Two Topics in Applied Algorithmics

By

Patrick R. Morin

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

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

January 1998

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
aceeptance of the thesis,

Two Topics in Applied Algorithmics

submitted by

Patrick R. Morin

Dr. Evangelos Kranakis
(Director, School of Computer Science)

Dr. Jörg-Rüdiger Sack
(Thesis Supervisor)

Carleton University
January 1998

Abstract

This thesis examines two largely unrelated problems in applied algorithmics, motivated by the search for efficient geometric algorithms.

In the first part of the thesis, we consider the problem of finding efficient parallel algorithms for *heterogeneous* parallel computers, i.e., parallel computers in which different processors have different computational potential. To this end, we define a formal computational model for heterogeneous systems and develop algorithms for commonly used communication operations. The result is that many existing parallel algorithms which use these communication operations can be adapted to our model with little or no modifications.

In the second part of the thesis we consider the problem of geometric models which allow for varying levels of detail. To this end, we extend the *progressive mesh* representation introduced by Hoppe. The main technical contribution of this part is an efficient scheme for refining only selected regions of a progressive mesh. Using this scheme, we develop important applications in the fields of computational geometry and geographic information systems.

Both parts of the thesis are supported by experimental results which show that our algorithms are of considerable practical relevance.

Acknowledgements

A number of people have been instrumental in the completion of this thesis, and I would like to acknowledge them here.

While writing this thesis I received funding from the Natural Science and Engineering Research Council of Canada and Almerco. It goes without saying that this was greatly appreciated.

Jörg Sack has proven to be an excellent supervisor. There are many reasons for this, but the most important of these is his dedication to, and willingness to work for, his students.

Anil Maheshwari has provided input on many aspects of this thesis. It is because of his comments and questions that many of the ideas in this thesis are developed to the extent that they are.

While writing parts of this thesis, I was lucky to have been a visitor at the Heinz Nixdorf Institut at the Universät GH-Paderborn. During this time, Ben Juurlink gave comments which significantly improved the presentation of the first part of this thesis.

Finally I would like to thank Silvia Götz for having supported me throughout the writing of this thesis, both as a colleague and close friend. For this I am especially grateful.

Contents

List of Figures

Part I

Coarse Grained Parallel Computing on Heterogeneous Systems

Chapter 1

Introduction

In recent years, parallel computing has been increasing in popularity. Individuals with limited budgets can now build workstation clusters from off-the-shelf processing components and interconnection networks [11, 52]. High speed networks are being used to interconnect traditional supercomputers in order to direct large amounts of computing power at Grand Challenge problems [8]. Even traditional supercomputers usually consist of a very fast workstation host connected to a number of slower in-the-box processors.

The three situations above, which cover nearly all modern parallel computing systems, are all potential examples of *heterogeneous systems*, i.e., systems in which different processors have different computational potential. In the case of workstation clusters, the processing components may be different because the system was grown incrementally and newly added processors are more modern than the originals. The same may be true in the case of supercomputer clusters, or the supercomputers may have even come from different manufacturers. Finally, in the case of traditional supercomputers, it may be beneficial to use the host processor, particularly for sequential portions of computations.

Traditionally, there have been two approaches to dealing with the varying processor speeds in such systems. The first and simplest approach, which we call the *ostrich approach* is to simply ignore the difference in processor speeds and use standard parallel algorithms. In many cases, this leads to the slowest processor becoming

