.

Another example is to find out the effect of flooding on an electrical utility's power lines. A map composed of TINs represents the accumulated precipitation on a terrain, which is overlaid with a map of polygons representing the power grid. The overlay of these two maps tells the electrical utility which lines it has to cut due to flooding, and how to reroute its electric supply through its network.

Map overlay is an important operation in GIS because of its versatility. Many GIS operations can be implemented by map overlay. One example is the range query operation. Assume we want to find out the schools that falls within a fire station's area of responsibility so that the fire chief can visit these schools for fire safety inspection. This operation can be implemented as a range query over a map showing the location of schools. But this operation can also be implemented as an overlay of the school board's map displaying the location of the schools with that of the fire station's map showing its coverage area, followed by an boolean operation to return only the schools that fall within this area.

Another operation that can be implemented by map overlay is buffering. Buffering is used to define spatial proximity [11]. A buffer zone is the area stretching around a point, line or polygon. (See Figure 3.2.) For example, a municipal by-law stipulates that there shall be no building five meters from a railway line in each direction. One way to illustrate the buffer zone is to compute the buffer zone around the railway line, store it in a separate map and the overlay it with the original map to find out if any building lies within the buffer zone violating the municipal by-law.

There are two necessary conditions for overlaying two distinct maps. First, the two maps have to be on the same scale. In the case of vector data, it is simple to zoom one or both maps to put them on the same scale. Secondly, both maps must have a common coordinate system. In our implementation, we do not specifically check for these conditions, but we assume that they are met.

Figure 3.2: Buffer zones around a point, line and polygon.

3.2 Difficulties

3.2.1 Sequential map overlay

Sequential algorithms for the polygon overlay problem have been around since the earliest days of Geographic Information Systems. White's landmark paper sums up the stages of the polygon overlay algorithm: [17]

1. Transform the input coordinates in order for the two maps to span the same geographic area.
2. Find the actual intersection points of the line segments.
3. Polygonize the new line segments.
4. Combine the non-spatial data from the input sets to assign new attributes to the newly created polygon set.

Note that phase 1 is not required since we assume that the maps to be overlaid have identical coordinate systems. Phase 2 is considered to be the most time consuming part of the algorithm [26] [18]. The intersection points can be computed by brute force, where every polygon from one set is compared for intersection with every polygon of the other set. Assuming the input consists of n objects (polygons),

each made up on average of m segments, the total running time using the brute force technique is $O(n^2m^2)$.

One method to reduce the running time of the intersection algorithm is called the *bounding box* technique. It consists of drawing a bounding rectangle around each polygon, and testing two polygons for intersection only if the two polygons' bounding rectangles overlap [22].

A class of more refined techniques to compute the intersection points of polygons is called *spatial sorting* techniques. One is to sort polygons along the horizontal or vertical axis and then fold the two lists of sorted polygons together for intersection computation. Another one is to superimpose a spatial grid over the input data sets and then use brute force technique to compute the intersection points [18]. Parsing the polygons into the grid is done in linear time, while the complexity of finding the intersections in each cell of the grid is reduced to the number of polygons inside the cell.

Another technique to reduce the complexity of the intersection algorithm is to employ a plane sweep technique. It is the technique that we have selected for our implementation. It is described in more detail in Chapter 4.

Polygonization is the operation of taking the line segments resulting from phase 2, and reestablish an explicit topological ordering upon them. If the input consists of TINs, it might not be necessary to polygonize the result, and it might be more effective to discard the line segments and use the intersection points to triangulate the result. To keep the program as general as possible, we decided to implement the polygonization operation. Polygonization is explained in detail in Chapter 5.

3.2.2 Map overlay in parallel

The map overlay problem is parallelized by running phases 2, 3 and 4 above in parallel on several processors, while reducing the overhead due to partitioning of data and message passing to a minimum.

The stages of the sequential map overlay algorithm are slightly modified for the parallel algorithm:

1. Partition the polygons or their edges among the processors.
2. Find the intersection points of each polygonal arc in parallel.
3. Polygonize the edges owned by each processor in parallel.
4. Exchange data between processors so that polygons that cross a processor's area are assembled.
5. Assign new non-spatial attributes to the newly created polygon set from the input polygon sets in parallel.

A key step, and thus a key design decision in parallelizing the map overlay algorithm is the partitioning of data between processors. Note that a partitioning scheme that does not take the spatial nature of the problem into consideration is problematic. If polygons are partitioned between processors in an ad-hoc way without regard to the topology then it is likely to generate a large number of messages when the polygons are overlaid, as there is no guarantee that two polygons that are to be overlaid are owned by the same processor. It is imperative therefore that a processor owns the same portion of the map in both input data sets.

Spatial partitioning of the input involves some difficulties as well. Polygons that cross the processor's boundary represent a special case that can only be resolved through processor interaction. There are three possible scenarios to resolve these problems:

- To assign a whole polygon to a single processor. - However, two crossover polygons that are overlaid and are spatially related may be assigned to two different processors.
- To duplicate the borderline polygon to every processor whose area it crosses. - This technique creates redundant work for the processors. Moreover, it does not eliminate message passing completely. Processors owning neighboring polygons still have to exchange messages in order for each polygon to know the identity of its neighbors.

- To clip the polygons that cross processor boundaries. - This ensures that no two processors labor redundantly, however extra work is needed for clipping the polygons and exchanging messages for polygonization.

Another difficulty encountered by the spatial subdivision of data is to guarantee load balancing of the processors. Speedup is seriously compromised when one or more processors remain idle during the execution. This is due to the fact that there is no guarantee that two processors covering equal sized areas have the same number of intersection points. In fact, in the case of *hierarchical data*, parts of a map can be more densely populated than others. A static partition of data equally between processors offers no guarantees on load balancing.

One attempt to achieve load balancing is called dynamic load allocation, where processors are allocated work on the fly when they become idle. In the case of the map overlay problem, it could be achieved by subdividing the maps into small, equal sized pages that are dynamically allocated to processors on demand. Polygonization however would be very complicated as it would be difficult to determine the owner processor of a particular polygon. Moreover, in general, dynamic load balancing also places considerable overhead [27] in allocating the work to the processors. Another load balancing technique is to partition the input data into smaller pages, and then statically allocate a certain number of pages to every processor. By increasing the number of partitions, this increases the probability that every processor has a similar number of workload without offering any guarantees for hierarchical data. This method is used in [20]. A third approach, that we have used in our implementation is to partition the input hierarchically between the processors. This technique attempts a static load balancing scheme but guarantees that every page has a pre-defined maximum number of polygons. This method is explained in more detail in Section 3.4.1.

3.3 Previous research

There have been several individuals and research groups who implemented all or parts of the polygon overlay problem in parallel. This section gives an overview, outlining

the methods employed and discussing the results they achieved.

3.3.1 Non-spatial partition method

Wang (University of Guelph) has taken a unique approach to polygon overlay. He did not attempt to subdivide the input data set spatially. Instead, he parallelized Denis White's Band Algorithm [17]. This algorithm subdivides each polygon into chains of line segments, where each chain stretches from between two endpoints of two polygon's boundary. The chains are sorted according to their minimum y values. Then the chains of the data sets are compared whether they can potentially intersect based on their maximum and minimum y values. For those that can potentially intersect, a brute force method is used to pairwise compare them to find (or exclude) the intersection point. This is the part of the algorithm that Wang has parallelized.

Wang's work stops at finding the intersection points, which he estimates to be 80% of the work of the complete overlay problem. He has experimentally achieved an efficiency of 95% over the sequential algorithm. However his algorithm was implemented on a shared memory architecture with a limited number - nine - of processors [26].

3.3.2 Uniform size partition methods

In [18], Franklin et al. advocate the use of a uniform grid technique. The input edges are parsed into the grid and the intersection of the edges with the cell boundary is computed. The grid cells are distributed to the processors in such a way to avoid giving two adjacent cells to the same processor. They first devised the uniform grid technique to reduce the complexity of the line intersection algorithm in sequential computation.

The line intersections are computed by a brute force technique inside each cell. Based on his input data, Franklin et al. decided against using a hierarchical method to partition the input maps.

For the intersection of the line segments, Franklin et al. have achieved a parallel efficiency of 67% using 15 processors. A technique to polygonize the output edges is

outlined, but it is not implemented.

In [19], Franklin and Narayanaswami computed the boolean overlay of two general polygons¹ using the uniform grid technique. The number of cells in the grid depends on the average length of the edges in the input. The partitioning of the edges into the cells of the grid is done in parallel by all the processors using a shared-memory data structure. After the line intersections have been computed in parallel, polygonization and boolean operations on the resulting edges are done locally on each cell. Using a shared memory 15-processor parallel computer, this algorithm has achieved a parallel efficiency of 70%.

The uniform grid technique was also used in Hopkins, Healey and Waugh's implementation of the parallel map overlay algorithm on a distributed memory architecture. One processor is responsible for reading the input data from disk, and passing the segments to the other processors. Each processor determines whether each segment belongs to the cell that it covers. If it does, the processor retains it, otherwise the processor passes it along to the next processor. If a segment crosses the cell boundary, it is clipped and the outlying segment is passed along. When all segments in the input set were passed along, the processor that was reading the input data broadcasts an end-of-file message. As soon as a processor has received all its arcs from one data set, it can start computing the intersection points while receiving more segments at the same time. Finding the intersection points inside the cell is done by a brute force method. The polygons that are crossing cell boundaries are re-assembled by message passing between neighboring cells [21].

In the intersection computation part of the algorithm, they achieved a speedup efficiency of 70% using 81 processors [20].

3.3.3 Variable size partition methods

Harding, Hopkins, Healey and Dowers have also developed a variable partitioning method for the map overlay paradigm [22]. The domain is subdivided into strips that stretch the full length of the input set, while the width of the strip is variable to

¹two *polygons* not polygon sets

the density of the polygons. The distribution of polygons into strips is done simultaneously for both maps, so a strip contains polygons from each set. An advantage of using strips to partition the data is that each strip has at most two neighbors, thereby reducing the amount of intercommunication between the processors.

One processor is responsible for allocating the polygons to the strips, and storing the strips on disk until a processor is available to process it. This is an on-demand dynamic load allocation where a processor requests a strip to process as soon as it becomes idle. The intersection computation inside each strip is done by plane sweep. A very detailed design of the algorithm is presented in [22] but an implementation is not discussed.

Shekhar et al. developed a density sensitive map subdivision method that is suitable for hierarchical data. In [27] they claim that this method is suitable for several GIS related applications, including map overlay. However, they chose to implement range queries over a polygonal set. Range query is an overlay of a rectangle over a polygonal set. The intersection computation is of course much less expensive than in the general map overlay case.

The basis of their spatial subdivision technique is the grid file [24]. Unlike Franklin's uniform grid [18], the grid file's cell sizes vary according to the number of elements in it. The input data is parsed into the grid file. The polygons that cross a cell boundary are not clipped, but the cell has knowledge about the whole polygon. The cells are allocated to processors in a round robin fashion which is called quasi-dynamic: a subset of the available processors, called leaders, are statically allocated a fixed number of cells. Then these leaders dynamically allocate their cells to other processors in their group when they become idle.

Because of the reduced scope of the problem formulation (range query) no processor communication is necessary to polygonize the result. For this problem, Shekhar et al. have achieved a parallel efficiency of 85%.

Ching [25] has developed a method to dynamically partition the input data between processors so that the load is about equal for each processor. It consists of sweeping a horizontal line on the input maps to find the optimal bipartition of the input graph. In [25], he claims that his technique is efficient on a shared memory

computer with two to four processors. He did not implement the polygon overlay algorithm.

Hecker used an MFP tree [31] to partition the input maps. Once the line segments of the polygons are subdivided between the processors, he used Bentley and Ottmann's Line Sweep algorithm [29], implemented in the LEDA library [34], to find the line segment intersections. In order to polygonize in parallel, each page was polygonized internally. Then, the algorithm descended from the root of the MFP tree, and every polygon that crossed a page boundary was assigned to a predetermined processor to assemble it. Hecker achieved a parallel efficiency of 70% [32].

