INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

A Vector Polygon Overlay Algorithm Built with Transparent Parallel Middleware

by

Chris Hecker

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment
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
May 2000

(c) Copyright 2000 by Chris Hecker

All Rights Reserved.

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

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

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 a accordé une licence non exclusive permettant à la Bibliothèque nationale du 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.

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.

0-612-57755-4

Canada

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

**“A Vector Polygon Overlay Algorithm Built with
Transparent Parallel Middleware”**

submitted by CHRIS HECKER, B.Sc. (Honours),
in partial fulfilment of the requirements for
the degree of Master of Computer Science.

F. Fiala

for

Thesis Supervisor

F. Fiala

for

Director, School of Computer Science

Carleton University

24 May 2000

Abstract

Vector-geographical computing can require expertise in Computational Geometry, Numerical Methods, and Algorithm Design. Large vector-based Geographical Information Systems usually require expertise in Complex System Design and/or Parallel Computing. Such levels of expertise are not present in the majority of people who are required to perform the implementation of such systems. Therefore, new aids to system design are required.

I describe the design and implementation of a software framework (*middleware*) that performs such a function in vector-geographical computation. The ATLANTIS Overlay Driver (“OVID”) is constructed with the purpose of providing accurate, efficient, and transparent parallel solutions to users of geographic overlay algorithms. The design and design principles of OVID are described herein. As proof of concept, the major phase (line segment intersection) of Vector Polygon Overlay has been implemented. The performance of the implementation is good for clustered spatial data, with as high as 75% efficiency.

Acknowledgments

Many thanks to my supervisor, Dr. J.-R. Sack, for his guidance and collaboration, his hard work on the paper we co-authored, his thoughtful examination of my thesis, and his skill at steering me through the thesis defence process. Thank you to Dr. Frank Dehne for his teaching and mentoring during my years at Carleton. Thank you to Dr. Rolf Wanka for his interest and sponsorship during my time at Universität Gesamthochschule Paderborn. To all my colleagues and mentors at the PARADIGM Group at Carleton University in Ottawa, thanks for their help and community spirit.

I appreciate NSERC, which sponsored my work with a PGS 'A'. This was influential in my starting a graduate degree. Thanks also to Almerco, which provided some funding for this research.

The software developed here was built on top of software developed by Doron Nussbaum. Thanks to Doron for his help and support of my work. Thanks also to Mark Lanthier for his help and collaboration in the lab, and to Jason Morrison, Stefan Christiansen, and Mike Valenta for their help in debugging algorithms and code. I appreciate the hard work and cheerful help given me by the secretarial staff at the School of Computer Science, particularly Linda Pfeiffer's efforts to shoehorn my thesis defence into the one place on the calender it could fit.

The contribution of Young-Soo Roh, who implemented some sequential polygonization and MFP data-structure-building code, and helped debug my algorithms, is appreciated.

Thanks to my parents and extended family for their help and support during the crazy days of my thesis. I wouldn't have survived without you!

This document was written in $\text{L}\text{\kern-0.1em}\text{\kern-0.1em}\text{X}$, the $\text{L}\text{\kern-0.1em}\text{\kern-0.1em}\text{A}\text{\kern-0.1em}\text{T}\text{\kern-0.1em}\text{E}\text{\kern-0.1em}\text{X}$ -based document processor. Thanks to the $\text{L}\text{\kern-0.1em}\text{\kern-0.1em}\text{X}$ team for taking $\text{L}\text{\kern-0.1em}\text{\kern-0.1em}\text{A}\text{\kern-0.1em}\text{T}\text{\kern-0.1em}\text{E}\text{\kern-0.1em}\text{X}$ into a new millennium.

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	Dealing with Complex Solutions	2
1.2	Intent of the Research	3
1.3	Thesis Chapter Structure	5
2	Background	6
2.1	Parallel Computing	6
2.1.1	Challenges of Parallel Computing	7
2.1.2	Definitions and Models of Parallel Computation	8
2.2	Computational Geometry	12
2.2.1	The Line Intersection Problem	13
2.2.2	The Planar Map Polygonization Problem	14
2.2.3	The Planar Triangulation Problem	14
2.3	Geographic Information Systems	15
2.3.1	Background & History	15
2.3.2	GIS Basics	16

2.4	Vector Polygon Overlay	17
2.4.1	The Generic Overlay Operation	18
2.4.2	Vector Polygon Overlay	19
2.5	The ATLANTIS GIS	21
2.5.1	The Composition of Atlantis	21
2.5.2	The NEMO Cell Driver and its Influence on OVID	23
3	Theoretical Approach.	26
3.1	The Multidimensional File Partition	26
3.1.1	Design Intent	26
3.1.2	Building an MFP	29
3.1.3	Theoretical Considerations	30
3.1.4	Algorithm Construction with MFP	34
3.2	Parallel Vector Polygon Overlay	35
3.2.1	Sequential Algorithms	36
3.2.2	Design of a Parallel VPO Algorithm	39
3.2.3	Parallel Line Intersection	43
3.2.4	Parallel Polygonization and Triangulation	46
4	Practical Approach	58
4.1	Software architecture.	58
4.1.1	A Three-Layer Architecture	59
4.1.2	Functional Division of Algorithms	62
4.2	Building an OVID Application	62

4.2.1	OVID Generic Overlay Driver Framework	63
4.2.2	Line×Line Overlay Driver	63
4.2.3	Supporting Operations	64
4.2.4	Building OVID Applications	64
4.3	OVID Class Structure	66
4.3.1	MFP Software	66
4.3.2	OVID Foundation Classes	66
4.3.3	OVID Base Classes	68
4.3.4	OVID Algorithm-Specific Classes	70
4.4	Numerical and Topological Issues	70
4.4.1	Background and General Approaches	71
4.4.2	Issues in Line Segment Intersection	73
4.4.3	Issues in Polygonization	77
4.5	Implementation Issues	78
4.5.1	Attribute Data: Loss and Recovery	79
4.5.2	Determining Neighbouring Processors	81
4.5.3	Issues in Line Segment Propagation	84
4.5.4	Issues in Triangle/Polygon Propagation	88
4.6	Implementation Aids	90
4.6.1	MPI	91
4.6.2	LEDA	91
4.7	Difficult Problems	93
4.7.1	Programming Line Segment Algorithms	94

4.7.2	Memory Management Issues	96
4.7.3	Meshing with Other Software	96
4.7.4	Known Problems in the Current Implementation	97
5	Experimental Approach.	98
5.1	Comparative Results	98
5.2	Quantitative Measures of Interest	99
5.3	Test Platform	100
5.4	Experimental Data	101
5.5	Hypotheses	102
5.5.1	Parallelism: Factoring the Number of Processors	102
5.5.2	Data Factors: Clustering and Size	103
5.5.3	Partitioning: Changing Data Granularity	104
5.6	Test Procedure	105
5.7	Results	107
5.7.1	Results Summary	107
5.7.2	Load-Balance in Datasets	109
5.7.3	Speed-ups and Run-time.	113
6	Conclusions and Future Direction	120
6.1	The Effectiveness of MFP-based Vector Polygon Overlay	120
6.1.1	The Effect of Parallelism	121
6.1.2	The Effect of Data Clustering	121
6.1.3	The Effect of Dataset Size	121

6.1.4	Comparisons to Other Implementations	122
6.2	Ease of Programming	122
6.3	Parallel Algorithms	123
6.4	Further Research	123

List of Figures

2.1	ATLANTIS Architecture Diagram	23
2.2	NEMO Communication Diagram	24
3.1	Page Level Designation in a 2x2 MFP.	31
3.2	Levels 0 to 3 of a 3x3 MFP	32
3.3	Overlay Algorithm: Finding Matching Pages	42
3.4	Four Stages of the Parallel Segment Resolution Algorithm.	47
3.5	Intermediate Processing Steps on a 64-Processor Mesh.	52
3.6	Open and Leaky Polygons on a Page Boundary.	53
3.7	An Example Polygon Visibility Hull	56
3.8	Inter-Page Messaging for a 36-Processor MFP Triangulation.	57
4.1	OVID's Three-Layer Architecture	61
4.2	Example OVID Vector Polygon Overlay Application	65
4.3	Class Diagram for OVID.	71
4.4	Five Common Examples of Floating Point Failures in Line Segment Intersection.	74
4.5	Ten cases of Line Segment Intersection	77
4.6	Three-Level Segment Attribute Recovery Structure	80

4.7	Unique IDs in a 2-Level 2x2 Processor MFP.	82
4.8	Neighbour-Finding in an MFP.	84
4.9	Special Messaging with Corner-Intersecting Segments	86
4.10	Segment Intersecting Page Corners, Collinear with Page Boundaries	86
5.1	Madagascar TIN	102
5.2	America TIN	103
5.3	Africa TIN	104
5.4	Australia TIN	105
5.5	Raster Partitioning: Percent Improvement in Standard Deviation, Australia TIN.	108
5.6	Standard Deviations of Overlay files' Per-Processor file sizes (High Threshold)	109
5.7	Standard Deviations of Overlay files' Per-Processor file sizes (Low Threshold)	110
5.8	Load Imbalance for Six Overlay tests, High Thresholds. Seconds vs. Number of Processors.	111
5.9	Load Imbalance for Six Overlay tests, Low Thresholds. Seconds vs. Number of Processors.	111
5.10	Performance Curve (total time). Seconds of Execution vs. Number of Processors. High-threshold Data Files.	114
5.11	Performance Curve (sequential overlay phase). Seconds of Execution vs. Number of Processors. High-threshold Data Files.	115
5.12	Performance Curve (post-overlay parallel segment resolution phase). Seconds of Execution vs. Number of Processors. High-threshold Data Files.	116
5.13	Performance Curve (post-overlay parallel page exchange phase). Seconds of Execution vs. Number of Processors. High-threshold Data Files.	116

5.14 Performance Curve (total time). Seconds of Execution vs. Number of Processors.	
Low-threshold data files.	117
5.15 Performance Curve (sequential overlay phase). Seconds of Execution vs. Number of Processors. Low-threshold Data Files.	118
5.16 Performance Curve (post-overlay parallel communication phase). Seconds of Execution vs. Number of Processors. Low-threshold Data Files.	118
5.17 Performance Curve (post-overlay page exchange phase). Seconds of Execution vs. Number of Processors. Low-threshold Data Files.	119

List of Algorithms

1	MFP Geometric Primitive Resolution	35
2	Page-Matching Algorithm for Dynamic Data Distribution	43
3	MFP Polygonization	50
4	MFP Polygonization with Triangulation	54
5	Determination of the Global MFP Page Map	83
6	Resolving the Doppelgänger Segment in a Received Message	89

Chapter 1

Introduction

Chapter 1 introduces this research, and the motivations behind it. The broad problems within GIS and Computer Science addressed by this research are discussed, as are the reasons for pursuing this particular line of inquiry.

1.1 Motivation

As we begin the twenty-first century, the computer is ubiquitous. The power and speed of computers continues to increase exponentially according to Moore's law (see [76]).

Yet, remarkable increases are occurring in the number of people using the internet, the amount and detail of data stored, and the complexity of computer networks. Such increases also affect the sophistication of the algorithms used to handle people, data, and networks. People find new uses for faster machines, or use them in more sophisticated ways to solve old problems more effectively or accurately. In some cases, software simply absorbs improvements in processor speed and storage capacity, by becoming less efficient from release to release. As a result, the demand for increased computing power is constant.

1.1.1 Dealing with Complex Solutions

Faster, more complex, and more sophisticated software necessarily requires more intelligent, knowledgeable and sophisticated programmers. Sack [77] writes that many industrial and scientific users do not use parallel computing because “the *threshold of accessibility* to this technology is known to be very high”.

Sack gives several reasons for this [77]:

- ★ Sequential algorithms are often difficult to parallelize.
- ★ The design of parallel algorithms may require “novel ideas combined with special knowledge” in order to fully exploit parallelism.
- ★ The implementation of parallel algorithms is “highly non-trivial”, often adding a further layer of complexity to an already complex piece of software.
- ★ “Environments supporting implementation of parallel algorithms are few.”

These points apply to many kinds of algorithms outside of parallel algorithms, as well.

Domain-Specific Algorithms

Highly specialized algorithms and software are required to solve problems in many domains of work and research. In many cases, the combination of innovation, algorithmic expertise, and specialist knowledge makes programming for technical specialties difficult, because of the high competency required of the programmer. Software development environments for these technical specialties is also lacking, not in the least because of the dearth of such multi-domain experts. Where such environments do exist, they can be prohibitively expensive.

End-users require the ability to solve their problems with custom computer-based tools and applications. Many of the problems in question will be specific to the subject area that the user works in, and

not addressed by generally available software. This is particularly true of GIS, which has high performance requirements and demands specialist treatment in spatial algorithms, *parallel* spatial algorithms, data structures, and numerical methods.

Such users will not be capable of learning the techniques necessary to create their own software; neither will they be able to afford an expert programmer to create software for them. In order for them to build the needed software themselves, such people will require the computer to communicate with them at their own conceptual level, and in that of their domain [38].

Anderson et al. [5] write that:

The real issues in computing in the next decade involve innovations that will allow relatively unsophisticated users to access the power of computer hardware, without having to become experts in programming an computer operating systems. The tools for GIS . . . should become easier to use . . . and at the same time [the users should] be able to take advantage of the new advances in hardware and software technology.

Such innovations are needed more now than when this was first written in 1991. Transparent handling of parallelism, complex data structures, sophisticated algorithms, topological error and irregularities, and numerical methods are needed to provide non-expert programmers and end-users access to advanced technologies and methods.

With the breaching of this threshold of accessibility, user communities will be provided with a much stronger technological base from which to base their research, and the development of their domain.

1.2 Intent of the Research

This research contributes to a continuing effort into software development for the GIS (Geographic Information Systems) domain. Specifically, it is part of the mandate of the PARADIGM Group to develop innovative methods of software development that enable users and programmers to build GIS tools and

applications without requiring expertise in parallel computing, numerical methods, sophisticated data structures, or geometric computing.

The goal of PARADIGM's GIS research is to develop useful, efficient parallel libraries of Computational Geometry and GIS software for use by non-specialist programmers and end-users. PARADIGM has a set of GIS drivers and libraries in continuous development, which are grouped under the name ATLANTIS. This thesis describes the design and implementation of a parallel engine for vector-based GIS overlay operations. The contributions of the research are:

- ★ The development of a customizable generic overlay driver, OVID. This is a proof-of-concept of several parts of the ATLANTIS vector-based GIS. It includes a demonstration that a self-contained overlay operation can be constructed on top of the generic overlay application.
- ★ A demonstration of the usefulness of the MFP (*Multidimensional File Partition*) data partitioning scheme for parallel computers, developed by Doron Nussbaum [63] [61]. This includes a demonstration by experimental result that the MFP-based overlay driver works well and gives a good speed-up.
- ★ The creation of MFP-based algorithms for parallel line segment overlay, parallel polygonization, and parallel triangulation. As well, several algorithms are given addressing implementation-specific issues within those general algorithms.
- ★ The creation of a relatively simple and easy-to-program API (*Application Programming Interface*) for OVID, that encapsulates parallelism and the multi-layered architecture.

OVID's success in these contributions will be measured in the following ways:

- ★ The usefulness of the MFP scheme and the overlay driver architecture will be measured via the parallel speed-up achieved by OVID's Vector Polygon Overlay (VPO) driver.
- ★ The effectiveness of the overlay driver's design principles will only become apparent as other types of overlay drivers are built on top of the base software. Nevertheless, the ease of implementation of the VPO will be an important indication of the success of this approach.

Intended Audience

This document has two purposes and two intended audiences. First, it is a description of my research intended for the academic community. Second, it is a documentation of my work for those in the PARADIGM Group that work with OVID after me.

1.3 Thesis Chapter Structure

The remainder of this thesis is broken into five chapters:

- ★ Chapter 2 provides a background to the work, and introduces important concepts, algorithms, and areas of research.
- ★ Chapter 3 deals with detailed theoretical aspects of the research, including the theory of parallel Vector Polygon Overlay (VPO) and the design of Nussbaum's MFP spatial file partitioning scheme.
- ★ Chapter 4 details practical, implementation-related concerns such as the design of OVID's software architecture, the machine architecture of the test platform, numerical issues, and parallel data propagation algorithms.
- ★ Chapter 5 details the experimental approach to validating this work, and reports the results and analysis of the experiments.
- ★ Chapter 6 gives conclusions and directions for future research.

Chapter 2

Background

This chapter gives an overview of the areas of research involved in this thesis. The research areas of parallel computing, Computational Geometry (CG), and Geographic Information Systems (GIS) are discussed. The body of research in those areas that forms the context for this thesis is presented. Specialized research in Vector Polygon Overlay (VPO) is noted. Finally, pertinent previous research done by the members of the PARADIGM Group at Carleton University is discussed, together with its influence on this work.

2.1 Parallel Computing

GIS has permanently outrun the sequential processor. This is due to the amount of data generated, the computationally intensive algorithms required, and the real-time demands put on GIS.

Both satellite and ground-based information-gathering technologies have progressed to the point where terabytes of data can be generated in a single day. Satellites can record ground-based data to a resolution below a meter. The planned Earth Observation System will collect terabytes of data daily [42]. The Space Shuttle's earth-mapping mission in February 2000 produced more than seven terabytes of data; this data will require a minimum of two years to process using current technology¹ [58]. Forestry

¹Even copying the data tapes recorded in-orbit required two months!

companies are now able to, using aerial and satellite images, manage forests down to the level of individual trees [45]. New, highly sensitive ground-based devices allow rapid measurement of large areas at a tiny scale [74].

Applications already exist that demand processing power beyond current sequential technology - applications such as airline route scheduling (which uses requires the use of Hamiltonian algorithms), the generation of weather reports, and predicting the movement of icebergs [49].

Many of the algorithms requiring a parallel solution are bound by real-time constraints. Predictions of iceberg movement must happen before the iceberg moves far, or the predictions become useless. New applications such as real-time World Wide Web queries on geographical datasets will push computer systems to do near-instantaneous computation.

The continued exponential increase of processor speed and capability will help. New miniaturization techniques and advances in molecular electronics will likely drive sequential processor improvement for another thirty years, producing processors perhaps a million times more powerful than currently available [44]. However, the amount of data, the complexity of its analysis, and the real-time limits placed on its analysis will demand performance above the level deliverable by sequential processors [21].

2.1.1 Challenges of Parallel Computing

The most important solution to the computing power crunch is parallel computing. Parallel computing solves a computation using more than one processor. This involves adding an extra layer of complexity, both to the hardware architecture used to build a computer, and to the algorithms and software used to compute the solution. This complexity manifests itself in several ways:

- ★ There are many different architectures for parallel machines. Algorithms designed for a specific architecture are often poorly suited to a different architecture². Algorithms designed for a general parallel machine may not perform adequately for the task in question.

- ★ There are many different ways to solve problems in parallel. Parallelization strategies that work well for one algorithm may not be relevant to another. Significant expertise in parallel algorithm design will be necessary to solve many problems in parallel.
- ★ Some problems are not parallelizable. That is, the inputs to each part of a computation are so dependent on the previous calculations that a sequential processing of data is enforced. Such problems are classified as *P-complete*³. Examples of such problems are the four colour index problem, and linear programming [54].

2.1.2 Definitions and Models of Parallel Computation

Although contemplated for more than a century [21], interesting parallel computers were first built in the 1970s. As different kinds of parallel computers were built, some method of classifying them was needed.

Flynn's Taxonomy

Flynn in 1972 divided computers into a basic classification that is still used today. Flynn's taxonomy contained four types of computer architecture [27]:

- ★ Single Instruction Single Data (SISD),
- ★ Single Instruction Multiple Data (SIMD),
- ★ Multiple Instruction Single Data (MISD), and
- ★ Multiple Instruction Multiple Data (MIMD).

SISD corresponds to the conventional sequential computer. The other classes are variations of parallel computers; of these, MIMD architectures are by far the most popular today. MIMD computers allow each processor to run many different subprograms on different data.

²This is a particularly important issue if one must port an algorithm from an obsolete parallel machine.

³Strictly speaking, P-complete problems are defined as problems that cannot be solved with a polynomial number of PRAM processors in running time $\leq O((\log n)^k)$, where k is an arbitrary constant [54]. See Section 2.1.2, below.

Hardware Architectures

There are many different hardware architectures for parallel machines, differing in the topology of the processor network and the number of connections available on a processor. These can also be classified as *static* or *dynamic* architectures. The advantage of the latter is that the connections between processors can be changed during the execution of the program.

Some of the most common static architectures are:

- ★ *meshes*, d - dimensional meshes of processors, typically with 2^d connections per processor.
- ★ *trees*, which form a tree graph, and are useful in algorithms that require multi-stage merging of results.
- ★ *hypercubes* (a mesh variant with d connections per processor).

Some of the most common dynamic architectures are

- ★ *bus systems*, in which all processors are connected to a common bus.
- ★ *multistage networks*, in which processors can independently connect to each other.

Parallel computers can also be classified as *distributed-memory* or *shared-memory*. Distributed-memory computers give each processor its own memory, and require communication to be done through special communications links.

Shared-memory computers let all processors access a common shared memory. Shared-memory computers are often faster than a comparable distributed memory computer, but are more expensive to build. Programs on a shared-memory system are in concept simpler to write; their algorithms can often be ported directly from sequential algorithms. Because the processors can write into the common memory, and interfere with each other, programs are usually harder to debug on a shared-memory machine.

Finally, parallel computers can be classified as *tightly-coupled* or *loosely-coupled*. In tightly-coupled machines, processors have fast, high-bandwidth access to each other and to memory; the processors in

loosely-coupled machines are more isolated. Thus the cost of communication is the dominating factor in loosely-coupled machines, in all but the most computing-intensive applications.

Parallel Machines

In the last ten to fifteen years, the distributed model of computing has become quite popular. Distributed, tightly coupled parallel machines such as Cray's T3D and IBM's SP2 dominate the high-end market. At the lower end, the "network of workstations" approach favors building loosely-coupled parallel machines from stripped-down versions of standard sequential computers. This is the approach taken by the Beowulf project [82], the PARADIGM Group's ASP-series machines, and many others.

The SETI project has created what is arguably the world's largest "parallel computer" by distributing data-crunching screen savers to over 1.3 million computer users worldwide as of October 1999 [4]. SETI obtains compute cycles from legions of sympathetic personal computer owners, that they cannot afford to pay for, to scan their vast datasets for signs of extraterrestrial intelligence.