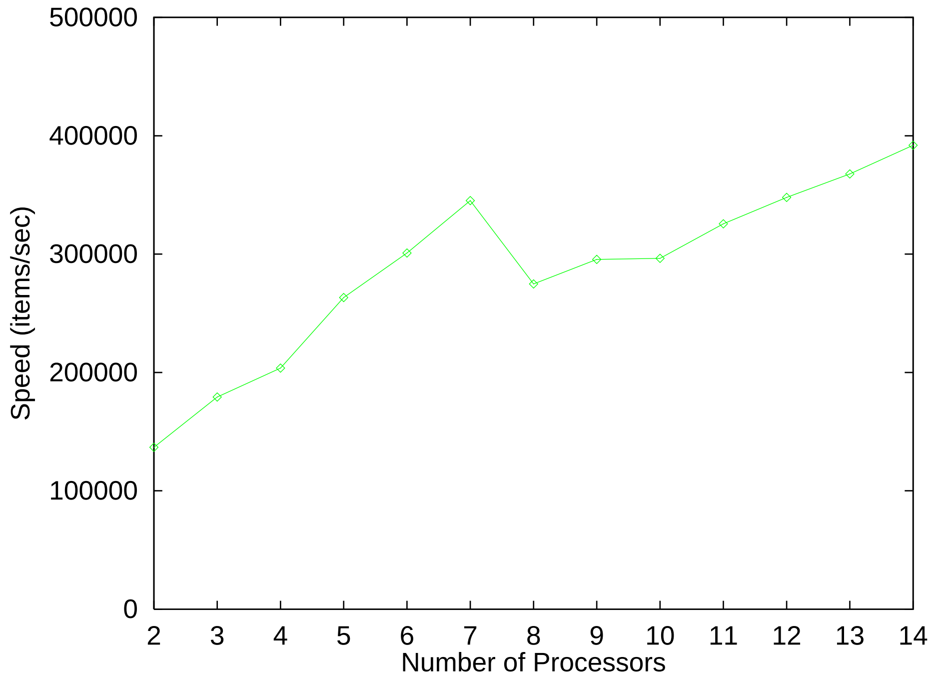

Figure 1.1: Performance of sorting algorithm with slow processors

a bottleneck, and effectively reduces performance to that of a machine in which all processors are equally slow. This can result in decreased performance when slow processors are added to a system.

Figure 1.1 shows an example of a sorting algorithm in which the overall performance of a system decreases with the addition of slow processors. The first seven processors are fast processors, while all other processors are slow processors. Important to note is the decrease in performance when the first slow processor is added to the system.

The second approach, which we call the *overpartitioning approach* is to break the problem into small subproblems, so that there are many more subproblems than processors, and assign subproblems to processors whenever they become idle, either by having a master processor assign all subproblems, or by having processor request subproblems from other processors when they become idle. This approach also has its disadvantages. Decomposing the problem and merging the solutions to subproblems is not always easy, nor is coordinating the processors, and these tasks have an overhead associated with them. Even worse, because of the high latency of communications networks, many processor cycles are wasted waiting for the network to deliver subproblems. In most cases, a healthy dose of performance testing, algorithm analysis, and common sense is required to determine the optimum subproblem size, and this procedure must be repeated when the system configuration changes.

The approach taken in this thesis is to modify fast parallel algorithms which have been shown to be efficient in homogeneous systems to run efficiently on heterogeneous systems. The class of algorithms we choose as our starting point is the class of *coarse grained parallel* (CGP) algorithms. Examples of such algorithms include algorithms for the *bulk synchronous parallel* (BSP) [61], *Coarse Grained Multicomputer* (CGM) [21], and *LogP* [15] models of parallel computation.

In these models a parallel computer is composed of p processors and is being used to solve a problem of size n, where $p \ll n$. The basic communication operation is the h-relation, an all-to-all communication operation in which no processor is the source or destination of more than h words. Algorithms based on these models work in supersteps, where a superstep consists of local computation, followed by global communication (routing an h-relation). The goal of algorithm design is to simultaneously minimize communication and computation.

The heterogeneous networks described above present a problem for standard CGP algorithms, since the slow processors in the network become a bottleneck for the computation. This is due to the fact that CGP algorithms are designed to distribute computation load evenly across processors. However, through careful modifications, these algorithms can be made to distribute computation load according to processor speeds without sacrificing efficiency.

This approach has the obvious advantage over the ostrich approach that it balances the computation according to processor speed and therefore improves performance (Chapter 5 bears this out with empirical evidence). This approach has two advantages over the overpartitioning approach. The first is that it minimizes the effects of latency (the algorithms described in Chapter 5 perform only a constant number of communication operations). The second is that it doesn't require extensive testing and measurements to determine optimum algorithm parameters. In fact, the only parameters used by the algorithms are the processor speeds.