3.4 Parallel domain partitioning method

3.4.1 Data structure for domain partitioning method

One of the first decisions in designing a parallel map overlay algorithm is the selection of the data structure that will subdivide and store the data that will be distributed to the processors. Nussbaum in [31] has identified several criteria for a domain partitioning method:

- Support spatial subdivision. - As it has been argued previously, the spatial nature of the problem calls for a spatial partitioning method.
- Reduce data redundancy. - The duplication of data between processors should be minimized.
- Adapt the data structure to different hardware architectures. - Since there is a large variety of parallel hardware platforms having different features such as number of processors, type of memory model and type of interconnection, it is preferable to use a data structure that is equally well performing on as many types of hardware as possible.
- Support data clustering. - It cannot be assumed that the input data is uniformly distributed throughout the domain. Therefore it is preferable to use a data structure that supports a variable size partitioning method.

The significance of the partitioning of the MFP-tree is related to the parallel hardware where the program is implemented. First, we are going to concentrate on a torus architecture (see Section 2.2.1), because the MFP-tree was specifically designed with the torus architecture in mind. Later, we are going to extend MFP-trees to other architectures as well.

A parallel architecture that has a torus as an interconnecting topology is a distributed memory architecture where each processor is directly connected with exactly four other processors. Schematically, the processors are aligned in a rectangular or mesh-like shape, where each processor is connected to its left, right, above and below neighbor. Each processor that is schematically located at the "edge" of the mesh is connected with a processor on the symmetric edge of the mesh. Processors that are directly connected are called neighbors. See Figure 2.2.1 for a schema of a torus architecture.

Assume the hardware is a torus architecture with k processors along the x-axis and l processors along the y-axis for a total of $k \times l$ processors. Then the problem's input domain is subdivided and parsed into a two-dimensional MFP-tree with $k \times l$ subdivisions at every level of the MFP-tree. These pages can then be mapped to the processors of the torus computer.

The pages of the MFP-tree are indexed using a Morton ordering space filling curve (see Figure 3.5). This enables the pages to be stored in a linear fashion and to query the pages using a binary search.

The nodes of the MFP-tree are mapped to the processors of a torus computer in the following way. At every odd level of the MFP-tree, a *forward mapping scheme* is applied, and at every even level a *reverse mapping scheme* is used. Reverse mapping is a backward mapping of the nodes to the same processors. Note that an internal node is assigned to a processor as well even though the node's area of responsibility was divided between its children. This mapping scheme ensures that pages that are adjacent to each other are either held at the same processor or at two adjacent processors in the torus [31].

To explain forward and backward mapping, we have to introduce some notations first. Every page in a two-dimensional MFP-tree can be identified with respect to

its parent page by the tuples (c_x, c_y) , where c_x and c_y represent the page's position in the X and Y direction. Similarly, every processor can be represented by the tuple (p_x, p_y) , where p_x and p_y represent the processors position on the torus. Moreover, let P_x and P_y be the number of processors in the X and Y directions of the torus. Then forward mapping is defined as the mapping of page (c_x, c_y) to processor $((p_x + c_x)P_x, (p_y + c_y)P_y)$, and backward mapping to processor $((p_x - c_x)P_x, (p_y - c_y)P_y)$. (See Figure 3.3.)

Next, we will show how MFP-trees can be used on other architectures than the torus computer. The ring architecture can be viewed as a one dimensional torus. E.g., a p processor ring can be mapped to a 1 by p MFP-tree with the same method as described above. Note that such an MFP-tree partitions the map into strips that stretch along the whole length of the domain.

The MFP-tree can also be used with a computer cluster architecture, where the interconnecting network is a bus architecture. Assume that the cluster has p processors. Then a two dimensional MFP-tree is used to partition the domain, with $k = \lfloor \sqrt{p} \rfloor$ partitions along the x-axis and $l = \lceil \sqrt{p} \rceil$ partitions along the y-axis. The same mapping scheme can be reused as for the torus computer, with the processors of the cluster mapped to a virtual torus.

Although MFP-tree is generic for d -dimension, all GIS data is inherently two-dimensional. As a consequence, GIS input cannot be mapped efficiently on a hypercube architecture of dimension three or more. The simplest technique is to map the pages to the hypercube's processors the same way as for a cluster.

The mapping of the MFP-tree to the processors is a static load allocation. Because each page of the MFP-tree contains at most *threshold* objects the load balancing is expected to be better than with a uniform grid technique.

Let us review how the MFP-tree relates to the criteria we have defined above:

- Support spatial subdivision. - Each page of the MFP-tree stores neighboring objects, therefore satisfying the spatial criteria.
- Reduce data redundancy. - Only crossover polygons or TINs are replicated among the pages.

- Adapt the data structure to different hardware architecture. - The MFP-tree's subdivision and the mapping of the pages to the processors is suitable for a wide variety of parallel hardware.
- Support data clustering. - Unlike uniform grid techniques, MFP-tree pages are subdivided according to the number of elements they contain. The number of elements each page holds is within a *threshold* amount from each other.
- Support operation pipelining. - Except for the objects that cross the page boundary, pages are independent of each other. Therefore operations can be performed on pages independent of the status of other pages. For example, as soon as the intersection points inside a page have been found, the polygonization of the segments in the page can commence. There are obvious limitations to the pipelining feature though to the necessity of the processors to cooperate with the crossover objects [31].

Note that the shape of the MFP-tree depends on the particular parallel hardware for the number of subdivisions along each axis and on the density of the data in the input map for the height of the MFP-tree. Input data can be parsed into an MFP-tree and then stored on each processor's local disk. The following section demonstrates how to overlay two non-identical MFP-trees corresponding to two different input maps.

3.4.2 Algorithm to overlay two MFP-trees

Using MFP-trees as a partitioning method leads to a challenge that does not exist with uniform grid techniques. Each input map has its own MFP-tree, whose structure might not be identical to that of another MFP-tree. It is because two maps, even if they represent the same geographic area, might not have the same density in the same location. As a result, the same location in two input maps might be covered by two pages of different sizes from its respective MFP-tree.

The following algorithm allows to overlay two MFP-trees:

Before proceeding to overlay, each processor has two lists of pages that are its share of the MFP-trees. The lists are organized using the Morton ordering mentioned previously. As the processor traverses the MFP-trees linearly, four possible cases can occur:

- Case 1: The two pages (page 1 from MFP tree 1 and page 2 from MFP tree 2) overlap completely.
- Case 2: One page covers the other one. - The larger page covers smaller pages owned by other processors.
- Case 3: The pages are disjoint. - The page with the lower Morton order has a corresponding but larger page that another processor owns.
- Case 4: One of the MFP tree has exhausted its pages. - It has to receive a page from another processor.

These are the four possible cases that can occur when overlaying two MFP trees. Algorithm 3.1 describes how to handle each of these cases. See also Figure 3.6.

Figure 3.6: Possible cases when overlaying two MFP-trees.

```

PROC overlay
LOCAL: MFPtree1, MFPtree2: MFP trees corresponding to input maps 1 and 2.
page1 := GetNextPage(MFPtree1)
page2 := GetNextPage(MFPtree2)
while page1 != NIL OR page2 != NIL do
  NOTE MFPtree1 or MFPtree2 is not empty
  if page1 and page2 overlap exactly then
    OverlayPages(page1, page2)
    page1 := GetNextPage(MFPtree1)
    page2 := GetNextPage(MFPtree2)
  else if page1 covers page2 then
    Broadcast(page1)
    repeat
      OverlayPages(page1, page2)
      page2 := GetNextPage(MFPtree2)
    until page1 covers page2
    page1 := GetNextPage(MFPtree1)
  else if page2 covers page1 then
    NOTE the reverse of "page1 covers page2"
  else if page1 and page2 are disjoint and page1 has a lower Morton order than
    page2 then
      page3 := ReceivePage()
      repeat
        OverlayPages(page1, page3)
        page1 := GetNextPage(MFPtree1)
      until page3 covers page1
    else if page1 and page2 are disjoint and page2 has a lower Morton order than
    page1 then
      NOTE the reverse of ".. and page1 has a lower morton order than page2"
    else if page1 == NIL then
      page3 := ReceivePage()
      repeat
        OverlayPages(page2, page3)
        page2 := GetNextPage(MFPtree2)
      until page3 covers page2
    else if page2 == NIL then
      NOTE the reverse of "page1 == NIL"
    end if
  end while
end while

```

Algorithm 3.1: Algorithm to merge MFP-trees

Chapter 4

Numerical Issues

This chapter explores the numerical problems arising in the implementation of geometric algorithms, why they are difficult to resolve and what solutions have been previously proposed. It presents our own solution for numerical problems in the line intersection algorithm.¹

The following convention is used in this chapter. A *fractional* number refer to a non-integer rational, independent of how it is represented by the computer. A *floating-point* value denote the common approximate representation of fractions by the computer. It is explained in Section 4.1. A *rational* number refer to the exact representation of a fraction as a numerator and denominator pairs. It is explained in Section 4.3.4.

4.1 Problem description

Most modern computer architectures are using a floating-point representation to handle fractions. A floating point number of size m is a series of m bits where the first bit is the sign, the next e bits are the exponent and the tailing l bits the mantissa (or fraction). According to the widely used IEEE 754-1985 standard, the value of the floating point number is $(-1)^s * 1.mantissa * 2^{exponent}$. The exponent gives the magnitude of the number, and the mantissa provides the precision (or the number of

¹Work in this chapter has also appeared in [28].

significant binary digits[39]).

Since floating point operations are implemented in hardware, these operations are fast. On the other hand, because they support finite precision only, certain fractions have to be rounded in order to be represented as a floating point number. Note that even if the precision of the mantissa could be user-defined, certain *decimal* rational numbers (like $1/10$), cannot be represented as a *binary* floating point number without rounding.

Two problems arise from the use of finite precision representations in geometrical or mathematical problems: inaccuracy and robustness problems (program is unstable or it outputs erroneous results.) In some situations inaccuracy is acceptable if it falls under a pre-defined threshold. In the case of real-world geometric problems, a certain degree of inaccuracy in the result is acceptable, as the input may not be accurate either. If the error bound on the input dwarfs the error bound due to floating-point computation then the latter can be discarded.

Robustness problems arising from the use of floating point numbers are more dangerous. Computer programs are written under the assumption that geometric axioms hold true for any input. However, because of rounding errors in floating point computation, it cannot be guaranteed that these axioms hold true under all conditions. For example, the equation of a line is given by $Ax + By + C = 0$. Then $EAx + EBy + EC = 0$, $E \neq 0$, is the equation of the same line. However, if constants A, B, C and E are represented as floating point numbers, these equations may not represent the same line.

A similar situation arises when computing the incenter of a triangle. The three angle bisectors of a triangle meet at a single point, called the incenter [40]. However, when the incenter is computed using floating point arithmetic, it could yield three different incenters (see Figures 4.1, 4.2.)

Geometric inaccuracy not only impacts the correctness of the coordinates of the spatial objects but also the topological relationship between objects and other topological information which is an integral part of spatial information systems and geographic information systems. In many cases topological information is extremely important. For example, in a GIS for electrical utility, it is not only important to

Figure 4.1: The three angle bisectors intersect in a single point

Figure 4.2: Under floating point computation, the three angle bisectors might intersect in three distinct points

know where a man-hole is but also to know on what side of the road it is located. Incorrect geometrical computations can lead to incorrect topological information in the databases and thus impact future topological queries (e.g., containment, intersect).

Here we introduce the notion of *topological correctness*. An algorithm produces results that are topologically correct if it is able to detect all the intersections correctly and distinguish which line segment intersects another line segment in a given point. However, inaccuracy is acceptable on the geographic location of the intersection points.

4.2 Difficulties

The limitation of floating-point computation accuracies impacts the robustness of a geometric algorithm in the following ways:

- program crashes - The reason for crashes is that programs enter into states that are illegal, i.e., into states that could never be reached if the topology were geometrically correct. One or more incorrect decisions may result in incorrect and unanticipated branching in the program's flow.
- program produces incorrect results - Wrong decisions are made which produce an output, but the output is incorrect. E.g., Shewchuk [43] observed that precision error problems may produce a "Delaunay triangulation" which has intersecting edges (see Figures 4.3 and 4.4.) (For further details on this example

see [43].) Sometimes an incorrect output is easily seen, while other times the output "looks right", but is in fact not correct. This motivates also the study of efficient algorithms which verify the correctness of an output independent on the construction methodology used to create it.

- program goes into infinite loop - Due to contradicting decisions algorithms may jump between two or more program states indefinitely.