With millions of computers and workstations sitting mostly idle in offices and homes around the world, much more can be done to harness distributed networks of processors. One effort to harness this idle compute power is COSM, a "set of protocols and applications designed to allow computers all over the world to work together on projects." [10] This open-source project aims to enable the sharing and selling of computing power across internets and organizational intranets. The success of such a venture could revolutionize scientific and industrial computing.

Other similar projects include the Globus project run by France's Argonne National Laboratory and USC, and Porivo Technologies, a for-profit distributor of computing cycles [70].

As hardware and software techniques develop and evolve, high-end shared memory machines will become more practical and more common. Examples of commercially viable shared memory machines include Nortel Networks' XA-Core (eXtended Architecture Core) telephone switch, SUN's Ultra 80 and 450 series workstations, and SGI's Power Challenge system.

Models of Computation

Several theoretical models of computation are of note. The most general model is the PRAM (Parallel Random Access Machine), developed by Fortune and Wylie in 1978 [28]. This model is analogous to a large shared-memory parallel computer in which all processors can access all memory locations simultaneously. Several flavours exist:

- ★ Concurrent Read Concurrent Write (CRCW)
- ★ Concurrent Read Exclusive Write (CREW)
- ★ Exclusive Read Concurrent Write (ERCW)
- ★ Exclusive Read Exclusive Write (EREW)

Algorithms developed for these models of computation are very general in nature, and can be adapted to more specific models of computation by transformation or simulation. However, such algorithms tend to work better on shared-memory and/or tightly-coupled machine architectures.

Distributed Models In the 1990s, models of computation were developed that dealt with both distributed and shared-memory models of computation. The first was the Bulk-Synchronous Parallel (*BSP*) model developed by Valiant in 1990 [85]. Dehne et al. published refinements to BSP as Coarse-Grained Multicomputing (*CGM*) in 1993 [18]. CGM is closely related to the BSP* model, introduced in 1995 [9].

The BSP model divides a parallel computation into *supersteps*. A BSP algorithm causes the exchange of data between processors during each superstep. After every superstep, with periodicity L , a synchronization takes place.

BSP algorithms model the computation time by the formula $g \times h + s$, where g is equal to the computation speed divided by communication bandwidth, h is equal to the number of supersteps, and s represents the startup costs. The BSP model allows more realistic consideration of distributed-memory

computers. Note, however, that the amount of data sent during each round is unbounded. The weakness of BSP is that one round of communication can represent a very small or very large period of time.

The CGM model refines the BSP model to bound the size of the superstep, or *round* of computation. Specifically, each round must have no more than $O(n/p)$ data sent to or from each processor; this means that the total amount of data sent by or received from a processor is $O(n)$ [18]. A CGM algorithm has λ CGM rounds, each being an *h-relation* with $h = O(n/p)$. The total algorithm has a communication cost of $O(\lambda gn/p)$. This gives the concept of *round* a meaningful definition. The division of a parallel algorithm into defined computation and communication phases is very useful.

For more information on parallel computers with application to GIS and computational geometry, please consult [21] [41] [75].

2.2 Computational Geometry

Geometry is an ancient discipline, dating back four thousand years to ancient Egypt and Greece [59]. With the advent of computers, geometric computation began to be done. The process of developing correct, robust, and efficient geometric algorithms started in several different fields: mathematics, computer graphics and animation, geography, geology, and other fields that used computers to perform geometric computation⁴.

Computational Geometry (CG) is a challenging field in that it combines the disciplines of algorithmic design and theory, numerical methods of computing, geometry, computer graphics, mathematics, and now parallel computing [33] [78]. The CG designer and programmer must be knowledgeable in all these fields (less perhaps parallel computing) in order to be effective. The definition of “Computational Geometry” was codified in 1975 by M. I. Shamos [72].

Three important CG problems are pertinent to this work. Line segment intersection (a.k.a. “Arrangement of Lines”) is the main sequential algorithm used in the vector polygon overlay algorithm. Planar

⁴This is so much so that similar or identical algorithms were often published by different groups of researchers in different fields, without knowledge of work in other fields.

map polygonization is the other part of said algorithm; it allows the linking of the intersected line segments to form a new polygonal map. Planar triangulation is an extra technique that may be useful in vector polygon overlay, to allow the simultaneous triangulation of polygons while creating a polygonal map.

Sequential bounds and algorithms for these problems are given below.

2.2.1 The Line Intersection Problem

Simply put, this involves computing the intersections (being points or lines) of two sets of line segments. The sequential line intersection problem has a lower bound of $\Omega(n \log n + k)$, where n is the size of the input set of line segments, and k is the size of the output number of segments. The quantity k can grow to as large as $\Theta(n^2)$ [72].

The well-known *plane sweep* algorithm was introduced by Bentley and Ottman in 1979 [11]. This algorithm causes line segments to be processed in a one-dimensional order, as if sweeping a line through a plane. It takes advantage of the behaviour of line segments along a sweep line. At each intersection point, the order of line segments along the sweep line can only change at the intersection point. Ergo, only the segments with extreme slope that emerge from an intersection need be tested. These are the segments that are adjacent to segments that don't intersect with them at the current intersection point, with their immediately adjacent counterparts along the line. This algorithm has an upper bound of $O(n \log n + k \log n)$.

Nievergelt and Preparata [60] developed an algorithm to solve the *red-blue* segment intersection problem. Here, a non-intersecting set of red line segments is intersected with another non-intersecting set of blue line segments, in $O(n \log n + k)$ time. This solves the subproblem of merging two convex planar subdivisions. The red-blue segment intersection problem is the central intersection problem of vector polygon overlay, although VPO must deal with non-convex planar subdivisions. Guibas and Seidel later demonstrated an $O(n + k)$ algorithm for merging two convex planar subdivisions.

Mulmuley [56] [57] developed a randomized algorithm for the general line intersection problem that works in $O(n \log n + k)$ time.

Chazelle and Edelsbrunner [16] developed an optimal $O(n \log n + k)$ algorithm for general line segment intersection, in “Twenty-One Easy Pieces.” This algorithm, claimed simple in execution but complex in concept, behaves well in practice.

Amato, Goodrich, and Ramos developed the first work-optimal deterministic parallel algorithm for line intersection[3]. It uses trapezoidation in parallel to complete $O(n \log n + k)$ work in $O(\log^2 n)$ time deterministically, or $O(\log n \log \log n)$ time in a randomized algorithm.

While relatively straightforward in concept, in practice large numbers of special cases, plus numerical difficulties, make the line segment intersection problem a difficult implementation.

2.2.2 The Planar Map Polygonization Problem

Given a graph containing n edges (line segments) that partition a plane, group them into simple polygons (if any). This has a trivial sequential lower bound of $\Omega(n)$; all one must do is traverse along each connected component of the graph containing the line segments to find cycles. Each cycle found is a polygon.

Note that each node of the graph must have its adjacent edges sorted in a circular order, for this to work. If this were not the case, cycles found may contain other polygons. Utilizing the list of sorted edges, one can always pick the edge adjacent to the one used to enter the endpoint node from, and thus ensure that a detected polygon contains no smaller polygons. Preprocessing a graph to sort the edge lists has a lower bound of $\Omega(n \log n)$; if we limit the nodes to a constant degree, we can achieve $\Omega(n \log n)$.

2.2.3 The Planar Triangulation Problem

Given a set of points, line segments, or simple polygons in a plane, triangulate them. That is, add edges to the existing graph until the entire graph (less the outer face) consists of only triangles. The problem of triangulating n points in a plane has a sequential bound of $\Theta(n \log n)$ operations [72]. This research will concern the problem of triangulating sets of line segments, which has a similar lower bound. Indeed, the latter problem is equivalent to completing a partial triangulation.

Both of these last two algorithms are not difficult in their sequential form; the parallel implementations discussed in Chapter 2 are of more interest.

Useful references in CG include [72] [57] [68] [12] [78].

2.3 Geographic Information Systems

Geographic Information Systems (GIS) are concerned with a sub-category of spatial data, namely geographic data. That is, the data in a GIS represents locations in space (and time). This usually means two-, two-point-five-, or three-dimensional space, possibly including a time dimension. Applications exist ranging from a trivial one-dimensional space up to four-dimensional space. Applications also exist for higher dimensions⁵.

Geographic data typically has attributes associated with an item of data. This could be, for example, information about what a polygon represents (“polygon 5269 represents a lake”), or attributes of the represented entity (“the road represented by arc 8376 is part of highway 17”). Attributes are a key part of GIS. Despite the fact that the important operations are performed on geometric primitives, it is the associated attributes that give those manipulations meaning.

2.3.1 Background & History

Computers became sophisticated enough to handle spatial data processing in the late 1950s. Cartographers began to use computers to automate the task of compiling and rendering maps at this time. This was also when computer graphics began to develop. The first major use of GIS technology was the Canada GIS, which was used to analyze Canada’s national land inventory, starting in 1963. Researchers at Harvard established their Laboratory for Computer Graphics and Spatial Analysis (LCGSA) in 1965. This lab produced spin-off companies that included ESRI, now the “leading application developer for the GIS industry.” [46]

⁵Higher dimensional spatial data is not classified as geographical.

GIS came into its own in the 1970s. Three factors drove this. First, computers had grown powerful enough to make analysis of GIS data cost-effective. Second, GIS software and spatial databases matured to the point of widespread usefulness. Third, the first generation of orbiting imaging and mapping satellites were launched, providing useful data. A GIS industry began to develop.

The use of GIS in the marketplace has grown steadily. With the introduction of the U.S. military's Global Positioning System (GPS) in 1985, GIS technology was enabled to be a precise guide anywhere on the planet. Increasing computer power and available data mean that GIS software is an essential tool for large institutions, governments, and corporations. With the recent introduction of the hyperspectral imaging spectrometer, an area can now be mapped at a one-meter resolution, sampled at 128 different wavelengths. Such detailed data allows researchers to discover information about an area that could not be determined by someone conducting a ground survey on foot [74]. The hyperspectral imaging spectrometer gathers that accuracy of data while passing over an area in an aeroplane.

The next stage of the growth of GIS is to the level of personal technology. Many GPS are already in use in automobiles, boats, airplanes, and by wilderness adventurers. Residents of Tokyo, confused by their labyrinthine street system, navigate by GPS. In the future, wireless telecommunications coupled with GPS and a powerful, wearable GIS (perhaps integrated into their stereoscopic virtual reality display) will enable anyone to know instantaneously where they are, what they are near to, who is near to them - and who in the vicinity wants to sell them something. GIS will become an essential part of our lives, just as the personal computer has over the last twenty years.

2.3.2 GIS Basics

The most basic division of GIS is by data representation. In a *raster* GIS, each point in space is represented as one item of data. One example of a raster representation is a bitmap image. In a *vector* GIS, each distinct geographical entity (or its constituent parts) is represented as a single item of data. One example of a vector representation is a contour line (or, *isoline*) map.

Maps also have different forms of data representation. Two important map data representations are that of the isoline and the *choropleth* map [47]. The isoline map represents differences in the value of an

attribute with lines that indicate where on the map the attribute has the same value. Isolines are often used for maps of continuous data, such as elevation or air pressure data. The choropleth map represents differences in the value of an attribute by dividing a map into regions; the boundaries indicate where the attribute value changes. Choropleths are often used for maps of discrete data, for example to indicate differences in soil type. *Choropleth* is essentially a geography-derived term for the partition of a plane into a planar map.

This research uses choropleth-based data from *Digital Elevation Models* (DEMs), specifically the *Triangulated Irregular Network* (TIN) data structure. Note that the geometric overlay and vector polygon overlay operations are not limited to DEMs; they can be used on a wide variety of data. The following describes the data models and structures used in this research:

A DEM is simply a finite, discrete function in two variables: $f(x, y) = z$. Here, x , y , and z denote coordinates in a three-dimensional universe. Note that for every pair (x, y) there can only be one value of z . This is referred to as a 2.5-dimensional model. A true three-dimensional model can specify an arbitrary number of z values for every (x, y) pair. DEMs therefore cannot represent every possible terrain; underground caves or overhanging cliffs are impossible to represent in a 2.5-d model.

There are many kinds of DEM; among the most popular are *regular square grids*, isolines, and TINs. TINs represent a terrain by a series of triangles. The triangle representation allows a terrain to be represented by a continuous mesh of triangles. Its advantage over other polygon meshes is that non-triangle meshes do not guarantee a continuous surface [47]. At the same time, TINs allow for abstraction of data (and thus a smaller data file) over areas of a terrain where the change in elevation is linearly monotonic. This is an advantage over a regular square grid, which represents every point in the grid of a terrain, thus requiring considerably more data storage.

2.4 Vector Polygon Overlay

This section describes Vector Polygon Overlay (VPO), which is a subtype of generic overlay.

2.4.1 The Generic Overlay Operation

Overlay is one of the fundamental algorithms of vector-based Geographic Information Systems. Overlay is “an analysis procedure for determining the spatial coincidence of geographic features” [26]. In general, this means the overlay of one set of geometric primitives on top of another. Two sets of primitives are intersected, and new primitives are derived from the intersection. These derived primitives can be of the same type as the input primitives, or they can be different. For example, when intersecting two sets of line segments, one can derive a set of intersection points, or a new set of mutually disjoint segments, or the partition of the plane made by the intersecting line segments.

Overlay in its generic form is a very abstract algorithm. As a model of geometric computation, it is powerful enough to compute the set of *binary operations*. Binary operations are operations $A \times B = C$ on two sets of spatial data A and B [63]. In order to come up with a concrete implementation, one must specify many details about what is being overlaid, how to compute the overlay, and what is to be derived from the overlay.

For example, it is not immediately obvious what overlay means at the level of overlaying individual primitives. There are many different ways of deriving new primitives from the input sets; most of these have analogues in set theory. Three of the most popular are [25] [63]:

- ★ **Union.** Given two sets of primitives, we want any result that derives from an operation on either or both sets. This is analogous to a logical OR operation.
- ★ **Intersection.** Given two sets of primitives, we want only results that derive from an operation on both sets. This is analogous to a logical AND operation.
- ★ **Difference.** Given two sets of primitives, we want only results that derive from an operation on exactly one set, but *not* both sets. This is analogous to a logical XOR operation.

Of course, one must still define what union, intersection, and difference mean in the context of the particular primitives and data attributes that one is dealing with! Building a generic overlay driver is not a trivial task; one must build a sparse framework in stages and let implementors fill in the details

as they go along. Deciding what constitutes generic overlay and what belongs to more specific sub-algorithms (and sub-sub-algorithms) is the major challenge and the most critical aspect of the design of OVID in the long run.

2.4.2 Vector Polygon Overlay

Vector Polygon Overlay (VPO) is the overlay of one vector-based planar polygon map on top of another. This operation is widely used in GIS, in a variety of disciplines and industries ranging from cartography to geology to transportation to ecology. The algorithm is built on a line- or polygon- intersection algorithm, and so has an $O(n^2)$ upper bound. Because of the order of the algorithm and the increasing size of GIS datasets, parallel solutions to the VPO problem are needed.

Two ubiquitous examples of VPO in GIS are:

- ★ Clipping: extracting a fixed window of data from a spatial data file. This is typically done by overlaying a single rectangle (or other window shape) with a file of geometric primitives. An optimal rectangle clipping algorithm runs in $\Theta(n)$ time [80].
- ★ Buffering: creating a buffer around a geometric entity. This can be done by overlaying offset copies of certain geometric entities onto the entire set. The optimal time here will differ depending on the application in question; a general algorithm will be $O(n^2)$.

Background

Vector Polygon Overlay was first worked on by ESRI and researchers at Harvard's LCGSA[88]. The VPO algorithm implemented in the Harvard group's WHIRLPOOL system is based on intersecting individual *chains* of non-intersecting polygon edges. The number of intersection tests is reduced by use of a linear sort; the actual intersections are done via plane sweep. Improvements to their intersection technique have been published by, among others, Nievergelt and Preparata[60], and Chazelle and Edelsbrunner [16] (see Section 2.2.1).

Franklin et al. [29] introduced the *uniform grid* method in 1989. This method divides the polygon-space into uniform grids; pairs of geometric primitives are only tested against each other if they fall into the same cell in the grid. This method works best on uniform distributions.

Orenstein [67] developed an overlay algorithm based on Morton-ordering, in which objects are approximated by bounding boxes described by the Morton-ordering. The boxes are sorted according to Morton-ordering; this is a one-dimensional sort. Orenstein therefore reduces VPO to a type of merge-sort on sorted linear lists.

Van Oosterom [86] developed an efficient sequential algorithm based on Guttman's R-trees [35], a specialized spatial index structure. R-trees are d -dimensional, height-balanced trees that store geometric data in bounding boxes. Each node in the tree is a bounding box; a parent node fully contains its children. R-trees therefore represent an efficient bucketing of geometric primitives somewhat akin to Nussbaum's Multidimensional File Partition (cf. Section 3.1).

Parallel Algorithms Parallel algorithms were pioneered by Atallah and Goodrich [7], and by Franklin using a variation of his uniform grid method, which is highly parallelizable. Atallah and Goodrich's algorithm featured the use of a binary tree to linearly sort objects and distribute contiguous portions of the data to parallel processors. This technique is called *strip decomposition*.

Wang [87] in 1993 introduced a parallel, shared-memory version of White's original algorithm. This is based on parallelization of operations within nested loop structures, and does not port well to loosely-coupled or distributed architectures[22].

The University of Edinburgh, U.K., GIS group developed in 1998 a VPO algorithm based on efficient task-sharing architectures and data distribution by strip decomposition[22] [73].

This topic is covered in more detail in Section 3.2.

2.5 The ATLANTIS GIS

This work is done within the context of the Paradigm Group's Parallel Spatial Modeling Environment. Within the PSME, PARADIGM has designed the ATLANTIS GIS. ATLANTIS is an integrated GIS toolkit giving applications programmers access to a powerful, parallel GIS applications framework.

In essence, ATLANTIS is *middle-ware*, a software framework⁶ that facilitates the task of parallel GIS application-building [75]. Applications implementors are able to concentrate on implementing the domain-specific model of their application, and allow the system to take care of difficult features. Such features include geometric algorithms, parallel messaging, load balancing of parallel nodes, file storage, numerical and topological accuracy issues, and data structure design.

2.5.1 The Composition of Atlantis

ATLANTIS is comprised of NEMO, Paradigm's raster-based GIS system [41], plus a vector-based GIS system. NEMO is a parallel NEighbourhood Modeling system. These two subsystems are implemented in the form of a set of drivers. These drivers enable users to implement most GIS applications on top of one or more of them; this is illustrated in Figure 2.1. The Atlantis drivers (as of publication) are [62] [63]:

- ★ Propagation Driver (PD). This driver implements vector-based propagation operations, i.e. operations where one entity affects those directly connected to it. It is well suited for operations such as TIN shortest path, Voronoi diagrams, etc.. It also contains software useful for implementing operations on the Nussbaum partition tree files, in other drivers. (This is not to be confused with the NEMO raster-based propagation driver [41].) Nussbaum and Lanthier built their parallel shortest path software on top of the PD [48] [61].
- ★ Overlay Driver (OVID). This driver supports an important subset of binary GIS operations. Binary operations are those involving the superimposition of two input data sets; this generates a

⁶I do not use *framework* here in the technical sense used by object-oriented software engineers. Rather, I refer to any software architecture that sets out a structure around which users build applications.

third data set as a result.

- Note that multiple data sets can be processed by repeated binary operations, providing that the result data set can be used as input into the binary operation in question. One example of this would be an overlay of maps - for instance, combining a road map with a map of waterways.
- ★ Transformation Utilities. These utilities convert data from one type of data to another. For example, points representing height measurements can be transformed into contour lines. Or, vector data can be converted into a raster format. These utilities will incorporate currently available NEMO file conversion utilities. They may also serve to pre-process data for the other Atlantis drivers.

Two components have yet to be developed as of this writing. These are the Visibility Driver (VD), a specialized propagation driver that generates the visible area of a surface, when given as input a surface and a viewpoint, and the Association Driver (AD), which processes unary vector GIS operations such as point-location or nearest neighbour queries. Note that the Overlay Driver model of computation contains the Association Driver model of computation.

These vector-based drivers join the NEMO drivers to form a comprehensive set of GIS drivers. The NEMO drivers are:

- ★ the Propagation Driver (PD), a driver that processes individual active cells in a raster image,
- ★ the Cell Driver (CD), a Cellular Automata driver that processes the entire image repeatedly (see [75]),
- ★ the Neighbourhood Driver (NAN),
- ★ the Map Algebra Driver (MD).

Note that many GIS applications may be implementable on more than one of the drivers. Each implementor must decide, based on the characteristics of the various drivers, which one is best suited to their task.

Figure 2.1: ATLANTIS Architecture Diagram

Within the above framework, the Overlay Driver is required to support a large variety of operations. Some of these are TIN overlay, line intersection, general polygon overlay, contour line intersection, and weighted Voronoi diagram generation. This dictates that the Overlay Driver have a very open, general architecture, to enable it to handle a wide variety of applications.

2.5.2 The NEMO Cell Driver and its Influence on OVID

The NEMO system predates the creation of ATLANTIS' vector-based GIS. The PARADIGM Group gained expertise in implementing the NEMO drivers and operations on top of them, that was applied to the creation of ATLANTIS. In particular, work on the NEMO Cell Driver (CD) was an important learning step towards the development of OVID. Our work was published in the December 1999 special edition of Future Generation Computer Systems [37].

Figure 2.2: NEMO Communication Diagram

The first contribution for me of the Nemo CD was its confirmation of the Paradigm group's vision of transparent parallelism. The software framework set up for the Nemo CD made developing parallel Cellular Automata (CA) applications as easy as developing a stand-alone sequential application. All that was required was functions that specified the behaviour of the CA on a sequential basis.

The second contribution was its emphasis on algorithmic compatibility with the hardware architecture. The CD was written for an ALEX AVX multicomputer, which is built on a two-dimensional mesh architecture. When we implemented a three-dimensional Cellular Automata application (the "dendritic grain growth", or crystal growth, application), on top of the CD, both the parallel performance and simplicity of application design benefited from the continued use of the 2-D mesh structure of the CD.

The third was the importance of numerical robustness in my calculations. While crystal growth was achieved with home-spun implementations of geometric calculations, I never achieved numerical stability until I ported the growth algorithm to use LEDA. Thus, while some exploration was done with custom line intersection algorithms, LEDA was an easy choice for the OVID geometric calculations.

The fourth was the importance of building a flexible software architecture. During (or shortly after) the

construction of the crystal growth application, it was necessary to change three different parts of the application and/or the driver. These were the geometric software library, the parallel communication library, and the underlying compute platform. The flexibility of the Nemo architecture (see Figure 2.2) enabled a relatively smooth transition through these changes, and contributed directly to the success of the project.