The main contributions of this part of the thesis are the following:

1. The definition of a parallel computation model called the *heterogeneous coarse grained multicomputer* (HCGM) which takes into account varying processor speeds— The model is simple enough to be easy to use, accurate enough to allow for the development of efficient algorithms, and portable enough to allow these algorithms to run efficiently on a wide variety of parallel architectures.

2. The identification of a number of communication patterns most commonly used in CGP algorithms and efficient HCGM algorithms for their implementation— These algorithms form the basis for translating existing CGP algorithms into HCGM algorithms.

3. A number of algorithms for the HCGM model—These algorithms are arrived at by describing existing CGM and BSP algorithms in terms of the previously mentioned communication patterns.

4. An implementation of these ideas—The implementation consists of a library of the previously mentioned communication patterns and some algorithms.

1.1 Comparisons with Related Work

In order to differentiate this work from other research on heterogeneous parallel computing, this section compares and contrasts this work with previous work in this area.

The topic of data partitioning in heterogeneous systems with simple fixed communication patterns is addressed in [14, 51], and semi-automatic methods of choosing the best partitioning scheme and parameters are described. Methods for the compile time scheduling of various types of parallel loops are described in [12]. The results in this thesis go beyond these in that the problems addressed have much less structure than simple stenciling operations on 2D grids or uniform parallel loops whose communication patterns can be analyzed at compile time. In Chapter 5 algorithms are presented for sorting, median finding, and a number of computational geometry problems.

Methods for dynamic load balancing such as those described in [54, 45, 64] can also be applied to heterogeneous systems. All these methods fall into the overpatitioning strategy category. The advantages of our strategy over such overpartitioning strategies have been described above. These are the minimization of the effects latency and simplicity of the algorithm parameters.

In [66] a mathematical model of a network of workstations is described. In [65], the authors describe a stochastic performance prediction methodology for this model based on the task graph of the parallel application. Although this model is an accurate predictor of performance, it is not clear that the model leads to the development of efficient algorithms. In fact, in the matrix multiplication tests described in [65], a 12 processor configuration actually performs worse than a 2 processor configuration.

The difference between the model in [66, 65] and the HCGM model is that the HCGM model is not intended to predict exact running times of parallel algorithms on parallel machines. Rather, it is designed to distinguish between "good" and "bad" algorithms, i.e., if the model says that algorithm $\mathcal{A}$ is better than algorithm $\mathcal{B}$, then $\mathcal{A}$ should perform better than $\mathcal{B}$ when implemented. This makes the HCGM model simpler, which in turn leads to a much simpler algorithms analysis procedure.

Chapter 2

Coarse Grained Parallel Computing Models

In this chapter, two coarse grained parallel computing models are described. However, before beginning with a description of these models we suggest some criteria which any coarse parallel computing model should strive for.

1. *Simplicity.* Any model should be simple enough that analyses under the model are simple to derive and understand.

2. *Accuracy.* The model should be an accurate reflection of real life in that algorithms which are efficient according to the model should perform well when implemented on actual systems.

3. *Portability.* The model should represent a wide range of parallel architectures.

Clearly, some of these goals are in conflict with each other. On one side is the goal of accuracy which seemingly requires a complex parameterized model, and on the other side is the goal of simplicity which requires a model with few parameters. The goal of portability is orthogonal to these two in that it can be achieved by a simple very abstract model through the model's abstractness, or by a complex highly parameterized model through the sheer number of parameters. The two models