Figure 4.3: Incorrect Delaunay triangulation due to numerical instability. *Reproduced from [43]*

Figure 4.4: Correct Delaunay triangulation. *Reproduced from [43]*

Numerical analysis techniques do not seem to be applicable to geometric calculations. In the implementation of a geometric algorithm, it is not realistic to assume that numerical problems won't happen. There are several techniques that deal with numerical robustness issues, which are surveyed in Section 4.3. We have devised and implemented a technique that deals with numerical issues specifically in the line intersection algorithm, which is presented in Section 4.4.

Another issue in geometric computation is *degeneracy*. A degenerate case is an exceptional case in the input that can stall the program if the algorithm was not designed to handle it. For example, four cocircular points are considered degenerate in computing the Voronoi Diagram, because it can lead to two different outputs. Similarly, two overlapping line segments are degenerate in the line segment intersection

problem, because it has an infinite number of intersection points. Note that numerical stability and degeneracy are two different issues. Although both can cause the program to crash, degeneracy can be viewed as a problem of the implementation, while numerical stability is inherent to the input.

A straightforward way to deal with degeneracy is to handle every degenerate case that can occur in the input. An alternative technique is to pre-process the data to eliminate degenerate cases from the input with a large probability. Preprocessing steps are discussed in Section 4.3.1.

4.3 Previous work

Over the years, a number of techniques have been developed to deal with numerical issues. In [28], we proposed to classify them into three broad categories. All these techniques have to at least guarantee the robustness of the program. The first category encompasses techniques that guarantee only the robustness of the algorithm. Epsilon tweaking and interval arithmetic fits in this first category. The second category includes techniques that guarantee the robustness of the algorithm and the correctness of the underlying topology but not the accuracy of the result. Floating-point filter algorithms make up the bulk of the techniques in this category. Finally, the third category encompasses techniques that guarantee both robustness of the program, the correctness of the topology and the exactness of the numerical results. Exact arithmetic and geometric predicates fit in this category.

4.3.1 Preprocessing of the input data

A simple way of dealing with degenerate cases is to add a small random number to every input coordinate, where the random number is generated independently for every input. A more deterministic way of achieving the same effect is to rotate every coordinate in the input set by a constant degree, execute the desired operations on the resulting plane and then rotate the plane back by the same degree. These techniques eliminate certain degenerate cases with high probabilities, such as two segments being

collinear. However, they do not solve the numerical problems themselves [41].

4.3.2 Epsilon tweaking

An often used and simple technique to alleviate the side effect of floating-point arithmetic is called *epsilon tweaking*. Two points, p_1, p_2 , are considered identical as long as they are less than a distance of ϵ apart ($\|p_2 - p_1\| < \epsilon$). The value of ϵ is either set to some arbitrarily predetermined value or found by trial-and-error: substitute an arbitrary small value into ϵ , and run the program. If the program does not crash then, accept the error level of ϵ as it is. Otherwise, repeat the execution of the program with a slightly larger value of ϵ [41]. Epsilon tweaking neither guarantees the accuracy of the result nor the accuracy of the underlying topology. Moreover, it is not well suited to large data sets, where re-computing results with different values of ϵ is not only very time consuming, but it can also produce different outputs depending on the value of ϵ . Setting ϵ to some arbitrary small value independent of the magnitude of the operands or of the type of operation performed is another common problem.

Moreover, epsilon tweaking does not preserve the transitivity property. Assume that points a and b are within ϵ from each other, and point b and c are also within ϵ of each other. Therefore points a and b are considered the same point and point b and c are considered the same point, however points a and c should not be considered the same point because they could be of a distance greater than ϵ from each other. (See Figure 4.5.)

Figure 4.5: Example where the transitivity property is not preserved for epsilon tweaking method

4.3.3 Interval arithmetic

A real value in the computation is represented by two floating point numbers: the upper and lower bounds. Arithmetic operations are performed on all endpoints, and they have to be rounded up or down after the operation. The upper and lower bounds initially depend on the precision of the floating point representation only, but it can widen once the operation is executed [38].

The advantage of this technique is that the error bound is computed and not just "guessed" as in the epsilon tweaking method. The drawback is that the error bound can grow large in the case of cascading computation, when a computed value serves as an input in a subsequent computation. The error bound can therefore dwarf the computed value, making the result useless.

4.3.4 Exact arithmetic

A fail-proof method to deal with rational numerical problems is called *exact arithmetic*. Every number is stored as a rational number, which is made of a pair of integers: the numerator and the denominator. These integers can typically be represented in a custom amount of bits. Operations on these numbers, called rational arithmetic, support addition, subtraction, multiplication and division. Two rational numbers can be compared exactly. Rational arithmetic in the line intersection algorithm guarantees the accuracy of the intersection points and hence that of the topology. The drawback of this approach is that it is very slow in practice. After each operation, the greatest common divisor (GCD) of the numerator and the denominator has to be computed, since otherwise the numerator and denominator risk overflowing. GCD is implemented in software, and it is CPU intensive to compute [41].

Achieving robustness may be expensive, cases have been reported where robustness costs a factor of 100 in speed (see e.g., [44]) or even an initial slow-factor of 10,000 when floating-point arithmetic is replaced by exact rational arithmetic (see [45]).

4.3.5 Adaptive evaluation (filters)

A floating point filter can be described by the saying "Why compute something that is never used?" [41]. Translated into computational geometry terms, approximative computation can be used to compute a result, but comparison between two values has to be exact. Therefore, every result is computed using fast, floating-point arithmetic. An error bound is computed dynamically with every computation. (Refer to Section 4.4.1 how to compute the error bound.) Two values can be compared using their error bound. If the two values lie beyond their error bound, the comparison was definitive. Otherwise, the filter is not applicable, and a more definitive technique (such as exact arithmetic) is required to compute the result. Hence, for every computation, a list of pointers to its operands has to be maintained in order to be able to recompute its value would the filter be inapplicable [36].

Instead of computing the error bound dynamically with every computation, a static upper bound on the error bound can be applied. A static error is an upper bound that can be computed from the computer supplied floating point number's precision and the size of the input data [36]. Dynamic computation gives smaller error bounds, hence doesn't require the use of rational arithmetic as often as the static error bound. However there is an overhead of determining the error bound at every computation.

There is a plane sweep algorithm implemented in the LEDA geometric library [37]. It uses floating-point filters. The error bound is computed semi-statically. According to the LEDA documentation [36], the plane sweep algorithm uses a data type called **Float**. **Float** does the computation with normal floating point numbers, but computes the error bound as well. Every **Float** encapsulates a directed acyclic graph (DAG) that points to the operands that were used to compute this value. Comparisons between two **Floats** are done by subtracting one value from the other and comparing the result to 0. If the result is less then the error bound, the DAG is used to retrieve its operands and exact arithmetic is applied to recompute the values. This filtering is not done at the algorithm's level, but it is encapsulated inside LEDA's **rational** data type [37].

LEDA has a separate data type, called **real**, which uses adaptive evaluation for

real arithmetic. It supports addition, subtraction, multiplication, division and square root operations. Each real number is stored as a floating point number with arbitrary precision. Every **real** number holds a DAG to every operand and operation that is used to derive its value. The value is only computed when a branching decision has to be made. It is therefore sufficient to compute the sign of an expression. For example, assume we want to know whether $A < B$, where A and B are two real numbers. Then it is sufficient to know the sign of the expression $A - B$.

The computation is done by approximating the value of the expression over a number of precision iteratively until the approximation is close enough to the value of the expression. By "close enough", we mean that the approximation A of expression E has the same sign as E . A is an acceptable approximation of E if $|A - E| < sep(E)/2$, where $sep(E)$ is the *separation bound* of E .

The separation bound of an expression can be computed the following way. Let $deg(E)$ be the number of square root operations in the expression and $b(E)$ be the value of E if every subtraction operator is replaced by an addition. Then $sep(E) \geq b(E)^{1-deg(E)}$ [37].

Another technique that is being investigated is the use of interval arithmetic (see e.g., [46, 47] or surveys such as [49, 50]) in combination with multiple-precision floating point arithmetic. For interesting alternate approaches, including the Core Library, see also [48, 51].

4.3.6 Geometric predicates

The notion of *geometric predicates* is a new technique pioneered by Liotta et al. [53]. Any arithmetic expression that supports only addition, subtraction, multiplication and division can be expressed as a multivariate polynomial of its input values. Most basic operation in a computational geometry problem can be expressed as a geometric predicate of the polynomial that is needed to compute it. The degree of an algorithm is the maximum degree of the polynomial of its geometric computations. In [53], Liotta et al. have shown that the number of bits of precision needed to compare two geometric predicates depends on the number of bits of precision of the input and the

degree of the polynomial of the predicate. One can therefore compute the results over sufficient number of bits of precision for the algorithm to function correctly.

Using this technique, Boissonat [52] designed a red-blue line segment intersection algorithm that computes the results correctly with twice as much precision required for the computation as for the input values.

The advantage of geometric predicates is that all the computation can be done using floating-point arithmetic. One does not have to determine the error bound, and using a sufficient number of bits for the computation, guarantees correctness and robustness of the algorithm. The drawback is that most hardware does not supply floating-point precision on demand. One has to either resort to software-designed floating point numbers with the time penalty that it entails; or, like Boissonat, reduce the degree of the algorithm so that the computation can use the 32 and 64 bit precisions the hardware provides.

4.4 Description of a line segment intersection filter algorithm

This section presents our solution to numerical problems arising in the implementation of the line intersection algorithm. It is a floating-point filtering technique that is inspired by the general purpose filter algorithm presented in Section 4.3.5. Like the general purpose filter, our filtering aims to compute with floating-point arithmetic, switching to exact arithmetic when necessary. Unlike the general purpose filter, it is specifically tailored to the line intersection algorithm. It filters explicitly for degenerate cases of the line intersection algorithm. By deterministically handling the degenerate cases, the algorithm guarantees the correctness of the underlying topology which in turns guarantees numerical stability. Before explaining the algorithm in detail, let us first review how to compute the error bound in floating-point computation and how a plane sweep algorithm works. The filtering algorithm is explained in detail in Section 4.4.3 below.

4.4.1 Computing the error bound in floating-point computation

The error bound of any given real number stored as a floating-point number depends on the precision of the floating point representation provided by the compiler, as well as the magnitude of the number.

Machine epsilon is defined as the smallest number that can be added or subtracted from a unit value one in order to change its value [38]. Machine epsilon depends solely on the precision of the floating-point number. Its value is

$$\epsilon = 2^{-p}$$

where p is the precision of the floating point representation. To compute the error bound on any given number stored as a floating-point number, one has to multiply its value by the machine epsilon. Thus, the error bound e_x of the value x , is given by the formula [38]

$$e_x = \epsilon * x = x * 2^{-p}$$

The error bound for the addition or subtraction of two numbers a and b is

$$e_{a+b} = e_{a-b} = (e_a + e_b) * (1 + \epsilon)$$

The error bound on the multiplication/division of two numbers is

$$e_{a*b} = e_{a/b} = (e_a * e_b) * (1 + \epsilon^2)$$

[37].

4.4.2 Plane sweep algorithm

The plane sweep algorithm is an efficient technique to solve geometric problems such as finding the intersections of line segments [29, 30], and computing triangulations and Voronoi diagrams [1]. In a plane sweep algorithm, an imaginary line is swept

(say) horizontally from left to right through the domain. At every position of the sweep line, the line segments intersecting the sweep line are called *active*. The points of interest for the sweep line are the endpoints of the line segments, which are referred to as *event points*. Note that the status of the algorithm only changes at these event points [1].

The plane sweep algorithm can be employed to efficiently compute the intersection points of the line segment intersection algorithm. Recall that the complexity of the line segment intersection algorithm of n line segments using the brute force method is $O(n^2)$, even though the number of resulting intersection points in particular problem instances may be $o(n^2)$. The plane sweep algorithm can be applied to reduce the complexity of the line segment intersection algorithm by utilizing the fact that only active line segments may intersect, and that the status of the algorithm is changing only at event points.

The optimal plane sweep algorithm for the line segment intersection problem was written by Balaban [30]. Its running time is $O(n \log n + k)$, where n is the number of input line segments and k is the number of intersection points found. Because of its implementation complexity, we decided to implement the more elegant and widely known Bentley-Ottmann plane sweep algorithm, whose running time is $(n \log n + k \log n)$ [29]. This algorithm is described below.

In the Bentley-Ottmann algorithm, in addition to the endpoints of segments, the event points also include the intersection points as they are being discovered.