Finally, the importance of being able to handle very large dataset sizes was highlighted. The 3-D crystal growth application was capable of generating rasters much larger than had been anticipated when Nemo was first developed. This contributed to the implementation of an overlay driver that was able to deal easily with huge files and computations by breaking them down in an MFP structure and swapping them out to disk.

Chapter 3

Theoretical Approach.

Chapter 2 describes the theoretical basis for this research. It provides a theoretical background for current and future practical research. The MFP data structure, central to this research, is explained. Theoretical aspects of vector polygon overlay are presented, as is a summary of previous research approaches to this problem. Important algorithms are introduced for the MFP data structure, that will be used for practical implementation in later chapters.

3.1 The Multidimensional File Partition

OVID is built using Nussbaum's *Multidimensional File Partition* (MFP) file format. The MFP is the primary method of division of data between processors in the parallel implementation. OVID also serves as an important validation of the effectiveness of MFP's load-balancing properties.

This section describes the design intent, design, properties of, and techniques of algorithm construction with, MFP.

3.1.1 Design Intent

MFP [61][62][63] is a spatial, parallel data structure. It's primary purpose is to store spatial data in a format that is efficiently and effectively used by parallel computers running a wide variety of geometric

and GIS-based algorithms. The MFP is not intended to handle rare worst-case scenarios or theoretical perils; rather, good overall performance in an industrial setting is desired.

The files that are stored in the vector structure are already organized spatially. Namely, all the items in the files are already partitioned and assigned to the appropriate pages. One of the most important properties of the partition is the fact that it can support clusterization. Clusterization occurs when items (entities) are not evenly distributed within the domain e.g. a province street data contains clusters in cities [66].

Each page is assigned to a processor in a parallel machine, so that adjacent processors contain adjacent pages of spatial data:

We maintain spatial proximity between the nodes by ensuring that neighbouring buckets of the same size will be mapped to neighbouring processors, or [if] the neighbouring cells are of different sizes, then the mapping between neighbouring processors will be along the separating dimension [62].

Nussbaum specified several required characteristics for the MFP data structure [62]:

- ★ The ability to utilize the **best sequential algorithms**.
- ★ **Ease of Manipulation**: “an operation on the data structure will retain the data structure characteristics without reconstructing it from scratch.”
- ★ **Cooperativeness**: “several structures can be manipulated at the same time,” for example in a binary operation.
- ★ **Memory-Efficiency**: the data structure “should minimize the duplication of data.”
- ★ **Flexibility**: the data structure should allow “on-the-fly conversion” to another data structure.
- ★ **Adaptiveness**: the structure can adapt to efficient function on different machine architectures.

- ★ **Uniprocessor-Friendliness:** the structure must support “efficient manipulation by a single processor computer.”
- ★ **Load-Balance:** the structure must promote balanced distribution of geometric primitives (and therefore work) to the processors.
- ★ **Spatial Proximity:** the structure should maintain, as much as possible, spatial proximity between neighbouring nodes. I.e., processors owning adjacent pages should also be adjacent in the mesh.

It is difficult to build a data structure or algorithm that is effective and efficient on many parallel machines and architectures. Although the MFP is designed to be adaptive, its primary design target is a MIMD machine with the following characteristics[62]:

- ★ The processors are arranged in a 2D (or 3D) mesh.
- ★ Each processor has four (or eight) links available to it in the mesh.
- ★ The mesh dimensions are $p = 2^a \times 2^b$, where a and b are both integers. (In the 3D case, we must have $p = 2^a \times 2^b \times 2^c$, $a, b, c \in I$.)
- ★ The number of processors is small relative to the amount of data, i.e. $p \ll n$.

While still adaptable to other hardware architectures, this machine model works well on distributed and/or loosely-coupled architectures, such as networks of workstations. Algorithms that work well on this machine model are generally classified as coarse-grained or CGM algorithms. In CGM algorithms, the best sequential algorithms are run on each processor’s subset of the data; the per-processor sequential solutions are then merged in parallel to generate the complete solution.

In particular, ignoring the restrictive power-of-two mesh dimensioning does not significantly affect performance in many applications.

3.1.2 Building an MFP

The MFP is a general data structure. Its formal definition is for multidimensional spatial data. For the purposes of this research, a two-dimensional realization of the MFP is adequate. First, I will give a two-dimensional definition applicable to my implementation.

Given a 2D parallel mesh containing $p = m \times q$ processors, a vector data file is divided into a corresponding mesh of p rectangular regions, known as *pages*. Given that the vector data file has dimensions of $v \times w$, each of the p pages has dimensions of $\frac{v}{m} \times \frac{w}{q}$.

Assume a threshold, t , which the number of geometric primitives g_i in each page cannot exceed. If $g_i > t$, for any page r_i , $0 \leq i < p$, then we recursively split that page (the parent) into p subpages, or *children*. This continues until all pages contain less than t geometric primitives. Note that t must be chosen carefully; an overly small t could result in excessive, or even infinite, recursive splitting. Because an MFP page contains each object that intersects it, there is a limit to the number of times recursive splitting can be done.

For that matter, cases exist – such as circle sliced in triangular wedges that meet in the center – that defeat any $t < O(n)$. These are extremely rare in practice. If necessary, the threshold t can be made dynamic in order to address such cases.

Formal Definition

More formally, a vector file containing d -dimensional spatial data partitioned by a d -dimensional processor mesh is divided into $p = \prod_{i=0}^{d-1} |H_i|$ pages, with one page assigned to each processor. H_i is the length of the processor mesh in the i -th dimension [62].

Each processor in the mesh can be defined by the d -tuple, $(p_0, p_1, \dots, p_{d-1})$, where $0 \leq p_i < H_i \vee_i, 0 \leq i < d$. Similarly, each child page of the parent can be designated by the d -tuple $(c_0, c_1, \dots, c_{d-1})$, where $0 \leq c_i < H_i \vee_i, 0 \leq i < d$.

Since each page in an MFP is recursively decomposable into p more pages, a mapping must be defined to determine which processor owns what page. A trivial, straightforward mapping assigns each page

to a processor using row major order. However, such a mapping does not address the need for spatial proximity.

In order to maintain spatial proximity between adjacent nodes, the following scheme is used:

- ★ At level 0 of the MFP, one page exists, which belongs to processor 0 of the set of processors $\{0, 1, \dots, p-1\}$. This is the parent page; when necessary it is subdivided into child pages.
- ★ Each mapping considers a parent page owned by processor $f_i = (p_0, p_1, \dots, p_{d-1})$. The processor identifier f_i is an integer in the range $\{0, 1, \dots, p-1\}$ uniquely identifying the processor. The tuple $f_i = (p_0, p_1, \dots, p_{d-1})$ breaks the processor id into components describing its coordinates in the d -dimensional mesh¹. Similarly, each child can be defined as a tuple $(c_0, c_1, \dots, c_{d-1})$ corresponding to its position in d -space relative to the parent page.
- ★ For each parent page at an odd level, the *forward mapping* is used. In the forward mapping, child page $(c_0, c_1, \dots, c_{d-1})$ is mapped to processor $((c_0 + p_0)|H_0|, (c_1 + p_1)|H_1|, \dots, (c_{d-1} + p_{d-1})|H_{d-1}|)$.
- ★ For each parent page at an even level, the *reverse mapping* is used. In the reverse mapping, child page $(c_0, c_1, \dots, c_{d-1})$ is mapped to processor $((p_0 - c_0)|H_0|, (p_1 - c_1)|H_1|, \dots, (p_{d-1} - c_{d-1})|H_{d-1}|)$.

An example of a two-dimensional level 3 MFP with 4 processors in a 2x2 mesh is given in 3.1. The numbers given below are the level at which the page in question resides.

Figure 3.2 (courtesy of Lanthier [48]) shows the division of a vector file into four levels of a 3x3 MFP.

3.1.3 Theoretical Considerations

The MFP structure nicely compartmentalizes geometric problems into separate sets of data that can be loaded into main memory and processed independently of each other. Parallel merging can then be

¹The processor identifier f_i can be calculated from its tuple $(p_0, p_1, \dots, p_{d-1})$ by using the following formula: $f_i = p_0 + p_1|H_0| + p_2|H_0||H_1| + \dots + p_{d-1} \prod_{i=0}^{d-2} |H_i|$.

3	3	2	2	2	
3	3				
2		2	2	2	
1			3	3	2
			3	3	
			2		2

Figure 3.1: Page Level Designation in a 2x2 MFP.

done by messaging. Note that this works perfectly well for a single sequential machine as well as pages distributed among a number of parallel nodes. This *virtual processor* technique allows great flexibility for the programmer.

Recall that the metric for CGM algorithms is the number of communication rounds required by an algorithm to complete. For a CGM-style geometric algorithm, with processors having no dependency on neighbouring data, running on a square mesh hardware architecture, the algorithm may have as low as $O(1)$. This is also the cases for a switched network architecture.

This changes when an algorithm is *neighbour-bound*, i.e. has a dependency on neighbour-to-neighbour communication. In this case, an algorithm running on a square mesh containing p processors has a general upper bound of $O(\sqrt{p})$ communication rounds involved in propagating data from one end of the mesh to the other. In d -dimensional space, the upper bound on propagation is $O(\sqrt[d]{p})$.

Figure 3.2: Levels 0 to 3 of a 3x3 MFP

The maximum propagation on an MFP requires a more complex analysis. The number of propagations depends on the size and topology of the mesh, the number of processors, the maximum level l of the mesh (which depends indirectly on the number, n , of geometric primitives and their distribution, and the threshold t^2), and the geometric object being considered. Because a single processor owns one page in every subdivision, the cost of a broadcast operation is also $O(\sqrt{p})$. Algorithms not dependent on neighbour relationships between pages should be able to reduce the number of rounds accordingly.

In general, given an MFP with p processors, l as the maximum level of the structure, we can expect $O(f(p,l))$ communication rounds in a CGM algorithm. This is a generic order because the algorithm also depends on the object being processed.

Processing Line Segments on an MFP

If we wish to be more specific, we must specify a geometric object. If, for example, we are processing line segments, then we must consider the path of the line segment across the pages.

In the given case, the most pages a single line segment could cross would be $O(p^{\frac{1}{2}})$. In the case of a uniform distribution, $l = O(\log(\frac{n}{t}))$, so the algorithm has order $O(\sqrt{np/t})$ communication rounds. In a worst-case distribution, with each line segment stretching diagonally from near one corner of the vector file to near the other, $l = O(\log(n^2))$; in this worst case, the algorithm has order $O(n\sqrt{p})$ communication rounds.

This worse case is extremely rare. Indeed, it is almost meaningless to consider it, since each processor must contain almost all of the line segments anyways; each processor ends up performing the sequential algorithm on the majority of the data.

When considering other geometric primitives and other problems, algorithms may have other average and worse cases.

² t is generally a function of the memory capacity of a processor, and is implementation- and machine- dependent in that sense.

3.1.4 Algorithm Construction with MFP

The MFP is a *virtual processor* strategy, because it divides a problem into k smaller parts that can be solved by using p processors, $1 \leq p \leq k$.

In any MFP algorithm, two basic phases exist: the sequential solution of individual pages of data, and the parallel solution of the entire problem done by merging the sequential solutions. This contrasts with an *external memory* solution, which deals with large problems, sequential or parallel, by efficient use of virtual memory. Both solutions are highly affected by locality of data, however.

To construct an MFP algorithm, one must first find an efficient sequential algorithm for the first phase. If necessary, the algorithm may be modified to better suit the needs of the parallel algorithm. Modifications may be made to reduce the amount of inter-processor communication necessary, or to produce data necessary or useful in the parallel merge phase of the algorithm. For example, in a geometric algorithm, reducing the number of objects connected to objects in other pages will normally reduce communication time.

Modifications may also be made to interleave stages of the sequential algorithm with the parallel algorithm; this will normally speed up the algorithm, since the parallel communication phase does not have to wait for sequential computation to complete in order to start. If the algorithm is communication-bound rather than computation-bound, phase interleaving will not be useful.

The second phase of an MFP algorithm is the parallel linking phase. Depending on the algorithm, there may be only one set of parallel operations to perform, or there may be a number of different tasks. Tasks independent of each other can be performed concurrently (during the same round of communication), while sets of dependent tasks must be performed consecutively.

In general, the second phase of an MFP algorithm works as in Algorithm 1.

In many cases, the parallel linking phase of MFP algorithms are neighbour-bound, and therefore restricted to a lower-bound of $O(\sqrt{p})$ rounds of communication. In some cases, one can do better and still meet the CGM criteria.

For example, consider an algorithm implemented on a switched-network architecture machine. Here,

Algorithm 1 MFP Geometric Primitive Resolution

for each dependent task, **do** on each processor:

 While all geometric primitives for this task are not resolved, **do**:

 1. For each MFP page, **do** (in parallel, with p processors) :

- (a) Take information about all unresolved geometric objects or primitives, and send them to the processor that needs to know about them. In most cases, this processor owns a page adjacent to the one owned.
- (b) Receive information about unresolved geometric objects or primitives that this processor owns.
- (c) Update the state of each object, and determine if each owned object is resolved. The task is done on this processor if all owned objects are resolved.
- (d) Perform any further sequential processing necessary between communication rounds.

endfor

mesh-computer restrictions such as neighbour-boundedness no longer apply. If the number of primitives being linked by a page, p_l , follows $p_l \ll O(\frac{n}{p^2})$, then all information about unlinked primitives can be broadcast to all other processors in one round of communication.

Such an $O(1)$ algorithm will likely require a significant amount of extra sequential processing during that round, such as sorting primitives, assigning them to the appropriate page(s), etc.. Such a strategy may be worth the extra implementation, and complexity of sequential algorithm, in cases where communication dominates computation. For example, network-based parallel machines may benefit greatly from this.

3.2 Parallel Vector Polygon Overlay

Vector Polygon Overlay is the overlay of two planar polygon maps. It thus requires the intersection of the polygons. Background material is given in Section 2.4.

3.2.1 Sequential Algorithms

Sequential VPO algorithms have five main phases [88]:

1. Intersection test reduction.
2. Intersection calculation.
3. Creation of new polygons.
4. Attribute Linking.
5. Topological Correction.

Intersection Test Reduction

In all but a few known bad cases, a polygon or a line segment will intersect only a constant or logarithmic number of the polygons or line segments in the other set [12]. Reducing the aggregate number of tests that have to be done from $O(n^2)$ is quite easy. Reducing the tests to the order of the actual number of intersections ($O(n)$ or $O(n \log n)$), thus producing an *output-sensitive* algorithm, is more difficult.

Because we are dealing with spatial data, and in particular planar polygon maps, most kinds of inputs will produce intersection patterns that are local to a given area. To eliminate distant line segments from consideration, a one-dimensional or spatial sort is quite effective. Techniques include [22]:

- ★ **Brute Force.** For each pair of primitives, perform an intersection test. This is trivial and inefficient.
- ★ **Spatial sorting.** Types of spatial sorts include:
 - **Full Sort**, in which primitives are sorted according to geometric coordinates. This can be time-consuming, but it allows a more efficient conclusion of the overlay.

- **Bucket Sort** divides geometric primitives into many collections of more closely related primitives. This eliminates most of the non-intersecting pairs. Franklin's uniform grid method [29] is a specialized bucket sort. This approach has its weaknesses:
 - Regular decompositions such as meshes or regular strip decompositions are vulnerable to clustered data;
 - Most of the primitives may end up in a small fraction of the buckets.
 - Large numbers of long line segments or clusters on the bucket borders may degrade performance over brute force in the worst case.
 - **Quadtree decomposition**, a type of bucket sort, divides an area by bisecting each dimension. An area need only be divided if an overly large number of primitives fall within it; this allows a more even treatment of clusters. MFP is an enhancement of Quadtree decomposition.
 - **Space-Filling Curve Sort**. Orenstein [67] employed this technique. Primitives are sorted into variable-size buckets; the buckets themselves are ordered by the Morton key (or other such keys). Map overlay is then similar to a merge-sort using sorted linear lists [86].
- ★ **Bounding Boxes**. When adding a line or polygon to the current overlay structure, a quick test using bounding boxes can avoid many unnecessary expensive geometric tests. Its weakness is cases where most of the primitives in the given dataset do intersect.
- ★ **Plane sweep**. This well-known technique combines one-dimensional sorting with a well-defined processing scheme that is easily parallelizable. By considering only primitives that intersect the sweep line, and then of that set only those that are adjacent along the sweep line, intersection tests are done on a small superset of the primitives that actually intersect.

Techniques for intersection test reduction are varied, and are often combined to produce maximum effect. For example, an efficient algorithm could be produced by first performing a quadtree decomposition, then fully sorting each bucket before performing a plane sweep. In the plane sweep, the

bounding box technique can be used to save time over performing a full intersection detection on each pair of line segments.

Intersection Detection

Intersection detection is simple in concept but brutal in practice when special cases, numerical problems, and degeneracies are taken into account. Simple rules are the exception here; robust and useful algorithms must deal with a variety of cases, which are problem-specific. Very often, degenerate cases compound this problem. Implementations have this trouble compounded by numerical problems, which can introduce topological errors into a dataset. Taken together, they present a formidable challenge. This is covered in more detail in Chapter 3.

Creation of New Polygons

New polygons are created from the intersections and line segment primitives generated in phase 2. Algorithms such as **union**, **intersection**, and **difference** are applied in order to produce a map that consists of either a union of the polygonal maps, an intersection, or the difference of one from the other.

Attribute Linking

GIS systems contain an amalgam of spatial data and attribute data. Having processed the spatial data successfully, the attributes belonging to the overlaid sets of spatial data are now merged together. Interpretation of what the attributes mean in relation to each other and to the new dataset must be done; this is often application-specific, and must often be specified by the applications programmer or end-user.

Topological Correction

If required, topological correction must be done. Ideally, this is not due to numerical errors. It is much preferable to generate numerically robust solutions. Numerical errors can cause substantial changes in

topology, so it is better to avoid these [53].

A treatment of *splinter polygons*, polygons that are very long and narrow, is usually necessary. Splinter polygons are natural byproducts of polygon overlay. They tend to complicate the analysis of data, and can make further operations on the data numerically hazardous or complicated algorithmically. Since changes in topology affect the meaning of attribute data both in the area of the polygon map and in the global context, the splinter polygon policy in an algorithm needs to be customizable by the end user.

Preprocessing may also be done to stabilize the data for further geometric operations, by removing potential problems such as a vertex lying extremely close to (but not on) a line. Techniques that can be used here include vertex splitting and edge cracking [53]. These may also be done as preprocessing operations to VPO.

3.2.2 Design of a Parallel VPO Algorithm

The parallel vector overlay operation is divided into several main steps. Generally speaking, there are six parts to a vector polygon overlay algorithm (cf. [22]):

1. **Extraction and Sorting** of the necessary data from the input datasets.
2. (a) **Partitioning** and (b) **Distribution** of the data to parallel nodes. Partitioning usually amounts to a spatial sort.
3. **Translation** of all data sets to the same coordinate range (in parallel).
4. **Intersection** of the geometric primitives (in parallel). These are usually line segments and/or points.
5. **Linking**. This is the process of linking primitives together to form polygons or other higher-dimensional constructs (in parallel). Attribute data can also be linked here.
6. **Output** of new geometric data (line segments, polygons, etc.), including attributes derived from the processing of old data.

The critical parts of the parallel algorithm are partitioning, intersection, and linking. Partitioning is critical for load-balancing and the efficiency of the parallel algorithm. Parallel merging of intersections is critical on loosely-coupled or distributed machine architectures, where communication is critical, as is parallel linking.

Most of the available parallel algorithms measure the performance of the algorithm over all six steps, or at least the first five.

MFP as a Geometric Database

Because the MFP is a ready-made file format for parallel GIS databases, the creation of MFP files accomplishes several tasks for us. The MFP contains geometric primitives that are already x-sorted, fulfilling part 1. The primitives are partitioned and distributed to per-processor files, fulfilling part 2. Part 3 is not explicitly done during the MFP preparation process, but the translation must be done by the GIS user if the file coordinate ranges are not identical.

The MFP is intended as an efficient, general format for the internal representation of GIS data. Once the data is stored in an MFP format, parts 1 and 2a no longer apply to performance considerations; that part of the algorithm is considered, in the MFP model, as a one-time conversion cost. The portion of the general algorithm pertaining to this research is parts 2b through 6. Of these, part 3 is trivial and can be done by file conversion utilities sequentially or in parallel.

Finding Overlay Page Assignments

Before intersecting any primitives, the MFP algorithm for Vector Polygon Overlay must determine the structural mapping between polygon maps A and B. That is, it must determine which pages from polygon map A to overlay onto the pages of polygon map B.

Presuming that the A-B pairing is not known in advance, the structural mapping between pages in map A and map B is also not known in advance, and therefore part of the MFP overlay algorithm is real-time distribution of data. This determination is not trivial, but the MFP has useful properties that make it less difficult.

Recall that each processor owns only a small subset of the pages within an MFP file. Even so, the processor can use its set of pages from both files to determine the overlay structure required. Within a single page-subdivision of an MFP at level l , each processor must own exactly one page at that level. If the page owned at level l is subdivided itself, then the processor must own one of the child pages at level $l + 1$.

For each page owned by the current processor in polygon map A, we know that one of four conditions holds:

1. The current processor owns a page in polygon map B that corresponds exactly to the page in question. The subdivision of polygon map B matches the subdivision of polygon map A exactly at this point in the file. In this case, we simply overlay the two pages.
2. The current processor owns at least one page in polygon map B that fits inside the page owned by map A. The owned page in map A is an *ancestor* page for the page in map B. We hence overlay the map B page on top of the map A page.

★ Furthermore, we know that the map A page is required for overlaying with map B pages on all processors, since the subdivision of map B is at a finer granularity than that of map A. We thus must send (or, *push*) the map A page to all other processors.

3. The current processor owns exactly one page in map B inside which the map A page fits. This case is identical to case 2, with the polygon map designations “A” and “B” reversed.
4. The current processor owns no pages in map B which correspond to the page in map A. Some other processor therefore owns an ancestor page which it must push to us; we must wait for such a page.

★ Similarly, there may be pages in map B with no corresponding pages in map A to overlay with, and the current processor must receive the corresponding ancestor page from another processor.

Figure 3.3: Overlay Algorithm: Finding Matching Pages

These cases are illustrated in Figure 3.3. The shaded areas indicate pages owned by the current processor. The page-matching algorithm is given in Algorithm 2.

Note that, once the “pushed” pages are received by all of the p processors, every processor has the pages it needs to complete the necessary overlay operations on the pages it owns. We can then proceed to part 4 of the general VPO algorithm on each owned page, the intersection of geometric primitives.