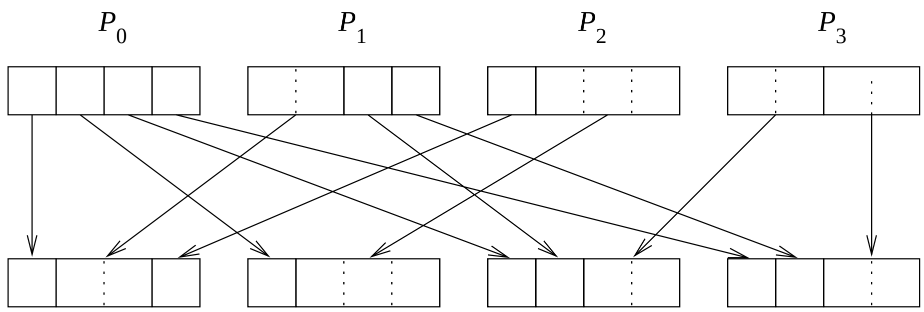

Figure 2.1: An example h-relation with $h = 4$ and $p = 4$.

presented in this section represent a tradeoff between these two sides which achieve these goals to varying degrees.

2.1 Preliminaries

The two models presented in this chapter have some features and terminology in common. We take the time to introduce these here. In both these models computation proceeds in *supersteps* consisting entirely of local computation, or entirely of global communication. These are respectively referred to as computation supersteps and communication supersteps. A parallel computation interleaves the two types of supersteps, and all processors must have completed the previous computation superstep before a communication superstep can proceed.

The complexity of a computation superstep is measured by the maximum amount of computation done by any processor during the superstep. Thus the "slowest"[1] processor dominates the computation time of the superstep.

The only communication operation supported by these two models is the *h-relation*, an all-to-all communication operation in which each processor sends at most h words of data and receives at most h words of data. An example h-relation is illustrated in Figure 2.1.

When discussing the performance of algorithms, we will often make use of a *coarse*

[1] In this case "slowest" means the processor which takes the longest to complete its computation.

grained assumption, $n \gg p$, i.e., the size of the problem is significantly larger than the number of processors. One possible interpretation of "significantly larger" is $p \leq \frac{n}{p}$. This leads to the following implications which can be arrived at through simple algebraic manipulations.

$$\begin{array}{rl} & p \leq \frac{n}{p} \\ \Leftrightarrow & p^2 \leq n \\ \Leftrightarrow & p \leq \sqrt{n} \\ \Rightarrow & \sqrt{n} \leq \frac{n}{p} \end{array}$$

The main advantage of the coarse grained assumption is that it is often possible to take a problem of size n, "compress" it to produce a problem of size p, and use the solution of compressed problem to solve the original problem. The coarse grained assumption helps here since the problem of size p is small enough to be solved sequentially on a single processor. This idea will become more concrete in Section 2.4 where it will be applied to the problem of computing the prefix sum of n elements.

2.2 The Bulk Synchronous Parallel Model

The *bulk synchronous parallel (*BSP*)* computer, defined by Valiant in [61] consists of a set of identical processing elements, each with their own (non-shared) local memory, interconnected by a communication network capable of routing an h-relation for any value of h. A BSP computer is characterized by three parameters: p, the number of processors, g, the ratio of processor speed to the bandwidth of the communication network, and L, the minimum time between computation supersteps.

The g and L parameters are used to measure the cost of communication of a BSP algorithm. The g parameter penalizes algorithms for their use of communication bandwidth. The L parameter represents network latency, and penalizes algorithms for the number of communication operations used. Routing a single h-relation on a BSP computer incurs a communication cost of $O(gh + L)$.

The performance of a BSP algorithm is given by its communication and computation costs. As an example, consider a BSP algorithm which has λ communication and computation supersteps. Assume that each communication superstep consists of an h-relationwith $h = \frac{n}{p}$, and that each computation superstep uses $O(\frac{n}{p})$ local computation per processors. Then the running time of the algorithm is given by $O(\frac{\lambda n}{p} + \frac{\lambda g n}{p} + \lambda L)$. The first term in this expression represents the computation time, while the second two represent the communication time.

2.3 The Coarse Grained Multicomputer Model

The coarse grained multicomputer (CGM) model was introduced by Dehne *et. al.* [21], and a number of algorithms have been defined with respect to this model [9, 10, 20, 21, 22, 23]. A *coarse grained multicomputer*, CGM(m, p), consists of p identical processors, labelled $P_0, \ldots, P_{p-1}$, each with $\Theta(\frac{m}{p})$ local random access memory. These processors are interconnected by a communication network capable of routing an h-relation with $h = O(\frac{m}{p})$.