The algorithm uses two main data structures: a queue of event points (event queue) and a list of the active line segments. Both the event queue and the list of active segments should be implemented as a balanced binary search tree for fast query, insert and delete operations. For horizontal sweep, the event queue is sorted horizontally based on the event point, while the active line segments are ordered vertically based on their intersection point with the sweep line.

The following is our outline of the algorithm. At the beginning, the event queue contains the sorted leftmost endpoints of the line segments. The sweep line sweeps through the domain, halting at every event point. The following list enumerates the actions that are taken at every event point.

- Event point is the leftmost endpoint of a segment.
 - Insert this segment into the list of active line segments.
 - Insert the right endpoint of this segment to the event queue.
 - Determine whether this segment intersects with either of its neighbors in the list of active line segments. These two neighbors are segments that lie just above and below the current segment intersecting the sweep line. If they do intersect, insert the intersection point to the event queue.
- Event point is an intersection point of two segments.
 - Swap the two segments that form this intersection point in the list of active line segments.
 - For both segments, verify whether they intersect with the segment above and below in the list of active line segments and to the right of the sweep line. If any of them do, insert the intersection points to the event queue as well.
- Event point is the rightmost endpoint of a segment.
 - Delete this segment from the list of active line segments.
 - Determine whether the segment above and below of this segment in the list of active line segments intersect each other to the right of the sweep line. If they do, insert their intersection point to the event queue.

The above list assumes straightforward cases only. Degenerate cases such as vertical segments, overlapping segments, intersection at an endpoint of a segment or more than two segments intersecting at the same point are described along with our technique to handle numerical issues in Section 4.4.3 below. (See also example on Figures 4.6 and 4.7.)

In our implementation, the event queue is implemented as Standard Template Library's (STL) data structure **multiset**. Multiset is a balanced binary search tree

which can store duplicate keys. Each event point is made up of its coordinate, type of event (leftmost/rightmost endpoint, intersection point) and a pointer to its corresponding segment in the list of active line segments. For each event point, the corresponding line segment in the list of active line segment can be found in constant time. Note that for an intersection, an event point corresponding to each line segment is inserted to the event queue.

The list of active line segments is implemented as a red-black tree. A red-black tree is a balanced binary search tree [42], [35]. Each active line segment has a pointer to its upper and lower neighbor in the list. These neighbors represent segments that lie just above and below the line segment intersecting the sweep line. A segment's neighbor can be found in constant time. Moreover, two line segments that are neighbors can be swapped in constant time.

Figure 4.6: Example of a plane sweep algorithm at point e

4.4.3 Filtering for line intersection test

Our approach to numerical issues is inspired by the floating-point filter methodology. We are taking advantage of the fact that we design a filter for a specific algorithm, namely the line intersection algorithm. Since this operation is of fundamental importance it merits attention. Knowledge about the application of the filter allows us to

Figure 4.7: The data structures of the previous example at point e . Note that e is inserted twice to the BST of events.

filter for specific instances arising during the execution of the algorithm, instead of filtering for every comparison performed by the program.

In general-purpose filtering techniques, every computation is performed with floating point arithmetic and an error bound is computed dynamically. A list is maintained to the operands that were used to compute the value. Every time two values are compared, error bounds are used to determine the ordering of the two values.

In the line intersection algorithm, which is presented here, we identify the computationally "hard" cases. These cases correspond to degenerate cases of the line segments intersection algorithm. The degenerate cases are: null line segment, overlapping line segments, intersection near a vertex, and, overlapping intersection points [54]. In "near" and "close range" we mean that it is inside the points error bound. (See Section 4.4.1 how to compute the error bound.) Just like in the general filter method, every intersection is computed using floating-point arithmetic, along with an error bound on the intersection point. Right after every computed intersection point, we test whether it corresponds to one of the above degenerate cases. If it does, we use exact arithmetic to recompute the intersection.

The following is the list of degenerate cases that the algorithm examines and how they are tested.

Figure 4.8: Segment of null size

Figure 4.10: Segments are potentially overlapping

Figure 4.9: Intersection points are close to each other

Figure 4.11: Uncertain intersection point

1. Null segment: the size of the line segment is less than ϵ . (see Figure 4.8).
2. Overlapping intersection points: If the intersection point is within an error bound of another intersection or endpoint (see Figure 4.9.)
3. Segments intersect near their endpoints: the endpoint of one of the segments is at a distance of less than ϵ from the computed intersection point. (see Figure 4.11).
4. Overlapping line segments: if all of the following three cases are true (see Figure 4.10):
 - The two segments have overlapping X ranges.
 - The slopes of the line segments are within their respective error bounds.
 - The two lines Y-intercepts are within their respective error bounds.

Although this filter technique requires a large number of comparisons it does not impose additional computation. The slopes and the Y-intercepts have to be computed to determine the intersection point regardless of the filter. Moreover, the slope and the y-intercept are computed only once per segment and applied for every intersection point involving this segment.

Figure 4.12 gives an example where the filter is non-applicable, and rational arithmetic is needed to determine a degenerate case. Take line segment S_1 : $(0, 1)-(2, 1.0002)$ and S_2 : $(0, 0)-(1.0008, 1.0001)$. Using floating-point arithmetic, the algorithm determines that S_1 and S_2 intersect at $(1.0008, 1.0001)$. Using rational arithmetic, it determines that S_1 and S_2 do not intersect, as point $(1.0008, 1.0001)$ is to the right of S_1 .

Figure 4.13 gives an example where two segments S_3 : $(0,1)-(2,3)$ and S_4 : $(0.5,1.25)-(1,1.5)$ are tested for overlap. Their X-ranges are overlapping. Using floating-point arithmetic, their slopes are both computed to .5 and their y-intercepts are both 1. Using rational arithmetic, slopes and y-intercepts are recomputed and they are indeed the same. Since S_3 completely covers S_4 , S_4 is removed from further computation.

Figure 4.12: Example of a degenerate case: do S_1 and S_2 intersect?

Figure 4.13: Example of a degenerate case: do S_3 and S_4 overlap?

The advantage of our method over the general purpose filter is that it does not have to maintain a list of operands at every computation. Moreover, the performance of the filter algorithm depends on the number of occurrences of the degenerate cases in the input and not on the number of computations which is related to the precision of the floating-point filter representation.

4.5 Discussion of the results

We implemented our line intersection filtering algorithm, and tested it experimentally against LEDA's implementation of the general purpose floating point filter [36, 37]. LEDA was selected because it is considered an industry and academic state of the art library for geometric computation.

We evaluated the merits of using our filtering technique as follows. We compared two commonly used representations of coordinate systems, integer and fractional, both with high precision (64 bits floating point representation which achieves 15 decimal points accuracy). Namely, we compared integer geometry and floating point geometry. For testing we used four different data sets. Three of the data sets are "real life" data sets: africa, madagascar and australia and one data set was randomly generated. The data sets contain 50,000, 20,000, 30,000 and 50,000 line segments respectively. The intersection experiments were conducted as follows. Each of the data set, e.g., africa, was duplicated as dup(africa). The line segments in each duplicated data set, e.g., dup(africa) were translated by ∂x and ∂y . Then each pair of data sets were tested for intersection, e.g., africa and dup(africa). The algorithm was executed several time for each pair for each of the data types (integer or float) and an averaging of the timing results was computed. The test computer was a SUN SPARC single 440 MHz CPU with 512 MB RAM running Solaris. All floating point arithmetic was using 64 bits (double precision) operands.

Figures 4.14 and 4.15 show the performance of our filtering algorithm's versus the general purpose filtering algorithm implemented in LEDA. It shows that our algorithm improves on the performance of LEDA by a factor of two to four if the input is composed of integer coordinates, and a factor of 10 to 20 in case of fractional coordinates (also see Tables 4.1 and 4.2).

We further investigated the cause of such a significant improvement in performance. First the "cost" of a filtering operation versus exact arithmetic using rational operation was evaluated. We observed that our filtering operation is much faster than a rational operation of LEDA by a factor of 20 (see Figure 4.16 and Table 4.3). Table 4.3 shows, for each data set test, the number of times that filtering operations

Figure 4.14: Running time LEDA versus our filter algorithm with integer coordinates

Figure 4.15: Running time LEDA versus our filter algorithm with floating-point coordinates

Input file	size	Running time our filter (s)	Running time LEDA (s)	Improvement of filter over LEDA (%)
random	50000	29	116	75
africa	20000	7.7	20	62
madagascar	30000	32	50	36
australia	50000	53	89	40

Table 4.1: Running time of our filtering algorithm versus LEDA's general purpose filtering algorithm when using integer based coordinates.

were called and the time to execute the filtering operations. The table also shows the number of times that the filter invoked rational arithmetic and the amount of time it took to execute these invocations. At last the table shows the time to execute 100,000 operations for each and the percentage in saving from using the filter.

The goal of the filter operations is to reduce the number of times that a rational arithmetic operation is invoked. The filter operation invokes a rational arithmetic operation only when the filter detects that the output falls into one of the four criteria (presented in Section 4.4.3) in order to accurately compute the intersection point.

Our filtering operation is much faster in comparison to a rational arithmetic operation of LEDA and achieves a performance ratio of 1:20. Thus, if one considers counting only float and rational operations and ignores other parts of the intersection algorithm, then, by reducing the number of rational operations such that the ratio

Input file	size	Running time our filter (s)	Running time LEDA (s)	Improvement of filter over LEDA (%)
random	50000	20	382	95
africa	20000	5.1	49	90
madagascar	30000	12	173	93
australia	50000	19	290	93

Table 4.2: Running time of our filtering algorithm versus LEDA's general purpose filtering algorithm when using float based coordinates.

Figure 4.16: Time it takes to execute 100,000 floating-point filter and rational operations

between rational operations and filter operations is better than 19:20 (19 rational operations for 20 filter operations), our algorithm would achieve superior performance to that of LEDA. In reality, one cannot ignore other parts of the intersection algorithm, e.g., the maintenance of the event queue, and thus our filtering algorithm must achieve a better ratio than 19:20.

In all the cases that we tested (fractional and non-fractional coordinates) the "savings" from using the filter is much better than 19:20 (see graphs in Figures 4.17 and 4.18). In the case of non-fractional coordinate system the ratio was between 1:2 and 1:10 which implies that the filter achieves between 50-90% efficiency. In the case of fractional coordinates, the filter achieves an efficiency rate of around 100%.

The following reasons can account for the performance improvement of the filter algorithm vis-a-vis LEDA:

- The notion of filtering: Computation done with floating-point arithmetic and

Input file	Number filter operations	Time for filter ops (s)	Number rational ops	Time rational ops (s)	Time per 100,000 filter ops (s)	Time per 100,000 rational ops (s)	Filter appli- cability rate (%)
random	349000	4.1	33000	7.5	1.17	22.73	90.5
africa	71000	0.7	14000	3.3	0.99	23.57	80.3
madagascar	158000	1.5	82000	18	0.95	21.95	48.1
australia	281000	2.7	137000	31	0.96	22.63	51.3
average	-	-	-	-	1.02	22.72	-

Table 4.3: The running time of filter and rational operations of overlaying maps with integer coordinates

Figure 4.17: Rate of applicability of filter with input of integer coordinates

Figure 4.18: Rate of applicability of filter with input of fractional coordinates

recomputing the values with rational arithmetic only if necessary is faster than straight exact arithmetic. Refer to table 4.3 for comparison of performance between filter and rational operations.

- Our algorithm filters specifically for degenerate cases in the input and for every comparison that LEDA's algorithm makes.
- Our algorithm always uses hardware supplied floating point operands. LEDA's filtering line intersection algorithm requires its own *rational* data type for input. That means the input data has to be translated into LEDA's data type before the algorithm can run, which significantly increases the running time.

Chapter 5

Polygonization

This chapter presents polygonization, the operation to construct face information along the arcs. Both sequential and parallel polygonization is presented. This chapter deals with the issue of intercommunication between processors.

5.1 Problem description

The result of the overlay operation is a list of line segments and their intersection points. The input to the polygonization operations is this set of line segments intersecting only at their endpoints, and the output is a list of polygons with their boundary chains and neighborhood information. This operation is useful in the following two situations:

- If the attribute data is associated with the face of the polygon.
- It is necessary to follow up the overlay with boolean operations.

If the input maps were made up of polygons or TINs, the output will consist of polygons. In this case, polygonization must be performed.