Analysis of the Page-Matching Algorithm

Recall the CGM criterion that no processor may send more than $O(\frac{n}{p})$ data in a single CGM round. The page-pushing algorithm above does not meet this criteria. For the pathological case, where one processor owns most of the data, one processor can end up broadcasting $O(n \cdot p)$ data primitives.

For a pathological (but load-balanced) case, with one processor doing all of the broadcasting, one

Algorithm 2 Page-Matching Algorithm for Dynamic Data Distribution

for each of the p processors **do**

- ★ According to the 4 listed conditions, push the required pages to all of the other $p - 1$ processors.
- ★ Receive all pages sent to the current processor by the other $p - 1$ processors.
- ★ Categorize each page (owned by or received by the current processor) so that its partner page in pairwise overlay from the other polygon map is known.
 - This can be accomplished by a point-location test on the owned and received pages [61].

end for

processor may end up broadcasting a volume of data approximately equal to one-third of the total size of the dataset. I.E. $\lim_{p \rightarrow \infty} b = \frac{k}{3}$, where b is number of broadcast pages, and k is the total number of pages. If we modify Algorithm 2 to broadcast in a tree pattern, we can achieve a CGM-compliant algorithm for the single broadcasting processor case in $O(\log p)$ rounds of communication.

For an optimal load-balanced (and broadcast-balanced) case, with each processor broadcasting on average $O(\frac{k}{p})$ of the pages, where k is the total number of pages, we can complete the data broadcast within one CGM round. This highlights the importance of load-balance to parallel algorithms.

In practice, with small p , and non-pathological data files, the page-pushing phase of the algorithm takes negligible time compared to the other parts of vector polygon overlay.

3.2.3 Parallel Line Intersection

A choice exists for intersecting geometric primitives here. One is to implement a full polygon intersection algorithm; the other is to intersect lines and polygonize the result later. The latter approach was chosen, chiefly for software architecture reasons, but also because line intersection is a widely understood, used, and implemented algorithm.

This is not to say that it is always a *well* understood, used, or implemented algorithm. Both the sequential line intersection algorithm and the parallel segment merge phase are difficult, topologically

and numerically. Dealing completely and correctly with these problems require rigour and precision in algorithm-building and implementation. These difficulties are detailed in Chapter 3.

The parallel portion of MFP-based line intersection occurs after all line intersections have been determined. The sequential line segment algorithm is performed on each MFP page, which contains all the line segments that intersect its boundaries, plus those segments contained fully within its boundaries. This generates the complete set of intersections within the page boundaries. None of these segments intersect one another, except at their endpoints.

Parallel Doppelgänger Segment Resolution

Once generated, the new set of line segments is processed to determine which ones intersect the page boundary. At this point, all the line segments inside the page boundary are correctly determined, and have been assigned unique IDs. We can ignore these from this point forward.

Those segments intersecting the page boundaries may or may not have the correct topology, and will have been assigned one unique ID in the current page. However, these segments, in various forms, will have duplicates on each page which they intersect. Such *doppelgänger*³ segments on other pages will also have been given different unique IDs, so that we cannot tell by looking at the IDs that these are really the same segments. Our task is to propagate the correct information about each line segment to each page which the segment intersects. This is made more difficult because no one page can know that it has all the correct information about a line segment which crosses a page boundary.

The first problem to be solved is that of the different unique IDs. Clearly, we cannot assign the unique ID of the parent line segment from an input set to its children, for there may be multiple children of one parent line segment. However, we do know that, for any pair of doppelgänger line segments, that they will share a parent line segment. Furthermore, any page can have at most two distinct doppelgänger line segments with the same parent segment. This suggests a relatively simple method of matching doppelgänger segment pairs across page boundaries, i. e. by keeping track of their parents' IDs separately. This requires a small modification to the sequential algorithm.

³Doppelgänger is German for “double”; the term is often used for a ghostly replication of a person or thing.

Event	Action	Transition
Own an Endpoint (at start)	Send segment data in the direction of the other endpoint.	Then wait to receive data about the other endpoint.
Own the Middle of a Segment (at start)	Wait to receive endpoint info from other processors/pages.	
Receive a first Endpoint	Update segment data and/or the segment and endpoint IDs.	Forward the endpoint data in the opposite direction, if necessary.
		Wait for the 2 nd endpoint.
Receive a second Endpoint	Update segment data, endpoint IDs. Mark this segment done.	Forward endpoint data in the opposite direction, if necessary.

Table 3.1: Parallel Processing of Segment Data

Propagation Algorithm The second problem to be solved is the propagation of the correct data to all pages owning a line segment. This data comes in two parts: the correct endpoints and node IDs of a segment, and the correct unique ID of a segment.

Each MFP page that owns an endpoint of a doppelgänger segment has the correct information about that endpoint. It is simple enough for each of those pages to send the correct endpoint information to the adjoining page that the line segment is shared with. These pages then wait for the correct information about the other endpoint to come to it.

Meanwhile, pages that own neither endpoint, i. e. the middle of the segment crosses the entire page, must wait dutifully for endpoint information to come to them, and propagate that information in the opposite direction from which it received it. A simple summary of this algorithm is shown in Table 3.1.

Each message containing endpoint data will also contain a unique segment ID, endpoint coordinates, and unique endpoint IDs for the endpoints of the doppelgängers. Because we are dealing with one-dimensional line segments, we can specify that a page adopts the data from the processor with the largest endpoint, based on a single parametric coordinate. This guarantees that the same unique segment IDs are used by all pages.

Figure 3.4 shows an example of the parallel doppelgänger segment resolution algorithm. In part (a), two segments, #5 and #3, are shown before the overlay operation. Part (b) shows the effect of the intersection on former segment #3: each page has assigned its copy of the segment a different unique ID. The page where the intersection took place assigned a different unique ID to each of its fragments.

Part (c) shows the result after one round of communication. The upper fragment of #3 is complete; the two pages holding copies have exchanged endpoint information and independently adopted the same unique segment ID. The lower fragment is not complete. The middle page of the three that hold copies knows all the information, and the correct unique segment ID, but the other two pages only know details of their own endpoint. At part (d), after the second round of communication, the page in the middle of segment 67 has sent details of opposing endpoints to the other two pages, and we are done.

3.2.4 Parallel Polygonization and Triangulation

Background

The first notable parallel triangulation algorithms were developed for the CREW PRAM. The first was an $O(\log^3 n)$ algorithm constructed by Aggarwal et al. [1]. Wang and Tsay in 1987 designed an $O(\log n)$ CREW PRAM algorithm to solve triangulation in parallel [84]. Given that the sequential lower bound is $\Omega(n \log n)$, this algorithm is optimal for the CREW PRAM. Their main observation was that funnel polygons could be used as a basis for triangulation on a set of points divided by strip decomposition.

Diallo et al. [19] extended Wang and Tsay's algorithm to other models of computation, primarily CGM. Their CGM algorithm computes a sequential triangulation locally on a set of points divided by strip decomposition. It then triangulates the space between local convex hulls using the funnel polygon algorithm described by Wang and Tsay.

Much work has been done with parallel triangulation on VLSI chips. When compared to CGM, the processor/primitive ratio is reversed for VLSI algorithms. In CGM, algorithms are designed for $p \ll n$. In the VLSI model, design is done for $p \gg n$. Typically, $p = O(n^2)$; n objects are solved on an $n \times n$ mesh. Jang et al. [43] solved the unconstrained triangulation problem on such a reconfigurable VLSI mesh in constant time, in 1997. Bokka et al. [13] improved their results in 1998, solving several kinds of constrained triangulations in $O(1)$ time on a reconfigurable $n \times n$ mesh. These constrained triangulations include triangulations of convex planar regions in the presence of rectangular holes, and triangulating in the presence of linear obstacles.

(a) Segments before overlay operation
(SID = "Segment ID")

(b) Segments after local overlay.

(c) Segments after one round of
communication.

(d) Segments after two rounds of
communication.

Figure 3.4: Four Stages of the Parallel Segment Resolution Algorithm.

See [2] for further reference on this topic.

Introduction

After the completion of the line segment resolution algorithm, some applications may be satisfied with a set of line segments. OVID's software architecture is constructed so that these applications can take the line segment-based solution and modify that (cf. Chapter 4). Other algorithms can continue on to polygonization and/or triangulation.

The most challenging part of the vector polygon overlay algorithm is the polygonization algorithm. Several issues contend for the algorithm designer's attention here: efficient sequential processing, tricky parallel merge cases, difficult worst case behaviours, and dealing with the structure of the MFP.

A logical place to start is the *chain-propagation* algorithm, which is a modification of the parallel line segment algorithm. In this context, a polygon can be described as a closed chain of line segments containing no other line segments.

One way to solve the polygonization problem is to trace, in parallel, the entire boundary of the polygon. Each page sends to both of its neighbours information about the partial chain of segments that it holds. On the second round of communication, these messages are propagated to pages two removed from their origin. This propagation continues until each page receives two messages that originate at the same page. At this point, each page along the polygon has received enough messages that it can independently reconstruct the entire polygon.

Knots, Stars, and Rings

The above algorithm has several severe problems. The first is that each page contains information about all polygons of which it owns even a single line segment. This problem is general; it forces any MFP-based polygonization algorithm to limit the number of polygons that a page can own.

The second is that there are large numbers of bad cases. The worst case is the "Knot" class of polygons, so named because they resemble knots in two dimensions. These polygons contains an arbitrary number

of segments, which thread their way through arbitrarily many pages an arbitrary number of times, preferably with each page containing individual chains of length at most two. One example of this kind of polygon is a maze. The chain-propagation algorithm runs in $O(n)$ rounds of communication on Knot polygons, with a high constant, which is unacceptably bad.

Knot polygons are admittedly rare. However, other classes of polygons can cause similar problems. Star-shaped polygons, for instance, can also force $O(n)$ behaviour from the algorithm. Concentric polygon rings can cause $O(\sqrt{n})$ behaviour, and pathological rings can cause $O(n)$ behaviour. Ring-shaped polygons are not so rare. An isoline map of a mountain provides exactly such a case, which cannot be ignored by the conscientious GIS programmer.

Better Algorithms

Another possible approach is *neighbour-polygonization*. In this approach, each MFP page first finds whole polygons and partial polygons (i.e., polygons shared with other pages) in that page. Then, each MFP page exchanges partial polygon data with the pages immediately surrounding it in its parent domain. This enables certain of the pages' partial polygons to be completed, and the others merged to form larger partial polygons. Or, to minimize the amount of repeated work, each local collection of pages can choose one page to perform its' calculations. (This approach is detailed later on in this section.)

A third approach is *neighbour-triangulation*. This approach is identical to *neighbour-polygonization*, except that a full triangulation of the MFP is done, rather than a simple polygonization⁴. Triangulation is not necessary in solving the polygonization.

A Good Average-Case MFP Algorithm for Polygonization

The following, Algorithm 3, is a good average-case MFP algorithm for parallel polygonization of the line segments generated by the overlay of two polygonal maps.

⁴Since the two tasks are closely aligned, and since the software architecture (cf. Chapter 4) is designed to let the two run separately, triangulation can be set up as a user-enabled option.

Algorithm 3 MFP Polygonization

1. **for** each MFP page, construct an internal polygonization of the page.
 - (a) Divide the polygons into three groups (cf. Figure 3.6):
 - i. *closed polygons*, which are completely enclosed by the page,
 - ii. *open polygons*, which are parts of larger polygons that span several pages, and
 - iii. *leaking polygons*, whose line segments are completely owned by the page, but overlap the boundary with one or more other pages.
 - (b) Mark the closed polygons as done; label them with a globally unique polygon ID (*GUPID*).
 2. **from** the maximum MFP level (denoted by l) **down to** 1
 - (a) **for** each *Parent Page Domain* (*PPD*) of level $l - 1$, **in parallel**
 - i. Calculate, for each page in the parent domain, for each page boundary separately, the Polygon Boundary Sequence (PBS).
 - ii. Every page in each parent domain sends to a single, pre-selected processor its polygon boundary sequences (PBSs) for each of its boundaries.
 - iii. The chosen processor merges the boundary sequences.
 - iv. The chosen page returns messages to all pages in a parent domain.
 - v. The “chosen page” for this parent domain now represents the parent page in the level $l - 2$ parent domain that will be processed later.
-

This algorithm is concerned with the creation of Polygon Boundary Sequences (PBSs). A PBS consists of an ordered list of all the open or leaking polygons that overlap with a neighbouring page. This list will also include line segments that overlap page boundaries.

Notes on Algorithm 3

Phase 2(a).i. ▷ When merging individual pages’ open polygons, phase 2.a.iii needs only to know that the open polygon exists, its globally unique polygon ID (*GUPID*), and which segments are its perimeter as it crosses the page boundary.

Phase 2(a).ii. ▷ Using only one processor for each parent domain is not a good solution for larger p , or if load-balancing problems occur. In these cases, intermediate localized processing steps can be used.

▷Figure 3.5 shows a sequence of intermediate steps that allows the efficient localized processing of intermediate PPD PBSs. This technique calculates PBSs for a local group of four pages, then merges those into a larger group of 16 pages, then 64, etc..

▷This works best with p being a power of 4, but can work with any size of p . The merging process in intermediate steps is identical to that described above.

Phase 2(a).iii. ▷ This results in a complete polygonization of the domain divided into sets of open, closed, and leaking polygons with respect to the domain, plus a list of open polygons that have been merged.

▷ This also produces the boundary sequences for the page’s parent domain.

Phase 2(a).iv. ▷ A message contains the final GUPIDs of their open polygons, plus the page ID of the pages that share the polygon fragments with them.

▷ Each page owning only part of a polygon (i.e. an open polygon w.r.t. the page) retains ownership of that part.

Phase 2(a).v. ▷ Note that, during the processing of the upper levels of the MFP, messages about new GUPIDs for merged polygons must be passed back down a chain of “chosen pages” for each parent domain, until every page has up-to-date information.

Analysis of the MFP Polygonization Algorithm

The simple version of this algorithm takes order $O(l)$ CGM rounds of communication. However, it would appear that we have violated one of the conditions for a CGM algorithm, that a processor communicates no more than $O(\frac{n}{p})$ data in a round of communication. (This condition bounds communication time for the entire algorithm.)

Each processor is represented exactly once in a PPD. Each processor sends $O(\frac{n}{p-a})$ of data to the chosen processor, where a is the total number of pages in the MFP file. Given that the chosen processor

Figure 3.5: Intermediate Processing Steps on a 64-Processor Mesh.

receives all of these p messages, it receives $O(\frac{n}{a})$ data. But, since $a > p$ in all cases, and given that we can select a different chosen processor for each PPD, we know that this algorithm is a CGM algorithm, and should be well-behaved.

It should also be observed that a convex hull or boundary structure rarely contains $O(k)$ data, where k is the size of the total input ($\frac{n}{p}$ here). In a uniform distribution, for example, only $O(\log k)$ data points sit on a convex hull [57]. For a uniform distribution, then, algorithms better than $O(l)$ will be possible in a CGM model.

The version of this algorithm with intermediate steps takes $O(l \cdot \sqrt[4]{p})$ rounds of communication, since the number of processors involved in the computation decreases by a factor of 4 in every round.

Figure 3.6: Open and Leaky Polygons on a Page Boundary.

A Good Average-Case Algorithm for Polygonization with Triangulation

The following (Algorithm 4) is a good average-case MFP algorithm for parallel polygonization *and triangulation* of the line segments generated by the overlay of two polygonal maps. This algorithm is similar to the polygonization algorithm; indeed, it contains all the same steps. I have included extra instructions necessary for triangulation. Please see Algorithm 4 for an explanation of the common steps.

Algorithm 4 MFP Polygonization with Triangulation

1. For each MFP page, construct an internal polygonization and triangulation of the page.
 - (a) Divide the polygons into three groups (cf. Figure 3.6):
 - i. *closed polygons*, which are completely enclosed by the page,
 - ii. *open polygons*, which are parts of larger polygons that span several pages, and
 - iii. *leaking polygons*, whose line segments are completely owned by the page, but overlap the boundary with one or more other pages.
 2. Mark the closed polygons as done; label them with a globally unique polygon ID (*GUPID*). Mark as done and label the triangles – distinct from the polygons, since we wish to keep both sets for different purposes – with globally unique triangle IDs (*GUTIDs*).
 3. From the maximum MFP level up to 1 (denoted by l), do the following:
 - (a) For each Parent Page Domain (*PPD*) of level $l - 1$, do the following, in parallel:
 - i. Calculate, for each page in the parent domain, for each page boundary separately, the Polygon Boundary Sequence (*PBS*).
 - ii. Calculate, for each page in the parent domain, the *Polygon Visibility Hull* (*PVH*) (see Figure 3.7).
 - iii. Every page in the PPD sends to a single, pre-selected processor its polygon boundary sequences for each of its boundaries, plus its *PVH*.
 - iv. The chosen processor merges the boundary sequences (*PBSs*).
 - v. The chosen processor calculates the polygon visibility hull (*PVH*) for the entire domain, and triangulates the area inside the *PVH* of the domain, but outside the *PVHs* of each individual page.
 - vi. The chosen page returns messages to all pages in a domain.
 - vii. The “chosen page” for this Parent Page Domain now represents the parent page in the level $l - 2$ PPD that will be processed later.
-

Notes on Algorithm 4.

Phase 3(a).ii ▷ To construct the *Polygon Visibility Hull* (PVH) (see Figure 3.7) calculate convex hull of the internal set of line segments, i.e. those segments that don't intersect the page boundary, and add the boundary-crossing segments. ▷ Modify this to replace convex hull segments that intersect boundary segments with a chain of segments that does not intersect it. That is, replace it with the convex hull of the polygon perimeter from either endpoint of the offending segment to the near endpoint of the boundary segment.

Phase 3(a).iii. ▷ Using only one processor for each parent domain is not a good solution for larger p , or if load-balancing problems occur. In these cases, intermediate localized processing steps can be used (see Figure 3.8).

Phase 3(a).iv. ▷ This results in a complete polygonization of the parent domain divided into sets of open, closed, and leaking polygons with respect to the domain, plus a list of open polygons that have been merged.

▷ This also produces the boundary sequences for that domain.

Phase 3(a).v. ▷ This message contains the final GUPIDs of their open polygons, plus the page ID of the pages that share the polygon fragments with them.

▷ Each page owning only part of a polygon (i.e. an open polygon w.r.t. the page) retains ownership of that part.

▷ Each page also receives the results of the triangulation from 3.(a).v, that pertain directly to it, including globally unique triangle IDs.

Phase 3(a).vi ▷ Note that, during the processing of the upper levels of the MFP, messages about new GUPIDS for merged polygons must be passed back down a chain of “chosen pages” for each domain, until every page has up-to-date information.

Figure 3.7: An Example Polygon Visibility Hull

▷ Each page on the perimeter of the current Parent Page Domain must also receive the results of triangulations that occur at higher levels of the MFP.

As mentioned in 2.a.iii.A, there may be a need for an intermediate set of processing steps to deal with larger p or load-imbalance. Figure 3.8 shows a sequence of intermediate steps that allows the efficient localized processing of intermediate PPD PVHs and triangulations. This technique calculates PVHs and triangulations for a local group of four pages. Once the triangulations are done on the group-of-four level, the entire PPD triangulation is complete, and results can be sent back to the other processors in the group-of-four.

A diagram is provided in Figure 3.8. The pointers indicate the migration of data from individual processors to the nodes that compute the local group-of-four data. Note that 25 of 36 processors are performing triangulation calculations. Only the right and bottom perimeter pages/processors are idle.

Figure 3.8: Inter-Page Messaging for a 36-Processor MFP Triangulation.

The merge step still must continue for the PBSs and PVHs into groups of 16 pages, then 64, etc.. This works best with p being a power of 4, but can work with any size of p . The merging process in intermediate steps is identical to that described above.

The analysis of the triangulation algorithm is similar to that of the polygonization algorithm. The number of extra communication rounds inserted for processing of the triangulation is constant.

Results of the Theoretical Approach

As a result of this chapter, the theoretical background and structure of the entire OVID overlay driver is complete. Design and implementation of the generic overlay driver, plus the entire vector polygon overlay driver, follows from this. The next chapter describes the portion of that work that I implemented.

Chapter 4

Practical Approach

This chapter examines the more practical issues surrounding the development of OVID. These include the software architecture, code structure, numerical issues, memory management, the use of LEDA libraries, and practical issues in the implementation of parallel algorithms.

4.1 Software architecture.

The goals of a good software architecture for OVID must first reflect the design specifications of that software. Here, this means that developing “useful, efficient parallel libraries of Computational Geometry and GIS software for use by non-specialist programmers and end-users” is the primary target of the architecture. Specifically, the architecture needs to satisfy the following requirements:

- ★ The architecture should support a **rich** yet **flexible** set of tools.
- ★ It should have a programming or user interface that is **simple** enough for a generic user of the GIS domain to understand and utilize.
- ★ It should exhibit **transparent parallelism**. That is, the architecture should allow the programmer to use the interface without concern for parallel computing issues.

- ★ It should have **good and useful default behaviours** with respect to numerical issues, topological correction, and the processing of attribute data. Changes to the system treatment of such issues should be an exception. At the same time, such changes should be easily available to the end-user.

The OVID software architecture must also take into account the nature of geometric computing, the overlay operation, and vector polygon overlay (VPO). Because overlay is such a powerful model of computation, the architecture needs to be flexible enough to handle multiple kinds of overlay. This means modularizing the algorithm-specific details – including things such as the contents of parallel messages – into contained, implementation-dependent modules. Finally, it must allow an efficient implementation of VPO.

4.1.1 A Three-Layer Architecture

In order to meet these requirements, a three-layer architecture was developed. This allows the flexibility needed to have several types of overlay exist in the same code space. The OVID architecture, as shown in Figure 4.1, contains these layers:

1. The **OVID Base**. This contains the basic algorithm used to complete OVID operations. The other parts of OVID are defined as function calls from the OVID Base. As well, this layer contains the functionality common to all OVID applications, or an interface to external functionality¹. This includes:
 - (a) File management
 - (b) Parallel communication primitives
 - (c) User interface and display functionality
 - (d) Memory allocation for the geometric primitives
2. The **OVID Algorithm Layer (OAL)**. This layer contains implementations of specific overlay algorithms. There are many kinds of overlays possible. Immediately apparent is the overlay

based on line segment intersection – a category that encompasses TIN or polygon overlays, line segment overlays, and contour line overlays. However, other overlay algorithms are just as valid; for example, one could build a weighted Voronoi diagram constructed from an input of points and polygons.

OAL accesses the functionality provided by the OVID Base. An algorithm can be executed stand-alone, or as part of an application.