The performance of a CGM algorithm is measured in terms of the amount of local computation performed and the number of supersteps. Both of these quantities can be functions of n and p.

The following observation (a version of which appears in [10]) relates CGM algorithms to BSP algorithms:

Observation 1. *Any* CGM(m, p) *algorithm which uses* $O(\lambda)$ *supersteps and* $O(T(n, p))$ *computation time is also a* BSP *algorithm with running time* $O(T(n, p) + \frac{g\lambda m}{p} + \lambda L)$.

In the CGM model, algorithms are classified based on λ, the number of supersteps in the algorithm. The three classes, in order of desirability, are as follows:

1. $\lambda = O(1)$
2. $\lambda = O(f(p))$, with performance degrading as f increases.
3. $\lambda = O(f(n))$, with performance degrading as f increases.

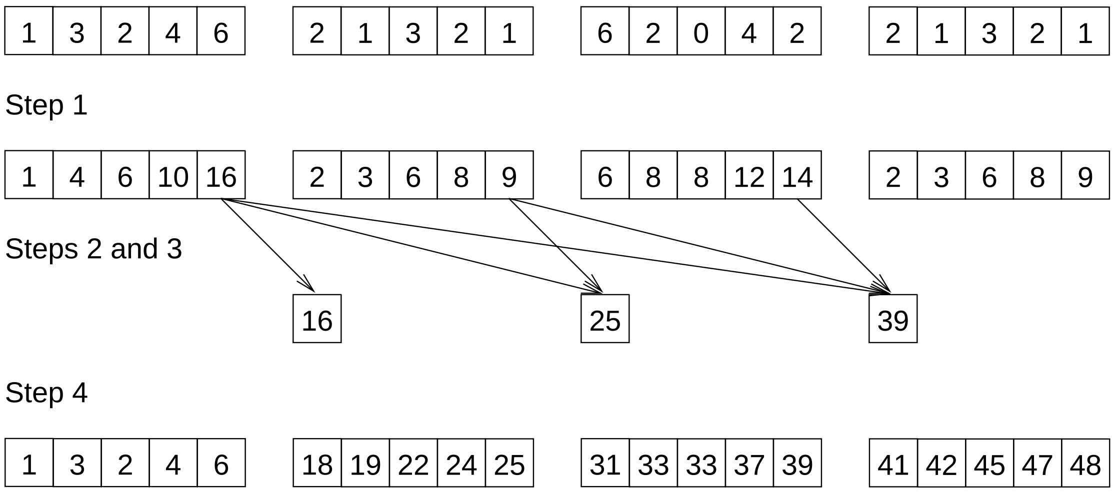

Figure 2.2: Computing the prefix sum of 20 elements using 4 processors.

2.4 A Simple Example: Prefix Sum

This section describes a coarse grained parallel algorithm for one of the most common operations in parallel algorithms: computing the prefix sum of n elements. The algorithm is analyzed both in the CGM and BSP models.

The prefix sum of n elements $x_0, x_1, \ldots, x_{n-1}$ is defined as the sequence $s_0, \ldots, s_{n-1}$, where $s_i = x_0 \oplus x_1 \oplus \cdots \oplus x_i$ and $\oplus$ is any associative operator. One method of computing the prefix sum of n elements on a coarse grained parallel computer is to first compute locally the prefix sum of the elements stored at each processor and then merge these values with other processors using the standard EREW-PRAM prefix sum algorithm (see e.g. [41]). Using this approach, one can devise an algorithm that uses $O(\log p)$ supersteps and $O(\frac{n}{p})$ local computation per processor.

However, a more efficient algorithm can be obtained for a restricted range of parameter by using a coarse grained assumption. The algorithm proceeds as follows, and an example of its execution is given in Figure 2.2.