A polygon set is a planar map, meaning arcs can only cross each other at their endpoints. The line segment intersection algorithm presented in the previous chapter insures that this is the case. Each polygon is said to be *simple*, meaning its boundary chain does not cross itself, not even on the vertices.

Figure 5.1a: Polygon set with boundary before polygonization begins

Figure 5.1b: Polygon after polygonizing node A along segment AE

Figure 5.1c: Polygon after polygonizing node A along segment AF

Figure 5.1d: Polygon after polygonizing node D along segment DC

Figure 5.1: Polygonization example

In our design, a polygon set is made up of polygons (faces), vertices (nodes) and edges (arcs). Each edge has pointers to its two vertices (node1 and node2) as well as a left and a right face id. Left face is the area lying to the left of the edge going from node1 to node2, while the right face is on the opposite side of the edge. Each node contains its geometric coordinates, as well as a list of unordered edges that are adjacent to it. There is one polygon structure per face, which holds pointers to its boundary edges.

Polygonization of the segments that lie inside the page is referred to as in-page polygonization. The algorithm does the following. First, chart the boundary of the polygon set that lies inside the page (see Algorithm 5.1.) Then start with the leftmost node that lies on the boundary. List all the node's arcs that lie inside the page in clockwise order. Then walk around each of these arcs in counterclockwise order, assigning the same face id to the inner side of each arc, until it reaches back to the starting node. Thus the algorithm walked around one polygon. While walking through the polygon, add its nodes to a queue. Proceed the same way with the remaining arcs of the starting node in clockwise order. Repeat recursively the same procedure with the nodes that are in the queue. Figure 5.1 gives a graphical example of the polygonization algorithm.

In order to ensure that the same face is not going to be polygonized twice, once all the faces around a node have been polygonized, the node is marked "done." At any subsequent polygonization of a face, if one of its boundary nodes has already been marked "done", it means that this face has already been polygonized. In this case, bail out of the polygonization of the face without adding the node marked "done" to the queue. The algorithm terminates once the queue is empty.

The problem of the algorithm described above is that it is going to miss "holes." A hole is single polygon or a cluster of polygons that is not connected to the remaining polygon set (see Figure 5.2.) To polygonize holes, it is sufficient to find the nodes that have not been marked "done." Then apply the same procedure as described previously (i.e.: chart the boundary, walk around the edges, etc.)

Note that the algorithm would have been faster if the arcs around each node were ordered clockwise.

Algorithms 5.2, 5.3 and 5.4 illustrate in-page polygonization.

The fact that in case of parallel polygonization the overlaid segments have been broken up into pages complicates the polygonization process, since certain polygons are crossing page boundaries. These polygons may have some of their arcs stored at different processors. These polygons will be henceforth referred to as crossover polygons and the process of polygonizing them as crossover polygonization. Similarly, polygons that are stored entirely on the same page will be called in-page polygons. The subsequent sections will look at crossover polygonization in greater detail.

Figure 5.2: Polygon 2 is completely surrounded by polygon 1

Figure 5.3: Page 1's view of arc 1 is different from page 2

Figure 5.4: A corner polygon

Figure 5.5: A polygon spanning three pages

5.2 Difficulties

The main challenge in crossover polygonization is re-assembling the polygons that cross page boundaries just like if they were on the same page. Crossover polygons have to keep the same face information and arc ids on every page they appear. Each page however does not need to know the whole extent of the crossover polygon. It is sufficient if it has correct information about the arcs that are crossing it.

There are several difficulties in this process. First of all, a page does not have complete information about an arc that is crossing over its area. This arc might have a different arc id on a neighboring page, and it might even have a different length if it has been intersected by a second arc in the neighboring page. (See Figure 5.3.)

A crossover polygon might span more than two pages, making a simple exchange paradigm between neighbors insufficient. An example is shown in Figure 5.5. Although such a case is rare when the input consists of TINs, the "corner polygon" occurs (see Figure 5.4.)

Figure 5.6: Polygon 1 and 2 are disjoint in page 1

Figure 5.7: ... but they are joined together in page 2 to form one polygon

A final difficulty lies with two or more disconnected crossover polygons of a page being joined together in the same polygon in another page. (see Figures 5.6 and 5.7.) It is possible to have several layers of crossover polygons enveloping one another.

Numerical issues, which were discussed previously for the line intersection algorithm, can also affect the crossover polygonization procedure. How to deal with it is explained in Section 5.4.5.


```

PROC: Create_border
  REQUIRE: start_nodes: vector to contain all nodes inside the page
  LOCAL: current_node: node we are visiting, start_node: starting node of a loop,
  current_arc: the arc following current_node
  for all current_node in start_nodes do
    start_node := current_node
    if current_node only leads to outside the page then
      erase current_node from start_nodes
      mark current_node OUT_BOUND
    next_for
  end if
  current_arc := first arc in page of current_node
  Mark respective face of curr_arc as external
  current_node := other node of current_arc
  while current_node is different from start_node do
    if curr_node does not have any arc other than current_arc that leads inside
    the page then
      erase current_node from start_nodes
      mark current_node as OUT_BOUND
      current_node := get other node of current arc
    next_while
  end if
  curr_arc := next in page arc of current_node in clockwise order from current_arc
  Mark respective face of current_arc as external
  current_node := other node of current_arc
end while
end for
RETURN start_nodes
ENSURES: start_nodes contains the nodes on the boundary of the polygon set.

```

Algorithm 5.1: Algorithm to create borders

```

PROC: polygonize_node_arc
  REQUIRE: start_node, current_arc: create polygon along node start_node and ad-
  jacent arc current_arc
  nodesStatus: keeps track, for every node, whether it already has been visited.
  undone_nodes: stack of nodes to be polygonized
  LOCAL: arc_chain: list of arcs in the polygon, current_node: node currently visited
  current_node := node adjacent to current_arc other than start_node
  while current_node is different from start_node do
    if current_node is out of page then
      RETURN
    else if nodesStatus of current_node is VISITED then
      RETURN
    else
      insert current_arc to arc_chain
      current_arc := next counter clockwise arc around current_node
      current_node := other node of current_arc
    end if
    Insert current_arc to arc_chain
  end while
  RETURN arc_chain

```

Algorithm 5.2: Algorithm to polygonize a node under arc.

```

PROC: polygonize_node
  REQUIRE: current_node: create polygons around current_node
  nodesStatus: keeps track, for every node, whether it already has been visited.
  current_arc := first arc in page of current_node
  start_arc := current_arc
  repeat
    if current_arc is not a boundary arc then
      call polygonize_node_arc with current_node and current_arc
    end if
    current_arc := next clockwise in page arc of current_node
  until curr_arc is different from start_arc

```

Algorithm 5.3: Algorithm to polygonize around a node.

PROC: polygonize_page
REQUIRES: a page structure with list of nodes and nodes pointing to adjacent arcs
LOCAL: starting_nodes: list of starting (leftmost) nodes for every polygon set in page, initialized to empty)
LOCAL: nodesStatus: a map of the status of each node, initialized to UNKNOWN
LOCAL: boundary_arcs: for every arc, it keeps track whether it is a boundary arc.
undone_nodes: stack of nodes to be polygonized (initialized as empty)
call Create_borders
for all start_node in starting_nodes **do**
 push start_node to undone_nodes
 while undone_nodes is not empty **do**
 current_node := pop undone_nodes
 call polygonize_node on current_node
 end while
end for
ENSURES: all polygons that do not cross page boundaries are recorded

Algorithm 5.4: Algorithm to polygonize a page.

5.3 Previous work

In [21], Waugh and Hopkins distributed polygons between the cells of a uniform grid. The crossover polygons are neither clipped nor shared between cells, but each crossover polygon is exclusively owned by a single cell. The cell that owns the crossover polygon initiates the linking (polygonization) by sending the relevant information (crossover arcs and left and right face id) to the neighboring cells. If the received crossover polygon does not cross to a third cell, the receiving cell completes the polygon by sending the updated information back to the initiating cell. If the crossover polygon crosses into subsequent cells from the receiving cell, the receiving cell sends it to these cells as well. Under Waugh and Hopkins model, it is sufficient if one cell (the one owning the crossover polygon) has complete knowledge of the polygon.

In [22], Harding et al. divided the domain into strips that stretch the full length of the input set. The width of a strip is set dynamically, so every strip is wider than the widest polygon. This ensures that every polygon crosses at most two strips. In [23], Mineter gives a thorough overview of the cases in connecting crossover polygons across strip boundaries.

In [32], Hecker used MFP-trees (see Section 3.4.1) to partition data between processors. His parallel polygonization algorithm uses two distinct communication phases: one at the arc level to ensure that each page has the correct arc id and arc endpoints, and another one to link the polygons that cross page boundaries. Pages that own an endpoint of a crossover segment are the initiators, while pages that an arc cuts without having an endpoint there are the receivers. It can take several rounds of communication to update an arc that cuts more than two pages.

5.4 Our solution: crossover polygonization in MFP-tree

This section presents the design of our crossover polygonization algorithm. First, there is a description of how to set up the crossover polygons on each page. Then

there is a generic description of the messaging mechanism so that crossover polygons are assembled. Finally, several concrete message passing mechanisms are discussed.

5.4.1 Algorithm to set up crossover polygons in a page

The following is an overview of the cases that can happen in crossover polygonization.

- In the *normal case*, all crossover polygons in a page are joined through their crossover arcs and have a single open side to the border of the page. (See Figure 5.12)
- In the *enveloping case*, the outer rim of one or more crossover polygons becomes the inner rim (border) of another polygon. The enveloping case differs from the normal case by having at least one crossover polygon that has more than one side open to a page boundary. (See Figure 5.13.)

Note that this case can also appear in a way that two or more crossover polygons appear to be disjoint, i.e. there is a gap in between them. It is possible, however, to view the "gap" as just another crossover polygon. Therefore this case satisfies the definition for the enveloping case. (See Figure 5.14.)

- The *cutting edge case* arises when one segment cuts through the page dividing it into two parts. This case can occur with either the normal or the enveloping case. (See Figure 5.15.)
- In the *single crossover polygon case*, there is a lone crossover polygon that may or may not envelop in-page polygons. This case is characterized by the lack of crossover arcs. (See Figure 5.16.)

Figure 5.12: Crossover polygonization: normal case

Figure 5.13: Crossover polygonization: enveloping case

Figure 5.14: Crossover polygonization: enveloping case II

Figure 5.15: Crossover polygonization: segment cutting through page

Figure 5.16: Crossover polygonization: No crossover arcs

In the first phase of crossover polygonization, all the information known by the page about the crossover polygon are stored together in a structure. This is composed of the crossover arcs, the arcs that are contained within the page, and the side(s) of the page that are being crossed by this crossover polygon.

The essence of the algorithm is to go through all arcs that cross the page boundary. Taking each arc, get the next counter-clockwise arc through the node located inside the page, and add it to the arc chain. Continue until it finds an arc that crosses the page boundary. This arc "exits" the page, meaning the incomplete polygonal chain stops there.

To handle the "cutting edge case", it is sufficient to determine whether the first arc crosses the page boundary twice.

To handle the "enveloping case", the algorithm has to determine whether there are arcs crossing the page boundary between "entry arc" and "exit arc". In order to accomplish this, all arcs crossing a page boundary are inserted into a binary search tree in order of their crossing point of a boundary. If there are arcs between "entry arc" and "exit arc" the algorithm has to walk backwards from "exit arc", inserting all arcs into the incomplete polygonal chain until "entry arc" is reached.

Algorithm 5.5 illustrates how to set up crossover polygons in a page.

Figure 5.17: Example of polygonizing crossover arcs of a mix of cutting edge and envelopping cases

5.4.2 Mechanism to complete crossover polygons between pages

The goal of crossover polygonization is that at the end of the process each page has the correct information with respect to the following criteria:

- All arcs in the page have the correct endpoints, even if the endpoint is out of the page.
- All arcs have two face ids, representing the two polygons it borders. These face ids are unique and consistent throughout the map.
- All arcs have unique and consistent arc ids throughout the map.