3. **OVID Applications.** The applications will utilize one or more of the OVID algorithms to process their data. They may also use other drivers and/or algorithms, provided common data structure formats are maintained. The result of the processing can then be analyzed, displayed, or post-processed by the application. OVID applications access the functionality of both the base and algorithm layer.

This division is immediately useful. Generic code can be written for the use of MFP component software, and the portions of the algorithms that deal with manipulating the data belonging to a page. Manipulating the actual data occurs at the algorithm layer, where routines must be written that perform the actual intersection, and prepare and process parallel messages.

For the development of OVID's VPO component, two sublayers are created: one for the general handling of line segment data, and one for various kinds of intersection implementations. The sublayering enables underlying implementations to be changed without major changes to most of the VPO code. Finally, an application need do no more than set up its display mechanism, and call the vector polygon overlay algorithm with the appropriate file names.

Reasons for choosing this Architecture

This architecture is intended to provide a simple framework for applications writers to use, yet still allow Atlantis designers adequate flexibility to create efficient and re-usable algorithms.

¹Much base functionality is built on top of third-party libraries such as MFP, LAM-MPI, and LEDA.

Figure 4.1: OVID's Three-Layer Architecture

An architecture where the OVID Base and OAL are one unit necessarily creates a monolithic product that must anticipate every possible type of overlay. The three-layer architecture allows three properties for the algorithm and application designers to take advantage of:

1. **Efficiency.** Rather than code a monolithic program that must handle every conceivable case, algorithms can be small, tightly coded programs optimized for specific types of overlay.
2. **Flexibility.** Applications builders can custom-design their algorithms based on a composition of several algorithms. This allows for easy customization of software.
3. **Compartmentalization.** Rather than build one algorithm for use in all cases, software designers can build an algorithm in several independent parts. This promotes code re-use, and lets the smaller algorithms run more efficiently.

- ★ Algorithms composed of many small pieces will usually run less efficiently than if they were a single piece of software. This is particularly true if the same data must be repeatedly accessed across the various pieces.

4.1.2 Functional Division of Algorithms

In order to make the VPO algorithm as flexible as possible, it was divided into three parts:

1. Line intersection overlay,
2. Polygonization, and
3. Triangulation.

This allows applications writers to manipulate data in a customizable fashion. Someone who requires only line intersections can have this done without a time-consuming wait for polygonization and triangulation, or the cluttering of his data files with unnecessary polygons and triangles. Users who wish to polygonize a single data file of line segments can do that without running the time-consuming overlay algorithm first.

4.2 Building an OVID Application

Like NEMO, its predecessor, OVID allows for quick and easy implementation of applications. The problems inherent in parallel vector algorithms - parallel communication, dataset partition, numerical accuracy, and dataset propagation - are handled transparently by OVID. As with the NEMO CD, a few sequential functions must be written in order to allow a user to customize OVID.

Completion Status

Only certain parts of the OVID Vector Polygon Overlay driver are complete, as of this writing. These are the generic overlay driver framework, and the line×line overlay driver implemented on top of it.

Three other planned components have not been developed as a part of this research: the triangulation, polygonization, and attribute processing functionality. Of these three, the attribute processing functionality is required to make basic attribute-based GIS operations available for line intersections. Its implementation will be relatively straightforward, and take comparatively little time. Triangulation and Polygonization are required to complete a full vector polygon overlay operation; parallel MFP algorithms for these two operations are detailed in Section 3.2.

The following description illustrates the full power of OVID, given the existence of all of these components.

4.2.1 OVID Generic Overlay Driver Framework

The generic overlay driver framework, part of the OVID Base, consists of the OVID framework and its class structure (cf. Section 4.3). In particular, it is implemented by the Overlay class (Section 4.3.3). It implements the VPO algorithm described in Section 3.2.2. This framework:

- ★ coordinates **parallel messaging**,
- ★ handles **manipulations** of the **MFP** data structure,
- ★ deals with the **file system**, and
- ★ **integrates** it all into a seamless whole.

Algorithm writers do not have to concern themselves with any of this. The Overlay class passes each pair of MFP pages to an algorithm defined by the OAL-layer, such as the line×line overlay driver. The Overlay class also creates a new page for the algorithm's output to be stored in.

4.2.2 Line×Line Overlay Driver

The line×line overlay driver takes a pair of pages to be overlaid, and overlays their line segments. It stores the intersected set of line segments and their associated endpoints into the new MFP page.

As described in Sections 4.4 and 4.5, it deals with the algorithm-specific numerical, topological, and implementational aspects of line intersection. This includes the parallel resolution of doppelgänger line segments .

The line×line overlay driver is also split into two parts. The first deals with the setup of the overlay, splitting pages that exceed the threshold for numbers of line segments or points, adding line segments (*arcs*) and points to the new MFP page, and parallel resolution of doppelgängers. The second does the actual overlay, and the attribute matching as described in Section 4.5.1. This allows a different overlay implementation to be substituted if the programmer desires this.

User customization of the line×line overlay driver can be done by defining a subclass that overrides the default inclusion and attribute processing behaviours.

4.2.3 Supporting Operations

Triangulation and polygonization operations are both unary operations performed on an already-completed overlay of line segments. These operations, when complete, will be defined in an object-oriented framework similar to the line×line overlay operation, but as unary operations. Although the triangulation algorithm contains the functionality of the polygonization algorithm, the classes implementing these operations will be separate; they may share common functionality as a separate superclass.

4.2.4 Building OVID Applications

When OVID is fully implemented, building an application on top of it will be relatively easy. The application will consist of a single program containing calls to OVID class methods, plus methods containing application-specific algorithms for the processing of attributes, and the inclusion/exclusion rules for the relevant geometric primitives. This is best illustrated by an example application.


```

/* The following classes contain user-defined attributes */
App_Overlay subclass of Line2Line_Overlay {
    method line_seg_include;
    method process_line_seg_attributes;
}
App_Polygonize subclass of MFP_Polygonize {
    method polygon_include;
    method process_polygon_attributes;
    method splinter_polygon_removal;
}
App_Display subclass of Polygon_Display {
    method polygon_display;
}
/* Main Program */
void main() {
    App_Overlay *MyOverlay = new App_Overlay(``OutputFile``, ``Input-
File_1``, ``InputFile_2``);
    MyOverlay->File_Overlay();
    delete MyOverlay;
    App_Polygonize *MyPolygonize = new App_Polygonize(``OutputFile``);
    MyPolygonize->Polygonize();
    MyPolygonize->Remove_Splinter_Polygons();
    App_Display *MyDisplay = new App_Display(``chaos:0.0``, window, My-
Polygonize);
    MyDisplay->Display_Polygons_2D(window);
    delete MyDisplay;
    delete MyPolygonize;
}

```

Figure 4.2: Example OVID Vector Polygon Overlay Application

Example Application

Pseudocode for an example application is given in Figure 4.2. When Ovid's Vector Polygon Overlay component is complete, only a few sequential methods are required to complete the application. These are indicated in Figure 4.2, and override dummy methods defined in the parent classes. Certain methods such as *line_seg_include* will be trivial; others such as *process_polygon_attributes* or *splinter_polygon_removal* will require extra work as is demanded by the application in question. This will typically involve processing geometric primitives with respect to their attributes, which are known only to the application.

4.3 OVID Class Structure

The software architecture detailed in Section 4.1 has been implemented as a set of C++ objects, plus assorted support functions. The following is a high-level description of the *existing* class structure and code structure of the software. The OVID class structure is shown in Figure 4.3.

4.3.1 MFP Software

The basis of the entire Atlantis Vector GIS is Nussbaum's MFP file system (cf. Section 3.1). This software implements Nussbaum's MFP file structure. It handles file I/O, administration of the MFP pages, addition and removal of geometric data to and from MFP data structures, sorting the geometric data, and point location of points within those pages. It also handles converting data from standard TIN files to the MFP files. This includes the process of sorting geometric primitives and distributing them to individual processors. More information on the MFP software can be found in [61].

Lanthier and Nussbaum's parallel shortest path software was the first application built on top of the MFP software. This is described fully in [48] and [61].

4.3.2 OVID Foundation Classes

In order to build good object-oriented software, an object-oriented interface is required. The foundation classes are an object-oriented front end to Nussbaum's MFP interface, which is written in structured C. In addition to controlling access to the MFP implementation, the OVID foundation classes contain extra data and functionality required for OVID. In several places, they integrate and coordinate operations that otherwise require several calls to MFP routines to be repeated in user code.

The foundation classes enable a straightforward inheritance to define the rest of the classes. Of the several algorithm types, all can be implemented as subclasses of one of the three foundation classes: the MFP file, the MFP page, or the MFP page index.

This also presents a unified schema to the user, and one that is relatively simple to grasp. The concepts

of file and page present sufficient information to an end-user that they should be able to write any algorithms required of them at that conceptual level, and leave the rest to the OVID and MFP code.

The MFP File Class

The MFP file class, **MFPfile**, handles interactions with the MFP file system. This includes:

- ★ opening, closing, and creating files,
- ★ adding MFP pages to those files,
- ★ allocating and managing data structures for the MFP files, and
- ★ destroying and deallocating the data structures and files.

Through **MFPfile**, the algorithm implementor manages the MFP pages that exist in the processor's memory. This gives the programmer two ways to manage memory: via virtual memory, and through manual loading and writing pages to and from disk. In any algorithm that is a binary operation $A \times B = C$, each of A , B , and C are instantiated as separate MFP files; the binary operation is a subclass of **MFPfile** encompassing three **MFPfile** objects.

The MFP file class also handles the transport of pages between processors. This is necessary in an overlay where the same geographic area of the two source files is owned by different processors (cf. Section 3.2.2).

The MFP Page Class

The MFP page class, **MFPpage**, deals with tasks assigned to individual pages. These tasks include:

- ★ the addition of and deletion of geometric objects from the pages,
- ★ memory management for the geometric objects, and

★ the splitting of pages.

The latter is an especially important task for the VPO software, as the overlay of two large TINs can produce many times more line segments than went into the overlay. Splitting the pages allows the processor to keep pages to a reasonable size.

The MFP page class also contains methods that determine which page a line segment intersects, in order that a message can be sent to the page that owns the copy (*doppelgänger*) of the line segment.

The MFP Page Index Class

The MFP page index class, **MFPndx**, operates on the level of a MFP *page index*. The page index is the part of an MFP file that keeps track of what pages reside in the file, what level of the file they live on, and their geographical location with respect to the rest of the pages. It is designed to enable fast point location searches. The MFP page index class is essentially a front end to the MFP page index class.

4.3.3 OVID Base Classes

The OVID base classes form the basis for the overlay driver. These classes have two functions. First, they perform the work at the base level of the overlay driver that is common to all kinds of overlay. This has to do with manipulations of the MFP files and pages, interprocessor messaging at the level of a generic message, and overall direction of the overlay operation. Secondly, they provide a base on top of which programmers can build the rest of the OVID infrastructure.

Overlay Class

The Overlay class is the heart of OVID. The Overlay class defines in implementation the generic overlay described in sections 2.4.1 and 3.2. The methods it declares are the only methods necessary to define any overlay operation.

The overlay class contains three files, corresponding to the binary operation $A \times B = C$, and the pages belonging to those files. It contains indexing mechanisms to track the progression of the overlay operation through the pages. It also contains the algorithm that determines which pages need to be overlaid, and their overlay partner pages (cf. Section 3.2.2 on page 40). It pushes the required overlay pages to other processors for this purpose.

The Overlay class contains a generic overlay algorithm that calls the operation-specific algorithm once all the generic processing of pages is complete; the specific algorithm is built using over-ridable methods provided by Overlay class. It then performs a general CGM h-relation, passing messages back and forth as specified by the methods overridden by the specific class. Overridable methods include `PageFetchExtraData()`, `PageOverlay()`, `PropagatePrepare()`, and `PropagateReceive()`. These all call the specific-algorithm overriding methods with a specific page or set of pages to operate with.

An example of a specific algorithm class is the `Line2LineOvly` class in Section 4.3.4.

Geom Class and Family

Geom Class is a superclass for buffers of geometric primitives that are handled by the OVID. Objects of this class can be passed to specific-algorithm subclasses without passing the entire MFP page buffer.

Some subclasses of Geom class include `Arcs2D` class, which is utilized in the `Line2LineOvly` class.

OvPropMsg Class

OvPropMsg Class is the virtual message class that Overlay class uses to manage messages. Arrays of OvPropMsg are created by calls to the appropriate subclass' message creation method. The values are filled in by calls to the `PreparePropagate()` method (an overridden method of Overlay class), and sent to/received from other processors by the send and receive overridden methods of Overlay class.

All memory management, value filling, and information processing on OvPropMsg class is handled by subclasses of itself (e.g. `L2Lmsg` class) and of Overlay class (e.g. `Line2LineOvly` class).

4.3.4 OVID Algorithm-Specific Classes

Line2LineOvly Class

Line2LineOvly is a specific-algorithm class for Overlay Class. It implements a line-to-line overlay, i.e. a line-segment intersection algorithm. It sets up the per-page overlay for the two pages it has been passed, and invokes an implementation of a line intersection algorithm. The implementation of line intersection is hidden from this class so that various implementations of line intersection can be used without the knowledge of this class.

This class implements messaging specific to parallel line intersection. Specifically, it implements the parallel portion of the line intersection algorithm, as described in Section 3.2.3.

Arcs2dOvly Class

Arcs2dOvly class is a subclass of Arcs2D class, which is a subclass of Geom.

Arcs2dOvly is the class that implements specific line segment intersections algorithms. Different subclasses of Arcs2dOvly class can be used to access various intersection algorithms using the same user interface.

L2Lmsg Class

L2Lmsg class implements message handling issues, in conjunction with Line2LineOvly class. It contains memory management, initialization, and accounting functions for messages used in the parallel portion of the line intersection algorithm, as described in Section 3.2.3.

4.4 Numerical and Topological Issues

Line- and/or polygon- intersection algorithms are, when implemented, notorious for numerical and topological problems. Topological problems most often arise as multiple special cases that must be

Figure 4.3: Class Diagram for OVID.

considered. These can be due to degeneracies in the data, or to inherent difficulties in the problem at hand. Numerical problems can not only cause inaccuracies in the coordinate values of geometric computation, but can cause myriad topological problems as well. Even a seemingly simple algorithm such as point location can be affected.

4.4.1 Background and General Approaches

Numerical problems have long been the bane of computational geometers. As a result, there are a variety of techniques designed to combat different problems.

A general discussion of floating-point arithmetic is given by Goldberg in [32]. This is an excellent introduction to floating-point arithmetic. It gives basic explanations of floating-point standards, rounding errors, summation and multiplication errors, and some work with error bounds.

Hoffmann [39] divides geographical data into two types: numerical and symbolic or topological. The challenge posed by Hoffmann is to handle both types of data in a consistent, accurate manner. He outlines several techniques:

- ★ Floating-point arithmetic is useful but problematic; even if we can prove error bounds, many cases arise that cause topological errors within these error bounds.
- ★ Limited precision exact rational arithmetic (proposed by Sugihara [83]) can give topologically exact results; however, the precision needed can reach eighty times the input precision. More promising is the variable-precision approach [30] [50] [51]. Normal precision is used unless an analysis of error bounds in a calculation demonstrate the need for higher accuracy. The precision of a calculation is refined until its result - numerical or topological - is guaranteed to be correct.
- ★ Purely symbolic representations can be useful, but the best known topological reasoning algorithms have exponential complexity. In [40] Hoffmann et al. propose a numeric algorithm that uses symbolic topological reasoning to resolve topological problems within the numeric data where accuracy is in question. This appears to be a promising approach.
- ★ Perturbation approaches involve subtle alterations to numeric or symbolic data, in order to make the resulting computations more robust.
 - One example of this is Milenkovic's edge-cracking and vertex-shifting techniques [53]. These techniques guarantee that no geometric primitives have a non-intersecting approach of less than distance ϵ . Milenkovic does this by:
 - shifting vertices - essentially merging them - when they are too close.
 - "cracking" edges, so that the edge actually touches the vertex.

Worst cases cause problems here too; edge-cracking and vertex-shifting when combined can only guarantee a maximum of $O(n\epsilon p)$ positional perturbation with $O(\epsilon)$ tolerance.

- A more mathematically complex method of perturbation was proposed by Edelsbrunner and Mücke [24]. “Simulation of Simplicity” involves perturbing all coordinates of a dataset by a polynomial in ϵ , such that all transformed geometric primitives are at least ϵ distance away from their neighbours.

The most promising techniques for vector polygon overlay appear to be variable-precision exact arithmetic, and numeric calculations assisted by topological reasoning. All of these techniques have some application, depending on the problem being solved.

4.4.2 Issues in Line Segment Intersection

There are many failure cases that need to be dealt with if we are to successfully implement a line intersection algorithm. The following five (cf. Figure 4.4) are common examples. A more comprehensive treatment is given in [20] and [79]. Note that the shaded squares in parts (a) to (d) of Figure 4.4 represent the resolution size of a floating-point representation.

Collinearity In the world of real numbers, a point will never fall on a line, unless it exactly matches the equation of the line. This can be emulated by exact arithmetic.

In the world of floating-point numbers, the point can easily be rounded to sit exactly on the line. The safest floating-point option, especially in the case of parallel boundaries, is to assume that no intersection occurs in the case of collinearity (Figure 4.4(a)).

False Intersection A second problem caused by rounding error in floating-point calculations is false intersection. An intersection that does not exist, but is formed by two nearly touching chains of a line segment, may be rounded in such a way that, in floating point representation, they do intersect (Figure 4.4(b)).

Figure 4.4: Five Common Examples of Floating Point Failures in Line Segment Intersection.

Denied Intersection The third problem caused is similar to false intersection. Here, two line segments may actually cross, but in the floating point representation, they are rounded so that they do not cross (Figure 4.4(c)).

Short Line Segment We may encounter a line segment that is “too short”; i.e., within the floating point representation on a machine, the endpoints of the line segment are so close together that they cannot be used to calculate accurately the equation of a line [20]. The calculated slope, for instance, will not be accurate (Figure 4.4(d)).

Braided Lines When two lines with nearly identical slopes approach one another, strange things happen with floating-point representations. Because the floating-point representation is discrete, the line is represented at the finest level of detail as a series of vertical and horizontal steps. It is quite easy to come up with a case where the vertical and horizontal zig-zags from two lines intersect repeatedly, forming a “braiding” effect [51]. This behaviour can cause the complete failure of geometric algorithms (Figure 4.4(e)).

Numerical Solutions in Line Segment Intersection

One method of solving numerical problems in line segment intersection was proposed by Fortune and Milenkovic [30]. The proposed solution involved the use of approximate arithmetic, together with perturbed representations of lines named *pseudolines*. The pseudolines are perturbed so that every intersection is precisely defined; this means allowing the lines to “bend” near intersection points, with the proviso that the pseudolines remain monotonic. This approach produced good results; however, it is vulnerable to the short line segment problem.

Another approach is to use Milenkovic’s edge-cracking [53]. This approach preserves the integrity of floating-point calculations by preserving a distance of ϵ between two segments. However, this approach changes the topology of the planar partition during the intersection. In a general-purpose polygon overlay algorithm, this is unwise; many algorithms implemented on top of a general-purpose engine may require the preservation of topology.

A third approach is to use exact, or sufficiently exact arithmetic. In this context, “sufficiently exact” means exact enough to preserve topology at all points in an intersection. This approach is taken by Melhorn et al. [51], who developed the LEDA library. The advantage of this is that it calculates a topologically correct solution. The disadvantage is that the sufficiently-exact solution takes longer to complete. Also, bad cases can cause unpredictably slow computation times.

Topological Cases in Line Segment Intersection

Topological issues tend to be problem-specific. Line segment intersection is a good example. There are ten different ways that one line can be related to another. This treatment goes further than simply examining topological cases of line segment intersection, even if we ignore numerical problems. Very often, an algorithm must account for and test for conditions that do not involve intersection; one such algorithm is the parallel segment resolution algorithm detailed in section. Differentiation is often useful between parallel and non-parallel cases.

The ten cases are [86] [20] (cf. Figure 4.5):

1. The line segments are not parallel, and do not intersect.
2. The line segments are parallel, and do not intersect.
3. The line segments intersect “normally”, i.e. the intersection takes place in the interior of both line segments.
4. The segments are parallel and intersect on a mutually shared endpoint.
5. The segments are not parallel and intersect on a mutually shared endpoint.
6. The segments intersect in one point, which is the endpoint of only one of the segments (a non-parallel case).
7. The line segments are identical.

Figure 4.5: Ten cases of Line Segment Intersection

8. The line segments are collinear but distinct. I.e. they possess the same equation but do not overlap.
9. One line segment completely contains the other (a parallel case).
10. The line segments are collinear and overlap.

4.4.3 Issues in Polygonization

Splinter Polygons

In any vector GIS system, a *splinter polygon* policy must be defined. Splinter polygons are very narrow or degenerate polygons that result from the overlay of finely detailed maps, or because of numerical

inaccuracies. The definition of “very narrow” for our purposes can be those that cause problems with the software or analysis in question. They are a problem because they clutter the output of an overlay operation with insignificant polygons; also, they can cause further numerical inaccuracies in subsequent calculations because of their narrowness. Depending on the analysis technique, they could cause huge errors in the user’s calculations; one example of this is a perimeter calculation.

On detection of splinter polygons, a number of techniques can be applied:

- ★ The polygon may be added to an adjacent polygon with which it shares some defining characteristic, such as type. A more subtle strategy can be used if no adjacent polygon shares a defining characteristic. In that case, one may add the polygon to a suitable adjacent polygon, and adjust its size downward, by giving more area to a polygon with the required characteristic.
- ★ The size of the polygon can be increased, as is done by Milenkovic [53].
- ★ The polygon may be added to another polygon with which it shares the longest perimeter for the reason that it reduces the number of polygons produced, and eliminates potential future problems with numerical inaccuracies.
- ★ The polygon may be split into pieces; the pieces can then be added to adjoining polygons. This eliminates cases of a “normal” (i.e. non-splinter) polygon with a splinter fixed to its edge.

Removing splinter polygons that have meaning within a map may cause the loss of significant information. The need for a policy alternative will need to be evaluated on an application-by-application basis. In many cases, application-specific alterations may be needed to preserve information desired by a specific algorithm or application.

4.5 Implementation Issues

The following issues arose during implementation. These issues, primarily in the areas of data management and parallel propagation, required in some cases special preprocessing or algorithm design in order to make the Line×Line overlay implementation work.

4.5.1 Attribute Data: Loss and Recovery

An important part of GIS is the integration of geometric data and its associated database attributes.