In order to fulfill the above criteria, it is sufficient in each page to send for every crossover polygon only the arcs that are crossing the page. That gives enough topological information for the crossover polygons to be assembled. (See Lemma 5.4.3 below.) The arcs belonging to a crossover polygon that do not cross page boundaries


```

PROC: setup_crossover_polygons
LOCAL: crossover_arcs: a binary search tree of all arcs crossing the page bound-
aries. It is ordered by the point the arc crosses the boundary, starting from the
lower left corner and going clockwise.
LOCAL: arc_chain: one for each crossover polygon.
for all arcs in page do
  if arc crosses page boundary then
    Insert arc into crossover_arcs by crossing point.
    Note: The arc can be inserted twice if it cuts through the page.
  end if
end for
for all start_arc in crossover_arcs do
  if start_arc crosses two boundaries of the page then
    NOTE: special case: cutting edge
    current_arc := start_arc's second crossing in crossover_arcs
    Insert current_arc to arc_chain
  else
    repeat
      current_arc := start_arc
      Insert current_arc to arc_chain
      current_arc := clockwise next arc of current_arc
    until current_arc does not crosses page boundary
  end if
  next_arc := arc following start_arc in crossover_arcs
  while current_arc is different from next_arc do
    Note: envelopping case
    current_arc := previous arc of current_arc in crossover_arcs
    if current_arc crosses two boundaries of the page then
      Note: cutting arc under an envelopping crossover polygon
      Insert current_arc to arc_chain
    else
      repeat
        current_arc := start_arc
        Insert current_arc to arc_chain
        current_arc := clockwise next arc of current_arc
      until current_arc does not crosses page boundary
    end if
  end while
  Insert arc_chain to MFP page
end for

```

Algorithm 5.5: Algorithm to to set up crossover polygons in a page.

do not have to be transmitted. The receiving page that has complete information about the crossover polygon can make the final decision about the face id and the arc id, which it then has to communicate to the sending page(s).

One way to broadcast this information throughout the domain is for each page to send its crossover polygons to its neighboring pages in a predetermined direction. For example, each page could send its crossover polygons up and right, and receive crossover polygons from below and left. However, this algorithm is very counter-intuitive. It is not straightforward to treat crossover polygons that are crossing more than two pages. Moreover, in an MFP tree [31], a page can have an arbitrary number of neighbors. Unless the page shares a common parent node with its neighbor, it doesn't even have exact knowledge of its neighbor's identity.

Our algorithm attempts to use the nature of the MFP tree to exchange crossover polygons in an efficient way. The order and direction for pages to send the crossover polygon information is the following. Each processor goes through its local pages (the leafs of the MFP tree) in Morton order. Each page sends its crossover polygons up in the MFP tree to its parent node. If the parent node resides on the same processor as the leaf node then the transfer happens in memory only. The processor that receives the crossover polygons creates an empty page at the parent node and inserts the crossover polygons into this page. Inserting the pages involves assembling the crossover polygons that are sharing common sides. Crossover polygons that are still crossing the parent page boundaries are sent upward the next level for assembly. This process continues until all crossover polygons are assembled at the root node of the MFP tree.

The root node then initiates sending back the information about the crossover polygons, so each page at the end has the correct information with respect to the criteria outlined above.

We assume that n is the number of crossover polygons and h is the height of the MFP tree. Then the total number of crossover polygons to be exchanged in this algorithm is $O(hn)$.

Theorem 5.4.1. *Our algorithm fulfills the three criteria defined above, i.e. all faces have unique and consistent face ids and all arcs have correct endpoints, and unique*

and consistent arc ids.

The following proofs are needed to prove Theorem 5.4.1. It has to be proven that (1) there will be no conflicting information about face ids and arc ids, (2) this algorithm terminates without deadlock, and (3) the crossover polygons are assembled correctly with respect to topology.

Lemma 5.4.1. *There are no conflicting information about face ids and arc ids.*

Proof. For every arc or polygon crossing the page boundary, the processor owning the parent node makes the decision about the arc and face id. Each processor has an exclusive range of numbers to use as ids, in order to avoid duplication. When partitioning the domain between the processors, this processor is responsible to propagate this information down to the leaves of the MFP tree. $\square$

Lemma 5.4.2. *The algorithm terminates without deadlock.*

Proof. Each processor processes all of its pages (the leaf nodes of the MFP tree.) This involves sending up all its crossover polygons before waiting to receive for crossover polygons at the intermediate node level. The same methodology is repeated with the intermediate nodes at every level: each processor treats all the intermediate nodes that can receive polygons before going to the next level. $\square$

Definition 5.4.1. *maximal polygon: A polygon is called maximal if it cannot be merged.*

Lemma 5.4.3. *The crossover polygons are assembled correctly with respect to topology.*

Proof. It follows from Definition 5.4.1 that a maximal polygon is a polygon that is either

- contained wholly in a page,
- formed of crossover polygons that have already been merged to form a complete polygon, or

- a crossover polygon that is still waiting for its peer from another page. In this case, the boundaries of the page can be viewed as the temporary boundaries of the maximal polygon.

The objective is to prove that at any time during the algorithm, the set of maximal polygons is maintained.

At the leaf level of the MFP tree, every page is made up of maximal polygons. These maximal polygons are either crossover or in-page. The "outside" is viewed as just a normal maximal polygon with face id 0.

During the merging phase, at any intermediate node of the MFP tree, a set of maximal polygons is maintained. These polygons are either complete polygons that are formed of crossover polygons that have been merged, or they are crossover polygons that cannot be completed because they cross the pages spanned by the current node of the MFP tree.

At the root node, a set of maximal polygons are maintained. These maximal polygons are all complete polygons made up of merged crossover polygons.

□

5.4.3 Implementation of the message passing algorithm

The following sections describe several possible implementation scenarios of the message passing algorithm. The goal of each of these scenarios is to utilize the number of available processors to a maximum by limiting CPU idleness. At the same time, it is necessary to keep the number of messages exchanged in the order of $O(hn)$.

The simplest implementation is called *level asynchronous message passing*. It means that an intermediate node has to receive and assemble all crossover polygons from its child nodes before sending up its remaining crossover polygons to its parent. However, each processor has to first send all crossover polygons from the leaf level, and be ready to accept crossover polygons from other processors to any node it owns simultaneously to avoid deadlocks.

The advantage of this mechanism is its relative simplicity. The only data structure needed is a queue to store the crossover arcs received by the processor. Certain

Figure 5.19: Examples of level asynchronous message passing mechanism

temporary data structures are needed to store crossover arcs at each node that have not yet received its peer(s).

The drawback of this algorithm is that if one processor is too slow transmitting its crossover polygons to its peers, the other processors could find themselves idle, yielding a low parallelization speed up. But if the slowdown happens at the lower levels of the MFP tree, the other processors are able to go ahead and process other nodes hoping that the slow processor would eventually catch up. This slowdown is therefore at its worse when it happens higher up in the MFP tree when the other processors might have no other pages to process and there are idle waiting for the slow processor.

In *border-first asynchronous message passing*, the crossover polygons are divided into two categories depending on whether they cross the outside perimeter of the node's area. (See Figure 5.20.) The perimeter of a node represents the outside

Figure 5.20: Examples of crossover polygons that cross and do not cross node's boundary

border of the pages that are descending from this node. Crossover polygons that are crossing the perimeter cannot be resolved at the current node and have to be sent up to the parent node for completion. These crossover polygons are sent up to the parent node immediately. The crossover polygons that do not cross the perimeter are stored at the node in the hope that they could be assembled locally. Note that a crossover polygon that does not cross the perimeter of a node might have a peer that does cross the perimeter, and thus has to be sent up to its parent eventually. This mechanism increases the asynchronousness over the level asynchronous algorithm by allowing nodes to send certain crossover polygons up while still processing other polygons locally.

The main data structures remain the same as in the level asynchronous scheme. What changes is that every processor has to simultaneously check for every intermediate node whether it has received crossover polygons and whether it is able to send

crossover polygons one level up.

Figure 5.21: Examples of crossover polygons in Border-first message passing mechanism. Crossover polygons A, B, and C are complete with respect to the current node and can be sent up to the parent node. Polygon D has to receive and assemble with its peer from page 2 before being sent up.

A third implementation option is called *fully asynchronous message passing*. It allows intermediate nodes to send up all their crossover polygons once they have received a given proportion (threshold) of crossover polygons from their children. The threshold can be expressed as a proportion of child pages that have already sent their crossover polygons up, or in terms of crossover polygons received out of the total. A threshold of 1 is equivalent to the level asynchronous message passing, while a threshold of 0 means no computation is done at the intermediate nodes' level, and all the computation is done by the single processor that owns the root node.

With an optimal threshold value, this message passing algorithm eliminates processor idleness due to nodes waiting after other nodes for crossover polygons. Note that certain processor idleness, such as a processor finishing ahead of another processor, is unavoidable in static load balancing. On the other hand, fully asynchronous message passing increases the messages to be sent, since a crossover polygon that could be treated at a lower level in the MFP tree may be sent up to be assembled.

From the implementation perspective, this approach is more complicated than the

previous ones. Queues of crossover polygons have to be maintained at every intermediate node, to hold crossover polygons that haven't been processed yet. Moreover, every intermediate node has a status indicating whether it is collecting crossover polygons, sending up crossover polygons automatically upward, or it is done receiving crossover polygons.

Level asynchronous message passing is straightforward to implement and should provide good scalability to the number of processors because it generates less messages than the alternatives. It was the message passing algorithm that we implemented for this research.

Border first message passing is also fairly straightforward to implement. It increases code complexity somewhat, but it does not impact the number of messages being exchanged between the processors. It could be implemented in a next revision of the program, and tested against the performance of the level asynchronous message passing.

Fully asynchronous message passing is promising by potentially eliminating processor idleness due to waiting after other processors. However, it is a fair bit of work to implement and test fully. Also, the success of this mechanism depends heavily on the two metrics defined above: the threshold at which a node can start sending incomplete polygons up, and the maximum number of levels that a crossover polygon can travel on the MFP tree while its originating node is incomplete. The general applicability of this mechanism depends on how much these two metrics vary with the input map.

5.4.4 Neighbor exchange message passing

One way to reduce the number of messages passed between the processors is to exchange crossover polygons between neighboring pages. This is done by selecting crossover polygons that are candidates to span only two pages, and exchange them between the pages. For sake of consistency, crossover polygons are sent left to right, bottom to top between the pages. The receiving page decides whether it is able to form a complete polygon from the crossover polygon it has received. If it is, it sends

the updated arc id and face id information back to its sender, else, it sends the still incomplete crossover polygon to its parent.

In the best case, a crossover polygon that was a candidate in the source page is matched with a crossover polygon on the receiving page to form a complete polygon, therefore saving messages by not having to send the crossover polygon to higher nodes in the MFP tree. If the candidate crossover polygon fails, this will cost one unnecessary message.

5.4.5 Numerical issues related to polygonization

Polygonization can be seen as a graph algorithm. Thus, one would assume that it is not affected by numerical issues (see Chapter 4 for a description of numerical problems.) However, this is not entirely true. A polygonization algorithm does require one geometric component: sorting the arcs around a node in clockwise ordering. This computation is numeric, because it consists of computing the slope of the arcs surrounding the node.

Our crossover polygonization is also affected by numerical problems. Crossover polygons are reassembled using their crossover arcs' intersection point with the page boundary. The intersection point of an arc with the page boundary will be referred to as *clipping point*. Computing the clipping points is a geometric operation that carries a margin of error, hence the assembly is numerically unstable.

The solution we employed was to compute the clipping points of every segment in the input data *before* computing the intersection points. This ensures that upon assembly, the clipping point of a crossover arc computed from both pages is the same. Similarly, the crossover arcs of a node can be ordered in a clockwise fashion using these clipping points.

Chapter 6

Discussion of results

This chapter presents experimental results of the whole parallel map overlay algorithm.

6.1 Testing methodology

6.1.1 Setup

The following assumptions were used for the program:

- Input consists of two maps containing TIN data.
- Both maps are of the same scale. The lower left coordinate is (0, 0).
- The input consists of fractional coordinates, stored as 32-bit floating point numbers (float).
- The outer rim is considered the "outside" (TIN id 0.)
- Each map is read in by a single processor, which then recursively distributes the TINs among the processors (see MFP-trees in section 3.4.1.) The time to distribute the arcs between the processors does not count in the total running time. This is an unusual assumption, since task allocation between processors is an overhead in the realm of parallel processing. There are two reasons why we

chose not to count it. First, this part of the program was quickly coded and not optimized, because we concentrated on other areas of the algorithm. Second, since each input map is parsed separately into two distinct MFP-trees, one can assume a scenario where each map is pre-parsed into an MFP-tree with the optimal threshold, and stored locally at each node. This would allow processors to load in the map on demand, without data exchange between processors.

- The output of the polygonization is a polygon set.
- The resulting polygon set is kept in memory.
- The setup assumes that all data fits in the main memory of a single node. The reason behind this assumption is to avoid swapping. Obviously, good "parallel" speedups can be obtained by simply increasing the memory available to the program. We wanted to avoid this.

6.1.2 Hardware profile

The platform used for testing is a 64 node Linux cluster. Each node has 4 Intel(R) XEON(TM) processors at 2.00GHz with 512 KB of cache. Each node has 1.5 GB of main memory.

6.1.3 Input data classification

The input tests are divided into three categories:

- Uniform data distribution.
- Data distributed hierarchically but they are uniform across the two data sets. The two input maps are said to be *symmetric*.
- Data distributed hierarchically but the two input maps are asymmetric.

The distinction between uniform and hierarchical data distribution is trivial. Data partitioning using MFP-trees was designed to efficiently support hierarchical data.

The distinction between symmetric and asymmetric data is a crucial one. If the two input maps are symmetric, such as when a map is translated and overlaid on its peer, the MFP-trees of the two maps are similar. This allows overlaying pages without data exchange for the intersection phase. If the input is asymmetric, pages have to be exchanged for the overlay phase. This is a crucial symptom of the MFP-tree partitioning scheme.

Test	Input map 1	Input map 2	Num of vertices	Type ¹	data generated
Rand50k	r1Tin25k	r2Tin25k	50,000	U	random
Rand200k	r1Tin100k	r2Tin100k	200,000	U	random
Rand600k	r1Tin300k	r2Tin300k	600,000	U	random
Madagascar30k	Madagascar15k	self translated	30,000	H, S	real world
Madagascar100k	Madagascar50k	self translated	100,000	H, S	real world
Rand250k	r1Tin100k	r1Tin150k	250,000	H, A	random
Rand500k	r1Tin250k	r2Tin250k	500,000	H, A	random

Table 6.1: Input test cases

Figure 6.1: A 2 by 2 subdivision of a map.

Figure 6.2: A 3 by 3 subdivision of a map

¹U: uniform data, H: hierarchical data, S: symmetric maps, A: asymmetric maps

Table 6.1 summarizes the input tests. The label "self translated" indicates that every vertex in the first input map was translated by a constant value to produce the second map. Real world data means that the data was sampled from elevation models [13].

6.2 Results

The tests are were run on 1, 4, 9, 24 and 36 processors. The size of the data set determines the maximum number of processors it has been tested against. For small data sets, there is no point in testing it on 25 or 36 processors. The number of processors are chosen in such a way that it corresponds to the X and Y partition of the MFP-tree (see Section 3.4.1 for MFP-tree partition.) Therefore, a 2 by 2 partitioning needs 4 processors, 3 by 3 needs 9 processors, and so on. The dimensions of the overlay has to cover the dimensions of the maps and be a multiple of the number of partitions in both X and Y dimensions. For example, take the overlay of two maps both covering a rectangular area from lower-left corner (0, 0) to upper right corner (1000, 1000). We assume an MFP-partition of 2 by 2 (4 processors.) Then the area of coverage by the MFP-tree is from (0, 0) to (1024, 1024). ($2^{10} = 1024$). Similarly, for a 3 by 3 partitioning of the same overlay, the MFP-tree's area of coverage would be from (0, 0) to (2187, 2187). ($3^7 = 2187$). See figures 6.1 and 6.2 for visual representation. This over-sizing of the map is necessary so that the partition would fall on an integer, which can be stored exactly in floating-point representation, thus avoiding numerical problems.

Note that it is not always feasible to find the optimal threshold. Its value varies with the input maps and the number of processors used. We have experimented with several sample thresholds, and used the best for the computation.

Each test was run five times, and the average values were used in this result.

Figure 6.3: Running time of r50k

Figure 6.4: Running time of r200k

Figure 6.5: Running time of r600k

6.2.1 Uniform data

This section summarizes the results for uniform data input maps, namely tests r50k, r100k and r600k. The input maps were generated randomly, with a uniform distribution of the vertices over the maps coverage area. The vertices were then triangulated using LEDA's implementation of Delaunay Triangulation [37]. Because of the uniformness of the data, the MFP-tree pages are of the same size, and the MFP-tree structures corresponding to the input maps are identical. That means no inter-processor communication is necessary for the overlay (line intersection computation) itself, although there is message passing during crossover polygonization.

Tables 6.2, 6.3 and 6.4 show the running times of the overlay of the randomly generated maps r50k, r200k and r600k. All running times are in seconds. The parallel efficiencies range from 40% to 96%. Figure 6.7 summarizes the parallel efficiencies

Number of procs	Time for intersection (s)	Time for polygonization (s)	Time (total) (s)	Speedup	Parallel efficiency (%)
1	11	55	66	1	100
4	4	17	21	3.1	79
9	3	12	15	4.4	49
16	1	5	6	11	69

Table 6.2: Running time of r50k assuming best threshold

Number of procs	Time for intersection (s)	Time for polygonization (s)	Time (total) (s)	Speedup	Parallel efficiency (%)
1	63	434	497	1	100
4	21	109	130	3.8	96
9	13	64	77	6.4	72
16	6	41	47	10.6	66
24	5	23	28	17.8	74

Table 6.3: Running time of r200k assuming best threshold

of these tests per number of processors. Tests done with r200k yield the best parallel efficiencies (from 66% to 96%), while the biggest sample, r600k, has the worst efficiencies, from 40% to 52%. Figure 6.6 shows that the input maps are scalable to the number of processors tested. The difficulty of approximating the best threshold accounts for the large variance in parallel efficiencies for tests r50k.

Figures 6.3, 6.4 and 6.5 show how long different parts of the algorithm takes. In-page polygonization dominates the algorithm at lesser number of processors, while crossover polygonization gains in importance as the number of processors increases. As the number of processors increases, the time for intersection decreases to the point where there is no more noticeable performance improvement.

6.2.2 Hierarchical data and symmetric maps

This section presents results about tests with input maps that have a hierarchical distribution of data. It means that there is a significant difference in data density throughout the map. By consequence, the pages of the MFP-tree will be of varying

Number of procs	Time for intersection (s)	Time for polygonization (s)	Time (total) (s)	Speedup	Parallel efficiency (%)
1	194	551	745	1	100
4	52	365	417	1.9	48
9	26	137	163	4.6	51
16	14	76	90	8.3	52
24	11	62	73	10.2	43
36	9	43	52	14.3	40

Table 6.4: Running time of r600k assuming best threshold

Figure 6.6: Speedup for uniform data distribution

Figure 6.7: Parallel efficiency for uniform data distribution

size. (See Figure 6.8.) However, the two input maps are symmetric, which means that a high density area on one map has the same location as the corresponding high density area on the other map. This creates identical MFP-tree structures. For two identical MFP-trees, corresponding pages are of the same size and they are mapped to the same processor, but they contain different data. This means that just like for uniform data distribution, hierarchical but symmetric data distribution does not require processor interaction for the overlay phase.

Tables 6.5 and 6.6 show the running time of Madagascar30k and Madagascar100k maps. Both of these maps are scalable to 16 processors, with the larger Madagascar100k yielding better performance. Its parallel efficiency ranges from 88% to 98%, while Madagascar30k's efficiency ranges from 45% to 63%. In all cases, lesser number

Figure 6.8: Madagascar30k: example of a hierarchical and symmetric overlay.

Number of procs	Time for intersection (s)	Time for polygonization (s)	Time (total) (s)	Speedup	Parallel efficiency (%)
1	9	41	50	1	100
4	4	16	20	2.5	63
9	2	7	9	5.6	62
16	1	6	7	7.1	45

Table 6.5: Running time of Madagascar30k assuming best threshold

of processors yield higher parallel efficiencies (see Figure 6.12.) Just like for uniform data distribution, in-page polygonization dominates the running time until 8 processors, then crossover polygonization becomes more important (See Figures 6.9, 6.10.)

6.2.3 Hierarchical data and asymmetric maps

This section presents the results of overlaying maps with hierarchical and asymmetric data distribution. This means that not only portions of each map contains more data than others, but also that these higher density portions are at different position from one map to the other. (See Figure 6.13.) As a consequence, the overlay of pages itself for intersection computation could involve the exchange of pages between processors.

Tables 6.7 and 6.8 show the running time of tests r250k and r500k. The larger

Figure 6.9: Running time of Madagascar30k

Figure 6.10: Running time of Madagascar100k

Number of procs	Time for intersection (s)	Time for polygonization (s)	Time (total) (s)	Speedup	Parallel efficiency (%)
1	31	181	212	1	100
4	10	44	54	3.9	98
9	5	20	25	8.5	94
16	3	12	15	14.1	88

Table 6.6: Running time of Madagascar100k assuming best threshold

r500k has a better efficiency, from 48% to 89%. r250k's parallel efficiency has a shorter range of 52% to 65%. Page exchange during intersection had a large variance from processor to processor, which explains the different parallel efficiencies at different number of processors. (See Figures 6.14 and 6.15.)

6.3 Summary of results

The results show parallel efficiencies varying from 40% to 98% up to 36 processors. The results demonstrate not only that MFP-trees are scalable to a fairly large number of processors, but that it is able to support hierarchical data without sacrificing efficiency at uniform data.

The results show that crossover polygonization increases in importance in the

Figure 6.11: Speedup for hierarchical data distribution with symmetric maps

Figure 6.12: Parallel efficiency for hierarchical data distribution with symmetric maps

Number of procs	Time for intersection (s)	Time for polygonization (s)	Time (total) (s)	Speedup	Parallel efficiency (%)
1	74	414	488	1	100
4	34	155	189	2.6	65
9	55	50	105	4.6	52
16	10	45	55	8.9	55
24	17	19	36	13.6	56

Table 6.7: Running time of r250k assuming best threshold

algorithm with the increase in the number of processors. This is an obstacle to the scalability of the algorithm over a large number of processors. Moreover, for hierarchical data sets, the height of the MFP-tree increases. On one hand, we want a low threshold to achieve a better load balancing for hierarchical data sets, on the other hand, we want to limit the number of crossover polygons to be assembled. These are contradictory goals, since a lower threshold results in more pages and this in turn increases the number of crossover polygons. By employing a neighbor exchange message passing mechanism, as described in Section 5.4.4, in addition to the level asynchronous message passing would decrease the time required for crossover polygonization. Hence only the level asynchronous message passing was implemented

Figure 6.13: Example of a hierarchical and asymmetric overlay: overlaying R250k.dat (right) with R100k.dat (left)

Number of procs	Time for intersection (s)	Time for polygonization (s)	Time (total) (s)	Speedup	Parallel efficiency (%)
1	151	1447	1598	1	100
4	203	279	482	3.3	83
9	78	165	243	6.6	73
16	35	77	112	14.3	89
24	49	48	97	16.4	69
36	54	38	92	17.4	48

Table 6.8: Running time of r500k assuming best threshold

for this phase of the algorithm.

Figure 6.14: Running time of r250k

Figure 6.15: Running time of r500k

Figure 6.16: Speedup for hierarchical data distribution with asymmetric maps

Figure 6.17: Parallel efficiency for hierarchical data distribution with asymmetric maps

Chapter 7

Conclusion and further research

The goal of this research was to parallelize the overlay algorithm for maps containing vector data. Map overlay is an important operation in GIS and one of the most processor intensive. In the course of our research, we have encountered two main obstacles:

1. how to partition data between the processors, and
2. how to solve numerical issues in an efficient way.

Partitioning the data between processors is non-trivial, because input maps can have variable densities. Using a uniform sized partitioning technique on maps of variable densities would yield unsatisfactory results due to poor load balancing. To overcome this obstacle, we have used an MFP-tree to partition the input maps. An MFP-tree was designed for maps with variable data densities. Moreover, an MFP-tree associates pages to processors statically, which eliminates the overhead of dynamic load balancing.

Spatial partitioning of data presented us with another challenge. Arcs that are crossing page boundaries that are owned by different processors have to be polygonized. The difficulty lies in that no single processor has a complete view of a crossover polygon. To resolve this problem, we took advantage of the tree structure of the MFP-tree. Arcs that are crossing page boundaries are sent up to their parent node in the

MFP-tree for polygonization. This procedure is repeated all the way to the root of the MFP-tree. We have analytically evaluated three message passing mechanisms between pages permitting polygonization of an MFP-tree and have implemented one of them.