While using LEDA's intersection algorithms is much preferable to coding all the special cases and numerically sensitive code one's self, LEDA does not handle attribute data. In the case of line segment intersections, the minimum attribute data that needs to be preserved is the identity of the parent (or parents) of a line segment. I.e., if a line segment is split, one must know which line segment(s) from the input file(s) are associated with it. This allows us to recover attribute data such as adjacent polygon faces, and the related data attributes (elevation data, soil type, etc.).

What is the best way to keep the attribute data? There are template-based classes with which one can implement attribute-inclusive versions of LEDA code, but this is a lot of extra effort for the gain.

In the case of line intersection algorithms, the following approach was taken:

1. The line segments from the MFP files were input into LEDA data structures.
2. At the same time, these original line segments were input into a three-level search structure corresponding to its input file (described below and in Figure 4.6).
3. After the LEDA line segment algorithm completed, the line segments were retrieved from the graph data structure provided.
 - (a) For each line segment output, a search was done for its parent segment (if any) in each file.
4. The IDs of the parent segments were stored in the new MFP arc structure created for the new arcs.

Line Segment Search Structure

The three-level search structure (Figure 4.6) involves sorting data by three different criteria.

Figure 4.6: Three-Level Segment Attribute Recovery Structure

1. First, the data are inserted into a dictionary structure, sorted by slope. Because LEDA rationals are being used, sorting by slope is exact and not prone to problematic numerical errors.
 - ★ Now, it is not the *data* per se that is inserted into the slope-sorted dictionary structure. Since we desire to sort according to three different criteria, it is actually a second sort structure that is inserted into the dictionary.
2. Once the data is sorted according to slope, we can picture each dataset as a collection of parallel line segments; the sort is not unlike a polarization of light.
 - ★ The second sort done is to sort each set of same-slope segments by their intercept on the Y-axis. A sorted sequence was selected to slot each segment into a set of collinear segments with the same slope.
3. Since we desire a three-layer structure, a separate sorted sequence is inserted. This sequence stores the selected data, sorted in increasing order of x- and y- coordinates. Since every segment stored is on the same line, x- and y- coordinates increase linearly together.

By using the above structure, any line segment in the new, intersected set can be matched to its parent segment in $O(\log s + \log i + \log c)$ time, where s is the number of unique slope values generated by the set of segments, i is the maximum number of unique y-intercept values present within any of the unique slope value segment sets, and, and c is the maximum number of line segments within any of the unique-y-intercept search structures.

The actual complexity of this search is dependent on the distribution of the line segments. A TIN derived from a DEM model with most slope values concentrated around ratios of small integers (i.e. absolute slope values $\in \{3, 2, 1, 0, \frac{1}{2}, \frac{1}{3}, \infty\}$, with ∞ representing the slope of vertical lines), will usually have s of constant order, with $i = O(\sqrt{n})$. A randomly generated set of segments will have $s = O(n)$, with $i = O(\sqrt{n})$.

Suffice it to say that the search time will not exceed $O(\log n)$ in any case.

4.5.2 Determining Neighbouring Processors

In order to communicate effectively in parallel spatial data processing algorithm, a processor must know which other processor owns the data next to its data. In the MFP scheme, this is not as trivial as it seems at first. The reason for this is that the page next to the currently considered page may be subdivided arbitrarily. The subdivisions of a neighbouring page structure are not reflected in other localities within a page structure.

Unique Page IDs

In order to better determine a page's position within an MFP file, I implemented a unique page ID scheme within the MFP code. Given a page currently existing within an MFP file, a page can be uniquely determined within the file by a combination of *level* and *page ID*. The level refers to the level of subdivision that the page is at; the page ID refers to the position within an MFP map at that level that the page occupies. This differs from Nussbaum's ID scheme [61], which assigns an ID unique only within the Parent Page Domain of the current page. The scheme used for the page ID within each level is simply that page's number within a row-major order.

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Figure 4.7: Unique IDs in a 2-Level 2x2 Processor MFP.

Given a two-dimensional MFP with $p_x \cdot p_y$ processors, subdivided at level k , there are $(p_x \cdot p_y)^k$ pages at that level. Given an MFP with an arbitrary subdivision with a maximum level l , there are $\sum_{i=1}^l (p_x \cdot p_y)^i$ possible unique combinations of level and page ID. Refer to the parent-child processor ownership map is described in Section 3.1.

An example of a 2-level decomposition of a 2x2 processor MFP is given in Figure 4.7. Note that this is also valid for a 1-level decomposition of a 4x4 processor MFP.

Finding Neighbours' Levels, Page IDs, and Owners

Given an arbitrary page within an MFP file, the easiest way to find its neighbours is to pre-build a neighbour map. This is done with Algorithm 5.

Once a neighbour sequence is pre-built, it is used in propagation algorithms.

Algorithm 5 Determination of the Global MFP Page Map

1. **Each processor** broadcasts the page level and ID of every page that it owns to every other processor.
 2. **Each processor** can build a map of the entire MFP with the information it receives from the other processors. This can be done using Nussbaum's page index structure [61].
 3. **Each processor** can build a neighbour list for each of its pages by doing point-locations along its border with the adjacent pages.
 - (a) Because a page can have an arbitrary number of neighbour pages along each boundary, pre-compiling a neighbour list necessarily means determining the number and owner of each page along the boundary.
 - (b) For each of the four edges or boundaries along a page, perform the following (see Figure 4.8):
 - i. Do p point locations, using points ϵ across the boundary, and at the relative positions $\{\frac{1}{2 \cdot p}, \dots, \frac{2 \cdot i + 1}{2 \cdot p}, \dots, \frac{p-1}{2 \cdot p}\}$ along the boundary, where p is the number of processors in the dimension parallel to the boundary. (For $p = 2$, this would be the $\frac{1}{4}$ and $\frac{3}{4}$ positions, respectively. For $p = 3$, the $\frac{1}{6}$, $\frac{3}{6}$, and $\frac{5}{6}$ positions, etc..)
 - ii. If the page returned from the point location is the same for both points, then only one neighbour exists.
 - iii. If the pages returned are different, then further point locations are done. When every neighbouring page is found, then the list of these pages is returned.
-

Notes on Algorithm 5.

Phase 3(b).i $\triangleright \epsilon$ is defined as the world size in the dimension perpendicular to the boundary, divided by $2 \cdot q^m$, where q is the number of processors in the perpendicular dimension, and m is the maximum level of the MFP file.

This is illustrated in part (a) of Figure 4.8.

Phase 3(b).iii $\triangleright$ Point locations in the manner of 3.b.i are done recursively until, at the lowest depth of the recursion, two point locations return identical page identifiers. This ensures that every neighbouring page is found.

$\triangleright$ The list of neighbour pages is returned sorted by the recursion, and is kept in a sorted sequence sorted by boundary coordinate for each of the page's four boundaries.

Figure 4.8: Neighbour-Finding in an MFP.

► This recursion is illustrated in step (b) of Figure 4.8, where the fractional value in the y-coordinate is always in proportion to the size of the boundary in question.

4.5.3 Issues in Line Segment Propagation

Recall the description of the parallel line segment propagation algorithm given in Section 3.2.3. The following is a discussion of some issues in implementation of line segment propagation.

Segments Intersecting Page Corners

One operation that is done during the propagation of a line segment's data between pages is the calculation of the owner of the segment's doppelgänger on a neighbouring page. The calculation cannot

be based on the position of the segment's endpoints; rather, it must be based on the intercept of the segment with the page boundary.

Once the intercept is calculated, it is used to locate the owner's page and processor ID in the neighbour sequence. An appropriate message is then sent to the owning page. Since a message is sent once for each time a segment crosses a page boundary in a given direction, each segment requires $2k$ intercept calculations, where k is the number of page boundaries the segment crosses.

There is one exception to the above statement. When a segment crosses a page boundary *exactly* at the corner of a page, two messages must be sent in each direction. Intuitively, a crossing exactly at a corner implies that a single message be sent in each direction, to a *diagonal* neighbour. The MFP does not do this. Because a segment is considered to be owned by a page when it touches any part of that page, doppelgängers of the segment in question are owned by each of the pages that meet in the corner.

See Figure 4.9 for a visual description of the messaging in this case. Part (a) shows a segment crossing exactly on a corner. Part (b) shows the MFP ownership model for this situation: all four pages own a copy of the segment. Part (c) shows the messaging pattern used to update all owners of the segment with updated data from other processors.

The easiest way to resolve this “four doppelgängers problem” (Figure 4.9) is for the two processors that own the endpoints to send to their vertical and horizontal neighbours that border on the corners. These processors correct their own segment data, and pass the messages on to the neighbour that is diagonal to the original sender. A segment that crosses page boundaries only at corners will therefore require $4k$ intercept calculations.

Segments Intersecting Page Corners, Collinear with Page Boundaries

This is a special case of the Segments Intersecting Page Corners case, above. In this case, the segments intersecting the page corners are also collinear with a page boundary (Figure 4.10).

This is a difficult case, because the segment does not penetrate the interior of any page, but rather shares the boundary of two pages on each side of the page corner. Because this segment may cut other

Figure 4.9: Special Messaging with Corner-Intersecting Segments

Figure 4.10: Segment Intersecting Page Corners, Collinear with Page Boundaries

segments into pieces, it must necessarily be owned by all four pages on which it coincides. Functions that select segments for resolution across boundaries normally key on having one or more endpoints on the interior of the page. Most segments with one endpoint lying on a page boundary and one outside it, are completely owned by an external page, and require no action from this page. In the corner-collinear-boundary (CCB) case, this is not true, so special tests for boundary collinearity must be done by messaging functions.

The second major problem here is that, unlike the normal segment-intersecting-page-corners case, a page that owns part of a CCB segment cannot send double messages as above. Rather, it must only send a message (or messages) to the page (or pages) on the near side of the segment in question. This is in concept simpler, except that it is an exception to an exception case, which can make implementation more difficult.

Resolving Doppelgänger Across Page Boundaries: Details

When performing the doppelgänger resolution across page boundaries (cf. Section 3.2.3 on page 43), the purpose is to merge segments that are actually the same segment, but have been labeled differently during segment intersection on the various processors.

There are two phases to the doppelgänger resolution.

1. The **SEND** phase, consists of the processor collecting data about its unresolved segments and sending it to the adjacent pages. Data is only sent if new information about that segment's endpoint data is available to the current page. This is relatively simple.
2. The **RECEIVE** phase collects messages sent to each page, and applies the data in the messages to the corresponding doppelgänger in that page. It then determines whether the new information about that doppelgänger needs to be sent to the page in the opposite direction from which the message was received.

In the receive phase, the initial matching of segments is done by the ID of the parent segment that was used in the intersection. This allows a processor, upon receipt of a message containing segment data, to

determine which of the segments in the receiving page could be the other end of the segment described in the message. (The intersected segments are matched with the ID(s) of their parent segment(s) by the algorithm in Section 4.5.1.)

There may be many segments in a page that are descendants of a single parent segment. Thus, the receiving processor must determine which of these segments is the doppelgänger of the segment in the message. Upon finding the correct segment, it must further adjust the segment that it owns, to ensure that that segment resolves to the same ID, and the correct endpoint co-ordinates, on all pages that share the segment in question.

The following, Algorithm 6, describes this process. Recall that matching segments by definition are collinear (although not necessarily coincident).

Notes on Algorithm 6

1(a).ii ▷ This can happen if the message segment has an incorrect endpoint.

▷ Incorrect endpoints can arise if the messaging page did not contain the line segment that cut this segment in the receiving page.

4.5.4 Issues in Triangle/Polygon Propagation

Recall the parallel MFP polygonization and triangulation algorithms in Section 3.2.4.

MFP Polygon Boundary Propagation

The part of the parallel pure polygonization problem pertinent to propagation is the calculation of the Polygon Boundary Sequence (PBS). This calculation is quite simple. Given that a local sequential polygonization of the page has been completed, we have lists of closed, and of open/leaking polygons.

The open and/or leaking polygons can be sorted with respect to which boundary they cross with a simple segment-boundary intercept test (cf. 4.5.3). Then, they can be sorted in order of where they intercept

Algorithm 6 Resolving the Doppelgänger Segment in a Received Message

For each page segment that shares the parent ID with the message segment, **do**:

1. Calculate the number of endpoints the two segments share. This will be 0, 1, or 2.
 - (a) If the segments share neither endpoint, then check the following three statements. If these are true, then we have found the matching doppelgänger. Resolve the ID and endpoint values (step 2).
 - i. The two segments overlap.
 - ii. One segment is not completely contained within the other.
 - iii. The message segment did not come from the direction of an endpoint of the receiving segment, that is located in the receiving page.
 - (b) If the segments share 1 endpoint, then we know that that endpoint is either correct, or will be updated by another message at some time in the future. Check the following statement. If this are true, then we have found the matching doppelgänger. Resolve the ID and endpoint values (step 2):
 - i. The segments' endpoints opposite the shared endpoint are in the same direction as each other. (This must be the direction from which the message was received.)
 - (c) If the segments share both endpoints, then we know that either one or both endpoints are correct, and at most one will be updated by another message at some time in the future. We have found the matching doppelgänger. Resolve the ID and endpoint values (step 2).
 2. Resolve the ID and endpoint values, if the doppelgänger was found.
 - (a) The ID is a choice between the message arc ID and the receiving arc ID. This can be chosen arbitrarily by some function that fully orders geographical points, such as x- and y- coordinates.
 - (b) The endpoint values are resolved by choosing the pair $\{x, y\}$ from $x \in \{l_m, l_r\}$ and $y \in \{r_m, r_r\}$, where the distance between x and y is the minimum for all x and y . l_m and r_m are the left and right endpoints of the message arc; l_r and r_r are the left and right endpoints of the receiving arc. The existence of a shorter segment implies that at least one of the pages contains an arc that intersects the longer segment.
-

the boundary. The ordered list of segments that cross the boundary, plus the associated polygon IDs, are then shipped off to a single processor. This processor aligns the pairs of PBSs from the various processors on its internal page edges, and resolves the multiple polygon IDs into a single, unified set. Given that exact arithmetic is being used for these calculations, no topological inconsistencies will occur between each pair of PBSs.

The chosen processor also generates a new set of PBSs to propagate, recursively, up the MFP to the PPD at the next level up.

MFP Triangulation Propagation

The parallel triangulation algorithm is more operationally complex than polygonization. The propagation issues are easily solved by applying the polygon boundary propagation algorithm. The only major difference here is that, instead of sending a sequence of polygon IDs along the boundaries, the actual open and/or leaking polygons must be sent to the appropriate processor. The triangulating processor must know the location of every line segment in the Polygon Visibility Hull.

The parallel triangulation is accomplished in each Parent Page Domain by sending the data surrounding each page corner to one processor owning a piece of said corner. Because of this, a standard method must be put in place for determining where the area allocated to one corner ends, and the data allocated to the next corner starts. This way, every processor knows independently what area it needs to triangulate, rather than generating multiple extra triangles. The simplest way would be to choose the boundary-crossing line segment nearest to the center of each page boundary.

4.6 Implementation Aids

I used the following two software libraries in my development of the OVID framework and the line segment overlay driver.

4.6.1 MPI

MPI is the *Message Passing Interface*, an international standard developed for communication on parallel machines, primarily distributed architecture machines [52]. Its goal is “to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.” MPI consists of interface and implementation specifications for a general message-passing standard, covering point-to-point and collective communication, process topologies, and environment management. Its strengths are that it is the widely accepted standard, its specification of consistent interface and behaviour, transparent handling of communication failure, and its rich functionality. Its weaknesses are that efficiency and quality suffer between computing platforms and implementations. This is often the case.

I chose the freeware LAM version of MPI for my implementation, developed by Ohio State University, and now maintained by Notre Dame University.

4.6.2 LEDA

LEDA is “A Library of Efficient Data Types and Algorithms”, developed by the *Max-Planck-Institut für Informatik* in Saarbrücken, Germany. It provides a wide variety of data types, data structures, and algorithms designed for efficient and accurate processing of data. This is particularly useful in geometric computing.

LEDA is implemented as a series of template-based C++ classes. Data types and algorithms use the templates to create multipurpose tools for programming. Data structures such as lists, trees, queues, etc., can be built out of any data structure one might wish.

Sufficiently Exact Arithmetic

Floating-point numbers are sufficient for many algorithms. As discussed earlier, the use of the floating point representation is a serious problem for many kinds of computing, including geometric computing. LEDA offers two data types for exact computation:

- ★ LEDA reals. This type uses sufficiently exact²arithmetic. The internal code for computing the values of LEDA-real expressions analyzes expressions for the exactness required. A maximum bound on the error of the expression is determined; if a value has a calculated error of greater than the maximum error bound, the expression is recomputed with greater accuracy until the expression is guaranteed to be correct to within the maximum error bound [51].
 - One situation that LEDA reals are frequently used in is a geometric sign-test. When the value of a sign test is close to zero, floating-point numbers will often give incorrect results. Such incorrect results cause algorithms to make wrong decisions, or cause incorrect topologies to be calculated. Using LEDA reals is an elegant solution to this problem, that does not force the user to pre-process the geometric data, yet returns a provably correct result.
- ★ LEDA rationals. These fit the mathematical definition of rational numbers; they are formed by the quotient of two integers. LEDA rationals are also a kind of exact representation. They are smaller and take less time to compute with, yet they still provide geometric algorithms with guaranteed results [51].

Efficient Data Types

Several LEDA data structures were used for the implementation of OVID. These are [50]:

- ★ Doubly-linked linear lists. Items are kept in the order inserted.
- ★ Priority queues. Items are kept in a sorted order, based on a linearly ordered key, with access coming at the head of the queue. The underlying implementation is a linear list.
- ★ Dictionaries. These keep data sorted linearly, with fast random access. They are implemented by (2,4)-trees, and thus yield logarithmic-time search, insert, and delete operations.

²*Sufficiently exact* is my terminology.

- ★ Sorted sequences. These keep data sorted linearly, like a dictionary, and also have logarithmic-time search, insert, and delete. The advantage of sorted sequences is that the data, is accessible in constant time. I.e., sorted sequences have the access advantages of both lists and trees. Sorted sequences are implemented by skip lists. The price one pays for convenience and lower access order is higher memory use.

Line Intersection Algorithms

The main algorithm used here is one of the three LEDA line segment intersection algorithms [50] [51]. A standard algorithm is provided, as is one based on Bentley and Ottman's line sweep, and one based on Mulmuley's randomized algorithm [56] [57]. The algorithm that is practically best is the line sweep algorithm; this has a much more organized processing structure, and causes pages to virtual memory much less frequently. As expected, the randomized algorithm performs poorly when virtual memory is used. The line sweep algorithm runs in time $O((n + s) \cdot \log n + m)$, where n is the number of segments, s is the number of vertices of the graph G , and m is the number of edges to be intersected.

LEDA's Impact

LEDA unquestionably allowed the completion of this thesis. Its accurate geometric computation and line intersection algorithms allowed an error-free sequential implementation to be used. In areas where I was restricted from using LEDA types (such as in the MFP data structure), or in specialized algorithms that LEDA didn't cover, much time-consuming work was done in debugging, coding special cases, and experimenting with special configurations of the datasets. LEDA definitely fulfilled its promise. Although, as is described in the next section, it did not come without its frustrations.

4.7 Difficult Problems

The following section highlights difficult problems that I encountered during implementation. The learning from these difficult problems is presented for the future use of ATLANTIS programmers and

designers, etc..

4.7.1 Programming Line Segment Algorithms

The difficulty of programming in computational geometry was illustrated by my efforts. Despite knowing of the difficulties of the task, I persisted in trying to write my own intersection algorithm.

It became apparent that using a good approximate algorithm had great difficulties. Merely switching from Pavlidis' line intersection algorithm [69] to a LEDA version caused great improvements in accuracy in my implementation.

Also, implementing all the special cases in line intersection was a large task, even with a well-designed version of the line sweep algorithm by van Kreveld [12]. Faced with time pressure, I switched to the LEDA *SWEEP_SEGMENTS* function [51], which performs well in practice, but uses a great deal of memory.

There are several versions of the LEDA line intersection algorithms; *SWEEP_SEGMENTS* performs the best because of its sequential memory access pattern; this contributes to a high memory (and disk) cache hit rate. The line sweep algorithm works better than the other implementations; for example, the LEDA implementation of Mulmuley's randomized algorithm [57] [56] [51] has a widely spread memory access pattern. When accessing virtual memory, this slowdown becomes huge relative to the execution time of the algorithm; the processor has been observed spending 90% of its time waiting for disk I/O.

Debugging the system for logic and numerical errors proved very time-consuming. This debugging pushed me several months behind, and well into the time of my new job. The biggest problem was that debugging each problem took as many as 10 to 15 runs of a failed case to detect the problem, and running each case takes 20 to 50 minutes on large file combinations. I eventually scaled down the size of my data files to combat this.

Numerical Robustness in the Real World

One final problem caused me to alter my approach. One of the design criteria that I set out for myself at the start of this effort was that the overlay engine should be as numerically robust as possible, including the ability to perform overlays on arbitrary datasets. This proved to be a naïve goal.

I eventually found a serious roadblock. Despite using LEDA to perform exact calculations, I was still storing the results of those calculation in C++ double-precision floating point values, as provided by Nussbaum's MFP software. For a few of the segments in my data files, this was not accurate enough; topological errors were made as a result of inaccurate representations of the edges of the MFP pages, and the location of the segment endpoints.

Rather than rewrite all of the MFP software to handle arbitrary-precision floating-point data (I would still be coding as you read this!), I re-scaled my data to have integer co-ordinates for the corners of all the MFP pages. This meant that the size of the world had to take into consideration the maximum reasonable number of page divisions that could take place in my datasets. The world size I eventually chose was $62,208 = 2^8 \cdot 3^5$. This allows for 5 page-splits in a 3×3 mesh, and 4 page-splits in a 4×4 mesh. In anything but a pathologically unbalanced dataset, this should handle on the order of hundreds of thousands of edges, without running out of integer page coordinates.

This approach removed numerical inaccuracy from one operand of the comparison operation in question (the MFP page edge). In all probability, the different, larger, more round scaling factors used to scale the data removed numerical inaccuracy from the endpoint calculations also. This solution enabled me to complete my testing.

Another technique useful in ensuring numerical robustness was the use of prime or relatively prime scaling factors to scale the data to the proper world size. In a real application, it may not be possible to scale data on demand to a more numerically robust size. However, this technique can be useful.

My conclusion here is that real-world numerical robustness will often demand special pre-processing of datasets in order to utilize existing algorithms.

4.7.2 Memory Management Issues

Memory management is an important part of high-performance GIS algorithms. In my case, memory was the limiting factor in the computations, and therefore worthy of special notice.

Because each MFP page is individually packaged and processed, the pages can be written back to disk and retrieved as necessary. This provides a useful method (and, with small processor sizes, often a necessary method) of managing virtual memory, which is limited on the systems I was testing on. While this slows down the algorithm considerably, it enables it to complete for comparison purposes.

Judging the performance of an algorithm by its performance based on virtual memory or disk-based access may not be the purest way of judging an algorithm's worth or performance potential. However, the tests conducted based on virtual memory are a good gauge of how an implementation works in on a real machine.

4.7.3 Meshing with Other Software

Dealing with third-party software was one of the most challenging parts of this thesis.

During the coding stage of my project, I was developing for a nonstandard platform, Solaris for Intel x86 series machines. This system/OS combination was not well supported at the time I started implementation. This made obtaining and compiling the correct versions of software challenging. For example, I was forced to fix header files from one package, because they defined one set of constants for Sparc/Solaris, and another for Intel, but none for Intel/Solaris.

Using C++, in particular the g++ compiler, was another source of frustration. The standard procedure for linking C and C++ code together is poorly documented, counterintuitive, and technically difficult. It is as if the C and C++ compilers were not built to work with each other. Standard features such as C++ inline functions are not supported by the C compiler. The C++ language itself is a hybrid with many nuances that I was only able to learn from the error messages spawned by the compiler.

Upgrading software to take advantage of needed features in new versions was also difficult. LEDA in particular only supports software built on the most recent versions of g++, the Linux kernel, etc..

This meant downloading, compiling, and installing several software packages every time I wanted to upgrade to a newer version of LEDA or other software.

Finally, bugs in third-party libraries were particularly nasty. Because of the complex nature of the software that I was developing, I could never be sure that the bug was not a fault such as a bad pointer overwrite in my code. I worked around a crash in LEDA sort routines that could have been pointer bugs in my code. (Curiously, I have after that point found no pointer bugs that could cause such a thing.) Because I was using MPI as well, there may be an interaction problem between the two of them that I did not detect.

Misleading documentation caused me to waste three weeks trying to debug a LEDA “bug” that didn’t exist. The solution was only discovered after several communications with LEDA technical staff: a special, poorly documented mechanism exists for extracting intersected line segments from the data structured output by the LEDA intersection algorithms.

Nevertheless, the use of LEDA saved me an enormous amount of time in the completion of this thesis. In a thesis that went eight months over schedule as it is, that’s a life saver.

4.7.4 Known Problems in the Current Implementation

The Segments Intersecting Page Corners, Collinear with Page Boundaries case (see Section 4.5.3, above), is not implemented. Although this case did come up during testing, it was possible to avoid it by using prime scaling factors when creating the datasets.

Chapter 5

Experimental Approach.

Some things are easier to measure, experimentally, than others. Recall the contributions and research goals set out in Chapter 1. These include the verification of OVID's design principles, ease of development by an end-user, the complexity of end-application development, and the parallel efficiency of OVID . The ideal research scenario here would include non-expert user trials of the software, and measurements of how well the software was adapted to other kinds of overlay.

Unfortunately, such a scenario is outside the scope of this research. The avenue of investigation that will be pursued is a quantitative analysis of speed-up, efficiency, load-balance, and other standard criteria for parallel computing. The algorithm implemented here consists of a line segment overlay driver, with full support for triangulation and/or polygonization extensions.

The experimental approach is designed here in Chapter 4.

5.1 Comparative Results

Performance Figures from other Parallel Vector Polygon Overlay computations seem to be closely guarded trade secrets. Of the parallel VPOs for which publications have been made, only a subset publish concrete results. Even the GIS group at Edinburgh University, a leader in parallel GIS work, has not published their performance Figures.

Andrews et al. [6] published a comparison of several sequential algorithms. Their main results showed that, in general, spatial partitioning algorithms (such as the binary space partition) perform the same overlay faster than spatial ordering algorithms (such as plane sweep). A relatively new spatial-ordering type algorithm, Chan's Trapezoid Sweep [15], is shown to be competitive with the spatial partitioning algorithms.

Shekhar, Rouada, Kumar, Chubb, and Turner published one of the few performance analyses of a parallel vector polygon overlay algorithm. On a Cray T3D with 16 processors, they achieved a maximum speedup of 14 [17]. However, this was achieved on a limited subset of polygon overlay, rectangular range queries.

Hopkins, Healy, and Waugh [36] achieved a parallel speed-up of 3.5 when comparing VPO run on 17 processors to 81 on a 400-transputer Meiko supercomputer, using a grid-cell method. This was achieved on a dataset of $20,000 \times 40,000$ line segments. The efficiency reached was 70% on that range of processors.

Narayanaswami and Franklin [31] implemented line intersections on a hypercube computer in 1992. They achieved a top speed-up of 2.0 when comparing a 16-processor overlay to a 32-processor overlay. Their data set contained 3800 edges, and generated 16823 intersections.

5.2 Quantitative Measures of Interest

Several quantitative measures will be used to evaluate the performance of OVID, and therefore the validity of its approach. These measures are:

- ★ **Total Time:** the actual time spent from the launch of OVID until its completion. $\text{Total Time} = \text{Total Communication} + \text{Total Computation}$.
- ★ **Total Communication:** the total time spent in the communication phase of OVID operations.
- ★ **Idle Time.** This includes time in the communication phase spent waiting for other processors to finish their task.

- ★ **Total Computation:** the total time spent in the computation phase of all CD iterations.
- ★ **Imbalance:** for a group of p processors, $\text{Imbalance} = c_{\max} \cdot p - \sum_{i=0}^p c_i$, where $c_{\max}$ is equal to the maximum computation time registered on any of the processors, and c_i is the computation time spent on the i^{th} processor. This represents the aggregate time spent waiting by the processors. Divide imbalance by p to get a measure of per-processor imbalance. It should show an inverse relationship to idle time.
- ★ **Speed-Up:** $\frac{t_p}{t_1}$, where t_p is the total time spent on a task in parallel by a set of p processors, and t_1 the total time spent on the same task a single processor. A program is said to have *linear* speedup if $\frac{t_p}{t_1} \approx p$; the program is said to be *scalable* if linear speedup is maintained as p increases [48].
- ★ **Efficiency:** Speed-up divided by number of processors. This is usually less than 100%. It is difficult to do work more than p times faster than a uniprocessor given that one only has p processors.
- ★ **Rounds:** The number of communication rounds required to complete the segment linking (“doppelgänger resolution”).

Some analysis will be done on a phase-by-phase basis for various parts of the algorithm.

5.3 Test Platform

The testing described here was done on two computing platforms.

One is the PARADIGM Group’s ASP (*A Symmetrical Processor*) computer. This machine consists of 16 166-MHz Pentium microcomputers connected by a dedicated 100Mb/s Nortel/Bay Networks crossbar switch. It runs the Linux operating system. Each processor has 96MB of memory, except the root node which has 128MB. The switch enables arbitrary connections to be established between processors. In the past, it has been observed that the internal architecture of the switch seems to favour division into two subgroups of 8.

NAME	#VERTICES	#EDGES
Africa	5,000	14,798
America	5,000	14,787
Madagascar-15k	15,000	44,581
Australia	25,000	74,882

Table 5.1: Terrains used for testing the performance of the algorithm.

The second is a smaller ASP-2 machine, built from eight 450-MHz Pentium-II machines, connected by fast Ethernet. It runs Solaris for Intel x86. Each of these nodes has 128Mb of main memory. This was the platform that I used for development.

5.4 Experimental Data

The experimental data is the same as was used by Nussbaum and Lanthier [48] [61]. This tests the performance of the algorithm on real-world data.

The re-use of this is deliberate, as it helps establish a baseline for comparing results of MFP-based algorithms. While the characteristic behaviour of Nussbaum and Lanthier's shortest path implementation is very different from the polygon vector overlay implementation, using the same data removes one level of uncertainty in the comparison.

Nussbaum and Lanthier used TIN (Triangulated Irregular Network) data in their research. The data used by the following TINs were all formed from cropped DEMs, modified by the removal of vertices. Vertices were removed so as to optimize the volume difference between the gridded DEM and the reduced TIN [48].

The list of TINs is given in Table 5.1.

All ten of the TIN files have been reproduced here, in Figures 5.1, 5.2, 5.4, and 5.3.

Figure 5.1: Madagascar TIN

5.5 Hypotheses

There are many factors that help determine the performance of parallel algorithms. For this research, four factors will be examined as valuations of the OVID approach: efficiency, especially at higher degrees of parallelism, load-balance, particularly with clustered datasets, performance with large datasets, and behaviour under various partition sizes.

5.5.1 Parallelism: Factoring the Number of Processors

This parallel implementation is based on a CGM algorithm and executed on a coarse-grained, distributed system. Such implementations are well oriented towards obtaining good speedups. Since a small number of processors is utilized relative to the amount of data processed, parallel communication is kept to a minimum. All the advantages of a good sequential algorithm are utilized, yet with a smaller

Figure 5.2: America TIN

dataset for each processor.

When run on multiple processors, the algorithm is run with an aggregate processor cache, main memory, and swap space on disk $O(p)$ larger than the uniprocessor version. Coupled with an efficient algorithm that has a high cache hit rate, the multiprocessor algorithm can be expected to do quite well against its uniprocessor cousin.

5.5.2 Data Factors: Clustering and Size

Because the datasets are relatively large, and most of them are closer to uniform distribution than not, relatively consistent performance between processors is expected. This will be true in cases where files are more highly subdivided. Since the MFP was designed to promote load-balance on highly clustered datasets, some emphasis will be placed on datasets that tend to cluster.

Figure 5.3: Africa TIN

5.5.3 Partitioning: Changing Data Granularity

When data is split into large pieces between parallel processors, less communication takes place and fewer of the geometric primitives are duplicated. This tends to speed up the overall calculations. However, data partitioning into large pieces promotes load imbalance in clustered data, and may lead to use of virtual memory on a standard machine. Both of these tend to slow down the algorithm. The ideal partition size will depend on processor type, memory size, algorithm design, the characteristics of data, and a number of other things. My hypothesis is that partition into large chunks of data will be more efficient in all but the bad cases of data distribution, or on small numbers of processors.

Figure 5.4: Australia TIN

5.6 Test Procedure

The test plan tested the three main hypotheses listed above. That is, the tests concentrated on examining the efficiency, load-balancing, and large dataset behaviour of OVID. A series of tests were run isolating each factor in turn. Each test was repeated a number of times in order to obtain a statistically significant result.

The first dimension of the test plan was speed-up and efficiency. Various meshes of size from one to sixteen processors were tested: 1×1 , 2×2 , 3×2 , 3×3 , 4×3 , and 4×4 .

The second factor tested was clustered data. In practical terms, this requires tests overlaying rasters with uneven data distributions. Here, the America and Australia TINs were used as an example of an unevenly distributed dataset.

The third factor examined was multi-level partitioning. This tested the difference between few large partitions and many small ones on load-balance and overall efficiency. Of particular interest was the measure of the efficiency of OVID vs. that of the virtual memory system in Linux. The TIN files were partitioned with different thresholds of the maximum number of vertices in a page. Two complete sets of data were generated: :

★ Tests done using a high page splitting threshold:

- Maximum 5000 arcs per MFP page for input, and
- Maximum 10000 arcs per MFP page for output.

★ Tests done using a low page splitting threshold:

- Maximum 2500 arcs per MFP page for input, and
- Maximum 5000 arcs per MFP page for output.

The fourth factor researched was the effect of dataset size. Here, tests were done with overlays of relatively small datasets (about 10,000 edges), larger datasets (30,000 to 50,000 edges), and large datasets with small ones. The four TINs representing portions of Africa, Australia, America, and Madagascar were chosen as representative test data. Tests were done on the complete pairwise intersection of these TINs. All TINs were scaled to the same world size in order to perform the overlay.

Measurements were done on CPU time and real time to completion, on the algorithm as a whole, and on three separate phases of the algorithm:

- ★ overlay,
- ★ propagation of newly generated pages, and
- ★ resolution of doppelgänger segments.

Category	Quantity = Q_i
Number of processor meshes used	6
Number of TIN pairs on which overlay was performed	6
Number of different partition thresholds used	2
Number of times each test was repeated (minimum)	4
Product total = $\prod_{i=0}^3 Q_i$ = number of test cases executed	288

Table 5.2: Breakdown of Test Plan

Dataset sizes and their relationships to load-balance were also examined.

In total, 288 test runs were done. This number was broken down as given in Table 5.2.

The tests listed above were executed on a version of the overlay driver without attribute processing or inclusion/exclusion implemented.

5.7 Results

This section gives the results of the above tests. The tests are analyzed with respect to speed-up, file partitioning balance, and performance improvements due to better load-balance.

5.7.1 Results Summary

The results obtained in this research were pleasing. Given the variety of the data, and its load-imbalance characteristics, speedups of between 7 and 12 on a 16-processor distributed system are quite good.

The performance of MFP on clustered data was also satisfying. Although the tests executed did not fully harness the power of MFP to break down clustered data, breaking the data into more pages gave a 20% improvement in speed-up on the largest dataset considered.

A twofold decrease in the maximum allowed page size in an MFP gains up to 20% efficiency as a result of improvements in the load-balance. This occurs despite increasing total aggregate file sizes by as much as 15% to 20%, and increasing the number of pages handled by each processor. The trend seen in the data indicates that better results should be seen on larger datasets.

TIN file	High-T σ	Low-T σ	improvement (Low over High)
Africa	226760	226760	0.0%
America	149751	149751	0.0%
Madagascar	456896	456896	0.0%
Australia	1358502	1149390	18.2%

Table 5.3: Raster Partitioning: Standard Deviations for Per-Processor File Size, 16 processors.

Figure 5.5: Raster Partitioning: Percent Improvement in Standard Deviation, Australia TIN.

Figure 5.6: Standard Deviations of Overlay files' Per-Processor file sizes (High Threshold)

5.7.2 Load-Balance in Datasets

The Generation of Data Sets

Partitioning the rasters had significant effects, particularly in terms of parallel load-balancing. When dividing up an unbalanced dataset among processors, such as the Australia TIN, decreasing the partitioning threshold in an MFP page increases the number of pages. Data clusters formerly assigned to a single processor are divided between all of the processors.

The small rasters generally did not benefit from a lower partition threshold, since with $p > 1$ there was only a single page per processor, due to the small number of line segments. The large rasters did, particularly at larger p . While greater subdivision implies a higher number of shared segments and therefore a greater aggregate workload, the load-balance gain was significant.

Table 5.3 shows the standard deviations for the four rasters in question, with sixteen processors. The difference between High-Threshold and Low-Threshold partitions is highlighted. Note that the standard deviation for the 16 processor case for the three smaller TINs is identical. Only the Australia TIN shows

Figure 5.7: Standard Deviations of Overlay files' Per-Processor file sizes (Low Threshold)

a difference, an 18% improvement.

The larger Australia TIN had significant improvements in standard deviation for many of the mesh sizes. This is shown in Figure 5.5. Note the large drop in improvement from 6 processors to 9, and the subsequent increase to 16 processors. Of the other TINs, the Madagascar TIN actually shows less balance at $p = 4$ (above $p = 6$, the partitioning schemes produced identical results).

These results suggest that low-threshold partitions produce significant improvements in larger TINs. It also hints at the behaviour pattern of these improvements: while the trend is toward lower thresholds generating more balanced data distribution, discontinuities do occur in the pattern.

Primarily, as the number of processors, p , increases, partitions that take place with smaller p are not made, since the partition threshold is not reached when splitting data among more processors. This accounts for the sudden drop in improvement seen in the Australia TIN. Second, depending on the data in question and the partitioning scheme, higher partition thresholds may produce better partitions in some cases.

Figure 5.8: Load Imbalance for Six Overlay tests, High Thresholds. Seconds vs. Number of Processors.

Figure 5.9: Load Imbalance for Six Overlay tests, Low Thresholds. Seconds vs. Number of Processors.

Load-Balance of Output Dataset

The datasets output by the OVID algorithm give a better picture of the potential of MFP to load-balance parallel processing jobs. When using a high threshold of 10,000 nodes for page division, the resulting load imbalance is visibly worse than for the low-threshold case; it is also more variable, leading to more bad cases. This comparison can be made between Figures 5.6 and 5.7. Standard deviation is used as a measure of the variability of the file sizes assigned to the overlaid data set for each processor.

The benefit of the lower-threshold split is most obvious at low levels of multiprocessing; on the large Australia $\times$ Madagascar output, the standard deviation of a 4-processor file size is reduced by 52%. On the other hand, the standard deviation is 16% higher for low-threshold data on the same output, when compared to high-threshold.

The trend is apparent that, for most datasets, low-threshold splitting produces better average load-balance. However, a good (or bad) fit between a particular configuration of processor number, processor mesh dimension, and data clustering often combine to produce unusually good, or bad, load balances.

The load imbalance charts (Figures 5.8 and 5.9) show the aggregate total of time wasted during the sequential overlay phase of the line segment overlay. Because these charts show aggregate wasted time, a completely load-balanced computation would show a straight line gently sloping upward as the number of processors increased. The low-threshold computations have a clear advantage in this comparison; only in one case does the high-threshold computation show significant advantage.

The high-threshold computation diagram shows somewhat irregular results. It has some very good data points - a “sweet spot” in the 12-processor case of Australia $\times$ Madagascar wastes 42% less compute cycles than the corresponding low-threshold computation. The low-threshold diagram is much better for $p = 4$ and $p = 16$, and competitive for $6 \leq p \leq 12$. Although individual datapoints can be deceptive, the trend appears to favour low-threshold computations for higher p , in the wasted time category.

Furthermore, wasted time is not a priority measure; speed-up (see below) is.

Test	File Size	Imbalance σ	Speed-Up	Efficiency
Africa x America	6.5Mb	425kb	10.5	65.6%
Africa x Madagascar	31.5Mb	797kb	8.87	55.4%
Africa x Australia	36.4Mb	865kb	6.76	42.3%
America x Madagascar	34.0Mb	139kb	11.0	68.7%
America x Australia	36.4Mb	1046kb	7.41	46.3%
Australia x Madagascar	50.8Mb	1251kb	7.25	45.3%

Table 5.4: Speed-ups, for 16 processors, using high page splitting threshold.

5.7.3 Speed-ups and Run-time.

The high-threshold speed-ups for the entire line segment overlay operation were good (cf. Table 5.4). The best speed-up factor of 11 was obtained on the most load-balanced of the test sets; conversely, the least balanced operation showed the smallest speed-up.

Load-balance appeared to be the dominating factor. *Imbalance* refers here to “Imbalance σ ”, standard deviation of the result (per-processor) file size, graphed in the previous section. This particular measure of imbalance doesn’t have a direct correlation to the speed-up obtained, but an approximate inverse relationship can be discerned.

“File Size” is the cumulative size of the segment and point data stored in the file. In the previous section, load-balance was demonstrated to be a roughly predictable if irregular attribute of the MFP division scheme. Not so with speed-up. As can be seen in Figures 5.10 and 5.14, the speed-up curves are relatively smooth. This is particularly true with the extra load-balancing available with the low-threshold runs. The speed-up curves for low-threshold tests are smooth and predictable. The low-threshold data shows better speed-ups for the low-threshold partitions (Figure 5.5), from a minimum of ≈ 8 to a high of ≈ 12 .

Table 5.6 shows up to a 20% improvement in efficiency using low-threshold partitioning. With one exception (a 3.4% loss) the lowest-efficiency test cases from the high-threshold tests show the most marked improvement. This is unspectacular, but it shows a good trend. In theory, the low-threshold partitioning has better potential on larger, less balanced files. Given the increasing improvement on the Australia×Madagascar test, one can expect that better improvements will occur on larger datasets.

Figure 5.10: Performance Curve (total time). Seconds of Execution vs. Number of Processors. High-threshold Data Files.

Results for Various Algorithm Phases

Of note is the contribution of sequential overlay and parallel messaging to the total time spent processing. The sequential overlay phase of the algorithm (Figures 5.11, 5.15) is very smooth and predictable, and shows good speedups, from ≈ 7 to ≈ 12 depending on the test case. In some cases, the combination of extra memory and disk access capability produces near-optimal speedups on pure sequential overlay. Again, the low-threshold data sets showed good or better results than high-threshold, particularly for small p and larger data sets.

The parallel messaging (Figures 5.12, 5.16) produces a spike in the overall performance. This messaging is the actual parallel propagation of doppelgänger segments. The four-processor case shows the most messaging congestion, with dramatic improvements as p increases. The 1-processor cases may require time to do parallel messaging, as its data is broken into many separate pages in order to be able to execute the overlay algorithm within the memory on one parallel node.

The parallel page exchange phase of the algorithm (Figures 5.13, 5.17) works relatively well. This phase

Figure 5.11: Performance Curve (sequential overlay phase). Seconds of Execution vs. Number of Processors. High-threshold Data Files.

Test	Result Size	Imbalance σ	Speed-Up	Efficiency
Africa x America	6.5Mb	425kb	11.8	73.8%
Africa x Madagascar	31.6Mb	797kb	8.57	53.6%
Africa x Australia	40.5Mb	773kb	7.87	49.2%
America x Madagascar	34.0Mb	139kb	11.3	70.8%
America x Australia	38.3Mb	1193kb	8.40	52.5%
Australia x Madagascar	57.5Mb	1343kb	8.65	54.1%

Table 5.5: Speed-ups, for 16 processors, using low page splitting threshold.

includes the sending of pages generated from splitting during the overlay phase, plus the generation of node lists for the arcs, an $\Omega(n \log n)$ algorithm. Because page sizes decrease from a high of 1Mb in the 4-processor cases to an average of 100kb to 200kb in the 16-processor cases, both communication times and computation times decrease quickly. The uniprocessor cases show the dominance of the node list generation algorithm, since a single processor case needs do little communication here¹.

¹In the case of the larger files, a single processor must break the computation into several MFP pages, in order to be able to fit the overlay computation into memory.

Figure 5.12: Performance Curve (post-overlay parallel segment resolution phase). Seconds of Execution vs. Number of Processors. High-threshold Data Files.

Figure 5.13: Performance Curve (post-overlay parallel page exchange phase). Seconds of Execution vs. Number of Processors. High-threshold Data Files.

Test	Improvement using Low-Threshold Page Splits
Africa x America	12.7%
Africa x Madagascar	-3.4%
Africa x Australia	16.4%
America x Madagascar	3.2%
America x Australia	13.3%
Australia x Madagascar	19.3%

Table 5.6: Speed-up improvement for 16 processors using Low-Threshold Page Splits.