The second challenge we faced is generic to most computational geometry problems. Numerical issues affect the line intersection section of our algorithm. Numerical issues are related to floating-point computation, which introduces numerical instability to the algorithm. Although several techniques exist to solve numerical problems, such as rational arithmetic and general-purpose floating-point filters, they were not sufficiently fast for our purpose. We have decided to implement a line intersection specific floating-point filter, which outperformed a general-purpose filter implemented in the LEDA library from four to fifteen times.

We have tested our algorithm by overlaying maps of uniform and variable data densities. We have achieved parallel efficiencies between 40 to 96% scaling up to 36 processors.

7.1 Future work

There are possible improvements to the map overlay algorithm. Our implementation stops at polygonization. Allocation of attribute data to the resulting map and boolean operations on the map were not implemented for this first stage of our program.

The running time of the map overlay algorithm can be reduced still by implementing neighbor exchange message passing to speed up crossover polygonization. (See Section 5.4.4.)

At the present time, the threshold to partition the input domain is set statically. It would be interesting if this value could be found dynamically. It could be achieved by recursively partitioning the input between the processors until all processors own the same number of nodes within a bound. This procedure, however, requires a lot of fine-tuning. The performance would degrade if the height of the MFP-tree is getting too long. Therefore, a minimum threshold has to be established under which the page should not be partitioned any longer.

References

- [1] M. deBerg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2., rev. ed., Springer, 2000.
- [2] K. Hwang, Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*, McGraw-Hill, 1998.
- [3] I. Foster, "Designing and Building Parallel Programs", Addison-Wesley, 1995.
<http://www-unix.mcs.anl.gov/dbpp/>
- [4] J. Lou and J. Farrara, "Performance Analysis and Optimization on a Parallel Atmospheric General Circulation Model Code", Proceedings of the 11th International Parallel Processing Symposium, IEEE Computer Society Press, 1997, pp. 174-180.
- [5] Tz. Ostromsky and Z. Zlatev, "Parallel Implementation of a Large-Scale 3-D Air Pollution Model", in S. Margenov, J. Wasniewski and P. Yalamov, eds., Large Scale Scientific Computing, Third International Conference, 2001, Springer-Verlag, 2001, pp. 309-316.
- [6] I.B. Bazhlekoy, F.N. van de Vosse and H.E.H. Meijer, "Boundary Integral Method for 3D Simulation of Foam Dynamics", in S. Margenov, J. Wasniewski and P. Yalamov, eds., Large Scale Scientific Computing, Third International Conference, 2001, Springer-Verlag, 2001, pp. 401-408.

- [7] R.C.F. Melo, M.E.T. Walter, A.C.M.A. Melo, R. Batista, M. Nardelli, T. Martins and T. Fonseca, "Comparing Two Long Biological Sequences Using a DSM System", in H. Kosch, L. Boszormenyi and H. Hellwagner, eds., Euro-Par 2003 Parallel Processing, 9th International Euro-Par Conference, Springer-Verlag, 2003, pp. 517-524.
- [8] K. Popov, V. Vlassov, M. Rafea, F. Holmgren, P. Brand and S. Haridi, "Parallel Agent-Based Simulation on a Cluster of Workstations", in H. Kosch, L. Boszormenyi and H. Hellwagner, eds., Euro-Par 2003 Parallel Processing, 9th International Euro-Par Conference, Springer-Verlag, 2003, pp. 470-480.
- [9] J. Dangermond, "What is a Geographic Information System (GIS)?" Geographic Information Systems (GIS) and Mapping - Practices and Standards, ASTM STP 1126, A. I. Johnson, C.B. Pettersson, J. L. Fulton, eds., American Society of Testing and Materials, Philadelphia, 1992, pp. 11-17.
- [10] J. Star, J. Estes, Geographic Information Systems: an introduction, Prentice Hall, New Jersey. 1990.
- [11] T. Bernhardsen Geographic Information Systems: an introduction, second edition, John Wiley & Sons. 1999.
- [12] M. Lanthier, D. Nussbaum and J.-R. Sack, "A Parallel Implementation of Geometric Shortest Path Algorithms", (Special issue on High Performance Computing with Geographical Data), Parallel Computing, Vol. 29, pp. 1445-1479, 2003.
- [13] M. Lanthier, Shortest Path Problems on Polyhedral Surfaces, Ph. D. thesis, 1999.
- [14] M.N. Demers, Fundamentals of Geographic Information Systems, 2nd edition, John Wiley & Sons, 2000, pp. 498.
- [15] I. Heywood, S. Cornelius and S. Carver, An Introduction to Geographical Information Systems, 2nd edition, Prentice Hall, 2002.

- [16] J. Dongarra, S. W. Otto, M. Snir, D. Walker, "An Introduction to the MPI Standard", 1995. <http://www.netlib.org/utk/papers/intro-mpi/intro-mpi.html>.
- [17] D. White, "A Design for Polygon Overlay", First International Advanced Study Symposium on Topological Data Structures for GIS, Harvard Papers on GIS, vol. 6, 1977.
- [18] W. R. Franklin, C. Narayanaswami, M. Kanhkalli, D. Sun, M. Zhou and P. Y. F. Wu, "Uniform Grids: A Technique for Intersection Detection on Serial and Parallel Machines", Auto Carto 9: Ninth International Symposium on Computer Assisted Cartography, 1989, pp. 100-109.
- [19] W. R. Franklin and C. Narayanaswami, "Boolean Combinations of Polygons in Parallel", 1992 International Conference on Parallel Processing, vol 3, 1992, pp. 132-135.
- [20] S. Hopkins, R. G. Healey, T. C. Waugh, "Algorithm Scalability for Line Intersection Detection in Parallel Polygon Overlay", Proceedings of the 5 th International Symposium on Spatial Data Handling, vol. 1, 1992, pp. 210-217.
- [21] T. C. Waugh and S. Hopkins, "An algorithm for polygon overlay using cooperative parallel processing", International Journal of Geographical Information Systems, 1992, vol. 6, no. 6, 1992, pp. 457-467.
- [22] T. J. Harding, R. G. Healey, S. Hopkins, S. Dowers, "Vector Polygon Overlay" in R. Healey, S. Dowers, B. Gittings, M. Mineter, Eds., Parallel Processing Algorithms for GIS, Taylor & Francis Ltd., 1998. pp. 265-310.
- [23] M. J. Mineter, "Creation of Vector-Topological Data Structures" in R. Healey, S. Dowers, B. Gittings, M. Mineter, Eds., Parallel Processing Algorithms for GIS, Taylor & Francis Ltd. 1998. pp. 179-214
- [24] J. Nievergelt, H. Hinterberger and K. D. Sevcik, "The grid file: An adaptable, symmetric multikey file structure." ACM Transactions on Database System, vol. 9, no. 1, 1984, pp. 38-71.

- [25] Yu-Tai Ching, "Load Balancing for the Parallel Map Overlay-Operation in the Geographic Information System", *Journal of Information Science and Engineering*, vol 15, 1999, pp. 441-449
- [26] F. Wang, "A Parallel Intersection Algorithm for Vector Polygon Overlay", *IEEE Computer Graphics and Applications*, vol. 13, no. 2, 1993, pp. 74-81.
- [27] S. Shekhar, S. Ravada, G. Turner, D. Chubb, and V. Kumar, "Load Balancing in High Performance GIS: Partitioning Polygonal Maps", *Proceedings of the International Symposium on Large Spatial Databases*, Springer Verlag (Lecture Notes in Computer Science), 1995.
- [28] A. Frankel, D. Nussbaum and J.-R. Sack, "Floating-point filter for the line intersection algorithm", to appear in *Proceedings of the Third International Conference on Geographic Information Science*, GIScience, Springer Verlag Lecture Notes in Computer Science, 2004.
- [29] J. L. Bentley, T. A. Ottmann "Algorithms for reporting and counting geometric intersections" *IEEE Trans. Comput.*, vol. 28, 1979, pp. 643-647.
- [30] I.J. Balaban, "An optimal algorithm for finding segment intersections" in *Proceedings of the 11th Annual Symposium on Computational Geometry*, 1995, pp. 211-219.
- [31] D. Nussbaum, *Parallel spatial modelling*. Ph. D. thesis, Ottawa, 2001.
- [32] C. Hecker, *A Vector Polygon Algorithm Built with Transparent Parallel Middleware*, M. C. S. thesis, Ottawa, 2000.
- [33] H. Samet, "The Quadtree and Related Hierarchical Data Structures", *ACM Computing Surveys*, Volume 16 , Issue 2, 1984, pp. 187-260.
- [34] U. Bartuschka, K. Mehlhorn and S. Naher, "A Robust and Efficient Implementation of a sweep line algorithm for the straight line intersection problem." *Proc. Workshop on Algorithm Engineering*, 1997, pp. 124-135.

- [35] E. Martinian, "C++ code for Red-Black Trees And Interval Trees", http://www.csua.berkeley.edu/~emin/source_code/cpp_trees/index.html, 2002.
- [36] K. Mehlhorn and S. Naher, "The Implementation of Geometric Algorithms." 13th World Computer Congress IFIP 94, Elsevier Science B.V., Vol. 1, 1994, pp. 223-231.
- [37] K. Mehlhorn and S. Näher, "The LEDA Platform of Combinatorial and Geometric Computing", Cambridge University Press, 1999.
- [38] D. Goldberg "What Every Computer Scientist should know about Floating-Point Arithmetic", ACM Computing Surveys, v. 23 no 1, 1991, pp. 5-48.
- [39] L. N. Trefethen, D. Bau, "Numerical Linear Algebra", SIAM, 1997, pp. 97-100.
- [40] E.W. Weisstein, "Angle Bisector." From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/FibonacciNumber.html>
- [41] S. Schirra, "Robustness and Precision Issues in Geometric Computation", in J.-R. Sack, J. Urrutia, Eds., Handbook of Computational Geometry, 1999, pp. 597-632.
- [42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, "Introduction to Algorithms", MIT Press, 1997, pp. 263-280.
- [43] J.R. Shewchuk, "Adaptive Precision Floating-point Arithmetic and Fast Robust Geometric Predicates", Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, 1996.
- [44] G. Knott and E. Jou, "A Program to Determine Whether Two Line Segments Intersect", Technical Report CAR-TR-306, CS-TR- 1884, DCR-86-05557, Computer Science Dept., University of Maryland at College Park, 1987.
- [45] M. Karasick, D. Lieber, and L. R. Nackman. "Efficient Delaunay Triangulations Using Rational Arithmetic", ACM Trans. Graph., vol. 10, no. 1, 1991, pp. 71-91.

- [46] H. Brönnimann, G. Melquiond, and S. Pion, "The Boost Interval Arithmetic Library", In Proc. 5th Conference on Real Numbers and Computers, 2003, pp. 65-80.
- [47] H. Brönnimann, C. Burnikel, and S. Pion, "Interval arithmetic yields efficient dynamic filters for computational geometry", Discrete Applied Mathematics, vol. 109, 2001, pp. 25-47.
- [48] The Core Library, see <http://www.cs.nyu.edu/exact/core/>.
- [49] B. Hayes, "A Lucid Interval", American Scientist, Vol. 91, No. 6, 2003, pp. 484-488.
- [50] R.B. Kearfott, "Interval Computations: Introduction, Uses and Resources", see <http://www.cs.utep.edu/interval-comp/>, 2004.
- [51] C. Yap "On Guaranteed Accuracy Computation", chapter in: "Geometric Computation" (eds. Falai Chen and Dongming Wang) World Scientific Publishing Co., 2004 (to appear).
- [52] J.-D. Boissonat, A. Vigneron, "Elementary algorithms for reporting intersections of curve segments." Comput. Geom. Theory Appl., vol. 21, 2002, pp. 167-175.
- [53] G. Liotta, F.P. Preparata, R. Tamassia, "Robust proximity queries: an illustration of degree-driven algorithm design." Proceedings of the 13th Annual ACM Symposium of Computational Geometry, 1997, pp 156-165.
- [54] E. Rafalin, D. Souvaine, I. Streinu, "Topological Sweep in Degenerate cases", Proceedings of the 4th international workshop on Algorithm Engineering and Experiments, Springer-Verlag, Berlin, Germany, 2002, pp. 155-165.