Figure 5.14: Performance Curve (total time). Seconds of Execution vs. Number of Processors. Low-threshold data files.

Figure 5.15: Performance Curve (sequential overlay phase). Seconds of Execution vs. Number of Processors. Low-threshold Data Files.

Figure 5.16: Performance Curve (post-overlay parallel communication phase). Seconds of Execution vs. Number of Processors. Low-threshold Data Files.

Figure 5.17: Performance Curve (post-overlay page exchange phase). Seconds of Execution vs. Number of Processors. Low-threshold Data Files.

Chapter 6

Conclusions and Future Direction

This chapter presents the analysis and conclusions of this research, and presents future research directions leading from this research.

6.1 The Effectiveness of MFP-based Vector Polygon Overlay

Overall, the MFP data structure, and its associated algorithms, proved their worth.

Theoretically, the algorithms produced were as efficient as the best mesh-based algorithms, while bringing good load-balance properties to bear on problems.

Practically, the parallel efficiency of 45% to 73% was quite good for a spatial problem on 16 distributed processors. The dataset was taken from the “real world”, and the parallel approach worked well for the “real world” environment for which it was designed. In addition, the trends established show promise for good speed-ups on bigger data sets.

Implementationally, the numerical problems experienced were trying, as many, many special cases presented themselves. This is the most difficult area, and one that requires further work. But, although there are still special cases to be programmed, the implementation state of the algorithms is relatively robust numerically, and can be used with minor adjustments to handle a wide variety of data.

The weakness of the implementation side of OVID is its requirement that any two datasets be scaled to the same world size of raster. This currently requires a separate, manual facility for data processing.

6.1.1 The Effect of Parallelism

The effect of parallelism was acceptably good on this dataset. On a distributed machine with high relative communication costs, efficiencies bordering on 75% were recorded.

One trend worth noting is be the good speed-up of the overlay and per-arc node list calculation portions of the algorithm. The data for those parts of the algorithm showed a near-optimal speed-up. For significantly bigger datasets, in which computation greatly dominates communication, parallelism may allow for near-optimal or even super-optimal overall results.

6.1.2 The Effect of Data Clustering

Data clustering and therefore load imbalance was a major factor in determining the efficiency of a parallel algorithm. By observation of the four test rasters, and by the standard deviations of the resulting per-processor file sizes, load-balance was the primary agent in determining the speed-up of the MFP algorithm. Balanced datasets trended towards 70% efficiency; less balanced towards 50%. The MFP algorithm showed promise here, in that it yielded the most substantial increases for the largest and least-balanced datasets.

6.1.3 The Effect of Dataset Size

There was no significant observed effect on the MFP algorithm speed-up due to dataset size. While the Africa×America test case produced a high speed-up on a small dataset, and slowdowns were seen on the larger datasets, high speed-ups were seen on the America×Madagascar test case. The latter was within the same range of output size as three of the other four results, which had widely varying speed-ups. The largest test case, Australia×Madagascar, showed a lesser speed-up because of its load-imbalance.

6.1.4 Comparisons to Other Implementations

Chubb et al. [17] obtained a maximum speed-up of ≈ 14 on 16 Cray processors. My work, although not involving the entire Vector Polygon Overlay operation, managed a maximum speed-up of ≈ 12 on a distributed system. While the total time for the Cray to process an equivalent number of line segments is much smaller, a Cray machine also costs much more.

Healey et al. [36] obtained a speed-up of 3.5 on 81 processors, compared to 17 processors, on a Meiko supercomputer. This yields an efficiency of 70%, which is approximately equal to my best results. Again, the machine used to generate these results was expensive.

A new, faster system comparable to the Asp can be constructed today for something in the range of \$10,000; with a demonstrated good distributed speed-up, this should make parallel overlay operations widely available.

6.2 Ease of Programming

The framework delivered here is both fully generic – its model of computation able to handle a full range of binary operations – and customizable. Customization for a particular type of overlay or application may require work, but the architecture is set up to minimize that work.

The interface given in Section 4.2.4 on Page 64 is quite easy to use. The user must simply implement the operations pertaining to the application that he or she is working with. All parallel considerations are part of the lower layers, transparent to the user. Numerical aspects, except for those specific to the application, are dealt with in the OVID Algorithm Layer; the user must deal with these only in a limited way, for example in a splinter polygon method. Questions of meaning of data and attribute processing are left for the user, to program in a way pertinent to the application.

The architecture thus delivers into the hands of an application builder a framework that handles most of the difficult technical issues, while enabling precise direction for application-specific issues. When fully complete, OVID will provide GIS applications programmers with a useful, powerful, and flexible interface.

The development of a full application was not part of this research. This aspect of the Atlantis project will be left for future exploration. Nevertheless, this research contributes substantially to breaching the threshold of accessibility for the GIS domain.

6.3 Parallel Algorithms

The creation of parallel algorithms for MFP-based overlay is an important part of this research.

Given a library of general sequential geometric algorithms (such as line intersection and convex hull), a user can implement MFP-based parallel algorithms with not too much extra work. The MFP-based algorithms here give a basis for the development of the rest of the OVID Vector Polygon Overlay driver, plus a good start on a library of parallel geometric algorithms for use in GIS.

Particularly valuable is the specification of algorithms for propagating geometric primitives across MFP page boundaries.

6.4 Further Research

This thesis deals primarily with theoretical and technical issues. User issues, while not addressed directly here, are nonetheless an essential part of ongoing research within the PARADIGM group.

Future research goals should include:

- ★ A demonstration that an example application can be developed without recourse to expertise in parallel computing, numerical computing, sophisticated data structures, or geometric computing.
- ★ A demonstration by the same application that development can be done easily and at a conceptual level targeted to that of the end-user.
- ★ The evaluation of ease-of-use and application development by the size and complexity of the application that results from implementation on top of VPO and OVID.

- ★ A demonstration that datasets with multiple attributes can be easily processed. The mechanism for attaching attributes to pure geometric constructs is already in place.
- ★ The implementation and testing of MFP-based Polygonization and Triangulation algorithms.
- ★ Further tests on the effectiveness of MFP line segment overlay and MFP-VPO, on larger and more varied datasets.
- ★ Further improvements, and quantitative and topological analysis of numerical problems in the parallel intersection and merge phases of the algorithms and their implementations.
- ★ The development of automated tools to pre-process the datasets for use by the overlay driver.

Bibliography

- [1] Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaing, C., and Yap, C., "Parallel Computational Geometry", in *Proc. 26th Symp. Foundations of Comp. Sci.*, Portland, OR, pp. 468-477, 1985.
- [2] Akl, S. G., and Lyons, K., A., *Parallel Computational Geometry*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.
- [3] Amato N. M., Goodrich, M. T., and Ramos, E. A., "Computing faces in segment and simplex arrangements", *Proc. 27th ACM Symp. Theory of Computing*, pp. 672-682. New York, 1995.
- [4] Anderson, D., et al.. "SETI@home Web Page", <http://setiathome.ssl.berkeley.edu/> . University of California at Berkeley, 1999.
- [5] Anderson, W. H., Faust, N. L., and Star, J. L., "Geographic Information Systems and Remote Sensing Future Computing Environment," *Photogrammetric Engineering and Remote Sensing*, v. 57 no. 6, 1991, pp. 655-668.
- [6] Andrews, D. S., Boritz, J., Chan, T., Denham, G., Harrison, J., Snoeyink, J., and Zhu, C., "Further Comparison of Algorithms for Geometric Intersection Problems", *6th Int'l Symposium on Spatial Data Handling*, v. 1, pp. 709-724, Sept 1994.
- [7] Atallah, M. J., and Goodrich, M. T., "Efficient Plane Sweeping in Parallel," *Proceedings, 2nd Symposium on Computational Geometry*, York Town Heights, N.Y., pp. 216-225, 1986.
- [8] Avnaim, F., Boissonnat, J.D., Devillers, O., Preparata, F., and Yvinec, M., "Evaluating signs of determinants using single-precision arithmetic", *Algorithmica* vol. 17, 1997, pp 111-132 , 1997.

- [9] Bäumker, A., Dittrich, W., and Meyer auf der Heide, F., "Truly Efficient Parallel Algorithms: c-Optimal Multi-search for an Extension of the BSP model. *Proc. European Symp. on Algorithms*, pp. 17-30, 1995.
- [10] Beberg, A. L., *Cosm Phase I*, Mithral Communications & Design, 1999. Web Page at <http://cosm.mithral.com/index.html>.
- [11] Bentley, J. L., and Ottman, T., "Algorithms for reporting and counting geometric intersections." *IEEE Trans. Computing*, C-28 (1979), 643-647.
- [12] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, M., *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
- [13] Boka, V.V., Himabindu, G., Olariu, S., and Schwing, J. L., "Constant-Time Algorithms for Constrained Triangulations on Reconfigurable Meshes," in *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1057-1072, Nov. 1998.
- [14] Brownstein, M. A *\$1000 Supercomputer*, article in cnn.com web site, June 15, 1999.
- [15] Chan, T. M., "A simple trapezoid sweep algorithm for reporting red/blue segment intersections." in *Proc. 6th Canadian Conf. of Computational Geometry*, pp. 263-268, Saskatoon, Canada, 1994.
- [16] Chazelle, B., and Edelsbrunner, H., "An Optimal Algorithm for Intersecting Line Segments in the Plane." *Journal of the ACM*, vol. 39, no. 1, pp. 1-54, 1992.
- [17] Chubb, D., Kumar, V., Ravada, S., Shekhar, S., and Turner, G., "Load Balancing in High Performance GIS: Partitioning Polygonal Maps", *Proc. Intl. Symp. on Large Spatial Databases*, Springer Verlag (Lecture Notes in Computer Science), (1995)
- [18] Dehne, F., Fabri, A., and Rau-Chaplin, A., "Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers", in *Proc. ACM 9th Annual Computational Geometry*, pp. 298-302, 1993.

- [19] Diallo, M., Ferreira, A., Rau-Chaplin, A., and Ubaigueda, S., "Scalable 2D Convex Hull and Triangulation Algorithms for Coarse Grained Multicomputers", in *Journal of Parallel and Distributed Computing*, vol. 56, pp. 47-70, 1999
- [20] Douglas, D., "It makes me so CROSS" (1974), in Peuquet, D. J. and Marble, D. F. (eds), *Introductory Readings in Geographic Information Systems*, Taylor & Francis, London, 1990, pp. 303-307.
- [21] Dowers, S., Gittings, B., Healey, R., and Mineter, M., eds.. *Parallel Processing Algorithms for GIS*. Taylor and Francis, London, 1998.
- [22] Dowers, D., Harding, T. J., Healey, R., and Hopkins, S. , "Vector Polygon Overlay", in Dowers, S., Gittings, B., Healey, R., and Mineter, M., eds.. *Parallel Processing Algorithms for GIS*. pp. 265-310. Taylor and Francis, London, 1998.
- [23] Dowers, S., and Sloan, T. M., "Parallel Vector Data Input", in Dowers, S., Gittings, B., Healey, R., and Mineter, M., eds.. *Parallel Processing Algorithms for GIS*. pp. 151-178. Taylor and Francis, London, 1998.
- [24] Edelsbrunner, H., and Mücke, E. P., "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms", in *Proc. Fourth ACM Symp. on Computer Geometry*, pp. 118-133, June 1988.
- [25] ESRI, Inc. PC ARC/INFO: Powerful GIS Functionality for DOS and Windows.
- [26] ESRI, Inc. "Topological Overlay", in *The GIS Glossary* (http://www.esri.com/library/glossary/t_z.html#TOP_OVERLAY), ESRI, Inc., 1999.
- [27] Flynn, M.J.. "Some computer organizations and their effectiveness," in *IEEE Trans. Computers*, vol. 21 no. 9, pp. 940-960, 1972.
- [28] Fortune, S., and Wylie, J., "Parallelism in Random Access Machines". *Proceedings ACM Symp. Theory of Computing*, pp. 114-118, 1978.

- [29] Franklin, W.R., Narayanaswami, C., and David Sun, M. K., Wu, P. Y. P., and Zhou, M.-C., "Uniform Grids: A Technique for Intersection Detection on Serial and Parallel Machines." *AutoCarto 9*, pp. 100-109.
- [30] Fortune, S. , and Van Wyk, C. J., "Efficient Exact Arithmetic for Computational Geometry", in *Proc. 9th Annual Symp. Computational Geometry*, pp. 163-172, ACM, 1993.
- [31] Franklin, W. R., and Narayanaswami, C., "Edge intersection on the hypercube computer", in *Information Processing Letters*, v. 41, pp. 257-262, 1992.
- [32] Goldberg, D., "What Every Computer Scientist should know about Floating-Point Arithmetic", in *ACM Computing Surveys*, v. 23 no. 1, pp. 5-48, March 1991.
- [33] Goodman, J.E., and O'Rourke. J., *Handbook of Discrete and Computational Geometry*, 1997. CRC Press, Boca Raton, Fl., 1997.
- [34] Guibas, L. J., and Seidel, R. "Computing convolutions using reciprocal search." *Discrete Computational Geometry*, v. 2, pp. 175-193, 1987.
- [35] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching." *ACM SIGMOD*, vol. 13, 1984, pp. 47-57.
- [36] Healey, R. G., Hopkins, S., and Waugh, T. C.. "Algorithm Scalability for Line Intersection Detection in Parallel Polygon Overlay", in *Proc. 5th Intl. Symp. Spatial Data Handling*, Charleston, S.C.. pp. 210-218, 1992.
- [37] Hecker, C., Roytenberg, D., Sack, J.-R., Wang, Z., "System development for parallel cellular automata and its applications", *Future Generation Computer Systems*, vol. 16 (1999), pp. 235-247.
- [38] Hecker, C., *The Metasynthesis of Domain-Specific Languages*. University of Regina Technical Report CS-94-06, 1994.

- [39] Hoffmann, C. M., "The Problems of Accuracy and Robustness in Geometric Computation", *IEEE Computer*, pp. 31-41, March 1989.
- [40] Hoffman, C. M., Hopcroft, J. E., and Karasick, M. S., "Towards Implementing Robust Geometric Computations", *Proc. Fourth ACM Symp. on Computer Geometry*, pp. 106-117, June 1988.
- [41] Hutchinson, D., Lanthier, M., Maheshwari, A., Nussbaum, D., Roytenberg, D., and Sack, J.-R., "Parallel Neighbourhood Modeling", *Proc. ACM GIS*, pp. 26-34, 1997.
- [42] Hutchinson, D., Kttner, L., Lanthier, M., Maheshwari, A., Nussbaum, D., Roytenberg, D., and Sack, J.-R., "Parallel Neighbourhood Modeling", *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '96)*, Padua, Italy, pp. 204-207, 1996.
- [43] Jang, J., Nigam, M., Prasenna, V. K., and Sahni, S., "Constant Time Algorithms for Computational Geometry on Reconfigurable Mesh", in *IEEE Trans. Parallel and Distributed Systems*, vol. 8, pp. 1-12, 1997.
- [44] Joy, B., "Why the future doesn't need us," *Wired*, vol. 8 no. 4, pp. 238-262, April 2000.
- [45] Kirkpatrick, D., *1997 Cognos Lecture*, Carleton University, October 1997.
- [46] Klinkenberg, B., *Geography 516: NOTES on the History of GIS*. Web page, UBC, 1999.
<http://www.geog.ubc.ca/courses/klink/g516/notes/history.htm>
- [47] van Kreveld, M., "Digital Elevation Models and TIN Algorithms," in van Kreveld, M., Nievergelt, J., Roos, T., Widmayer, T., eds., *Algorithmic Foundations of Geographic Information Systems*, Lecture Notes in Computer Science no. 1340, pp. 37-78, Springer, 1997.
- [48] Lanthier, M. *Shortest Path Problems on Polyhedral Surfaces*. Ph.D. thesis, School of Computer Science, Carleton University, Dec. 1999.
- [49] Lee, F., "Parallel Ice Tracking", M.C.S. thesis, School of Computer Science, Carleton University, 1996.

- [50] Mehlhorn, K., and Näher, S.. “LEDA: A Platform for Combinatorial and Geometric Computing”. *Communications of the ACM*, v. 38, pp. 96-102, 1995.
- [51] Mehlhorn, K., and Näher, S.. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [52] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Document Number 860-0128-001. University of Tennessee, 1995.
- [53] Milenkovic, V. J.. “Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic”, in *Artificial Intelligence*, vol. 37, 1988, pp. 377-401.
- [54] Miyano, S., Shiraishi, S., and Shoudai, T.. *A List of P-Complete Problems*. Web page <http://www.i.kyushu-u.ac.jp/~seki/P-complete/all/all.html>, Kyushu University, Fukuoka, Japan, 1996.
- [55] Mount, D., *CSMC 427/828M: Computer Graphics*, lecture notes, Department of Computer Science, University of Maryland, College Park, MD, 1997. See §4 and 5.
- [56] Mulmuley, K., “A fast planar partition algorithm.” *Proceedings of the 29th Annual IEEE Symp. on Foundations of Computer Science*, pp. 580-589, IEEE, New York, 1988.
- [57] Mulmuley, K., *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1994.
- [58] National Aeronautics and Space Administration. *SRTM C-Band Radar Data Processing Timeline*. Jet Propulsion Laboratory, Pasadena, California, 2000. Website: http://www.jpl.nasa.gov/srtm/data_timeline.html
- [59] Nievergelt, J., “Introduction to Geometric Computing: From Algorithms to Software”, in van Kreveld M., Nievergelt, J., Roos, T., Widmayer, T., eds., *Algorithmic Foundations of Geographic Information Systems*, Lecture Notes in Computer Science no. 1340, pp. 1-19, Springer, 1997.

- [60] Nievergelt, J., and Preparata, F. P., "Plane-Sweep Algorithms for Intersecting Geometric Figures", *Communications of the ACM*, vol. 25, no.10, pp. 739-747, 1982.
- [61] Nussbaum, D., *Design and Implementation of Spatial Data Handling Systems*. Ph.D. thesis, School of Computer Science, Carleton University, in preparation, 2000.
- [62] Nussbaum, D., *Atlantis -a vector base GIS skeleton*. Internal unpublished document, draft version, October 15, 1998, Paradigm Group, Carleton University.
- [63] Nussbaum, D., *Data Modeling in Two and Three Dimensional Spaces on Parallel Computers*. Internal unpublished document, draft version, October 1998, Paradigm Group, Carleton University.
- [64] Nussbaum, D., *Spatial Tool Kit*. Internal unpublished document, draft version, October 1998, Paradigm Group, Carleton University.
- [65] Nussbaum, D., *Atlantis prototype C header file*. Internal unpublished document, draft version, October 1998, Paradigm Group, Carleton University.
- [66] Nussbaum, D., private email correspondence, Feb. 8, 1999.
- [67] Orenstein, J., "An Algorithm for Computing the Overlay of k-Dimensional Spaces." *Advances on Spatial Databases, SSD'91, Zurich*. pp. 381-400. Springer-Verlag, 1991.
- [68] O'Rourke, J., *Computational Geometry in C*. Cambridge University Press, Cambridge, U.K., 1994.
- [69] T. Pavlidis, *Algorithms for Graphic and Image Processing*, Computer Science Press, Rockville, MD, USA, 1982. Cf. §14.4 (pp. 326-333).
- [70] Pescovitz, D. "Power to the PC", *Scientific American*, April 2000, vol. 282 no. 4, p. 27.
- [71] Priest, D. M. "Algorithms for Arbitrary-Precision Floating Point Arithmetic", *Tenth Symposium on Computer Arithmetic*, Los Alamitos, CA, pp. 132-143, 1991.
- [72] Preparata, F., and Shamos, M. I., *Computational Geometry*. Springer-Verlag, New York, 1985.

- [73] Rewienski, M., *Parallel Vector-topological data input for Geographical Informations Systems*, web page (<http://www.epcc.ed.ac.uk/ssp/1996/ss9611.html>), Edinburgh Parallel Computing Centre, Edinburgh, U.K., 1996.
- [74] Robbins, J., "High-Tech Camera Sees What Eye Cannot", in the *New York Times*, September 14, 1999.
- [75] Roytenberg, D., *Developing Parallel GIS Applications on the Alex AVX II Computer*, Master's Thesis, Carleton University, Ottawa, Canada, 1996.
- [76] Raymond, E., Editor, *The Jargon Dictionary*, entry on *Moore's Law*. Web page, http://www.netmeg.net/jargon/terms/m/moore.s_law.html, October 1999.
- [77] Sack, J.-R., Proposal for the Establishment of a Chair, unpublished document, Carleton University, 1995.
- [78] Sack, J.-R., and Urrutia, J., eds.. *Handbook of Computational Geometry*. Elsevier Science, 2000.
- [79] Schneider, M. *Spatial Data Types for Database Systems*. Springer-Verlag, Berlin, 1997. See especially §2.4.
- [80] Shekhar, S., Ravada, S., Turner, G., Chubb, D., and Kumar, V., "Load Balancing in High Performance GIS: Declustering Polygonal Maps", in *Proc. Intl. Symp. on Large Spatial Databases*, pp. 196-215, Springer Verlag Lecture Notes in Computer Science, 1995.
- [81] Shewchuck, J. R., "Adaptive Precision Floating Point Arithmetic and Fast Robust Geometric Predicates", Carnegie-Mellon University Technical Report no. CMU-CS-96-140, 1996.
- [82] Sitaker, K.. ed., *Beowulf mailing list FAQ, version 2*. See <http://www.dnaco.net/~kragen/beowulf-faq.txt>
- [83] Sugihara, K., "An Approach to Error-Free Solid Modeling", in *IMA Summer Program on Robotics*, Inst. Math. and Applications, Univ. of Minnesota, 1987.

- [84] Tsin, Y. H., and Wang, C. A., "An $O(\log n)$ Time Parallel Algorithm for Triangulating a Set of Points in the Plane", in *Information Processing Letters*, vol. 25, pp. 55-60, 1987.
- [85] Valiant, L. G.. "The Bulk-Synchronous Parallel Model," in *Communications of the ACM*, vol. 3, no. 8, pp. 103-111, 1990.
- [86] van Oosterom, P. "An R-tree based map-overlay algorithm." in *Proceedings of the Fifth European Conference and Exhibition on Geographical Information Systems*, pp. 318-327, 1994.
- [87] Wang, F., "A Parallel Intersection Algorithm for Vector Polygon Overlay," in *IEEE Computer Graphics and Applications*, March 1993, pp. 74-81.
- [88] White, D., "A Design for Polygon Overlay", in *Harvard Papers on Geographic Information Systems*, vol. 6., Laboratory for Computer Graphics and Spatial Analysis, Harvard, Cambridge, Mass., pp. D. White/1 - D. White/9, 1978-79.