Prefix-Sum()

1. Each processor locally computes the prefix sum of its $\frac{n}{p}$ input elements.
2. Each processor, P_i, broadcasts the total sum of its input elements to all P_j, where $i < j \le p-1$, in a single communication superstep.
3. Each processor, P_i, computes the sum of the (at most $p-1$) input elements received in Step 2.
4. Each processor computes its final portion of the prefix sum by adding the value computed in Step 3 to each of the values computed in Step 1.

The analysis of this algorithm under the CGM(n,p) model is quite simple. Steps 1 and 4 use $O(\frac{n}{p})$ computation time. Step 2 can be implemented as an h-relation with $h = p \in O(\frac{n}{p})$. Step 3 uses $O(p) \subseteq O(\frac{n}{p})$ computation time. Therefore the algorithm uses $O(\frac{n}{p})$ computation time and $O(1)$ supersteps on a CGM(n,p).

By applying Observation 1 one can derive a BSP analysis of this algorithm of $O(\frac{n}{p} + \frac{gn}{p} + L)$. Although this analysis is correct, it is not as tight as possible. A tighter analysis can be obtained as follows: Step 1, Step 3, and Step 4, use $O(\frac{n}{p})$ local computation per processor. Step 2 is an h-relation with $h = p$, and therefore takes $O(gp+L)$ time, yielding an overall running time of $O(\frac{n}{p}+gp+L)$ for the entire algorithm.

These analyses point out the relative advantages of these two models. The main advantage of the CGM model is its simplicity; the expressions derived for running times of algorithms are simple to derive and understand. Analyses under the BSP model on the other hand, are usually more complicated. Often BSP algorithms have restrictions on g and L as well as n and p, and which make these algorithms less portable than CGM algorithms. The main advantage of the BSP model is that it rewards algorithms which are bandwidth efficient via the g parameter, whereas the CGM model does not, making the BSP model more accurate, and possibly leading to algorithms which make more efficient use of bandwidth.

From the point of view of machine architectures, the CGM model simplifies the BSP parameters by working under the assumption that the latency, L, is very high and the ratio of processor speed to bandwidth, g, is very low. Under this assumption, the communication cost is dominated by the latency, and counting supersteps is a reasonable method of determining communication cost.

In the remainder of this thesis, the CGM model will be used as a starting point for a heterogeneous coarse grained parallel computing model, and the resulting model will be used to analyze all algorithms presented. This decision was made in order to simplify the analysis of the algorithms presented so that the discussion can focus on the issues associated with heterogeneous systems rather than on the details of analyzing algorithms in the BSP model. It is worthwhile noting, however, that the modifications introduced to the CGM model to arrive at the heterogeneous CGM model could also be introduced into the BSP model, and the algorithms presented for the heterogeneous CGM model are also algorithms for this heterogeneous BSP model.

Chapter 3

A Heterogeneous Computing Model

This chapter discusses a generalization of the CGM model described in Chapter 2 which takes the presence of heterogeneous processors into account. This model, called the heterogeneous coarse grained multicomputer (HCGM) model maintains the simplicity of the CGM model while providing a means of modelling the effects of heterogeneous processors such as the anomaly described in Chapter 1.

3.1 The Heterogeneous Coarse Grained Multicomputer

A *heterogeneous coarse grained multicomputer* HCGM(m, p, s) consists of p possibly heterogeneous processors labelled $P_0, \ldots, P_{p-1}$. The value $s = \sum_{i=0}^{p-1} s_i$ represents the total speed of the parallel machine, where s_i represents the speed of P_i and is an integer. Each processor, P_i, can perform w units of work in $\frac{w}{s_i}$ time units. Each processor knows the values of $s_0, \ldots, s_{p-1}$ as well as the value of s.

For conciseness, we define $s^{max} = \max\{s_i : 0 \leq i \leq p-1\}$ and $s^{min} = \min\{s_i : 0 \leq i \leq p-1\}$, i.e., s^{max} and s^{min} are the speeds of the fastest and slowest processors, respectively. Similarly, we define $P^{max} = P_{\min\{i : s_i = s^{max}\}}$ and $P^{min} = P_{\min\{i : s_i = s^{min}\}}$.

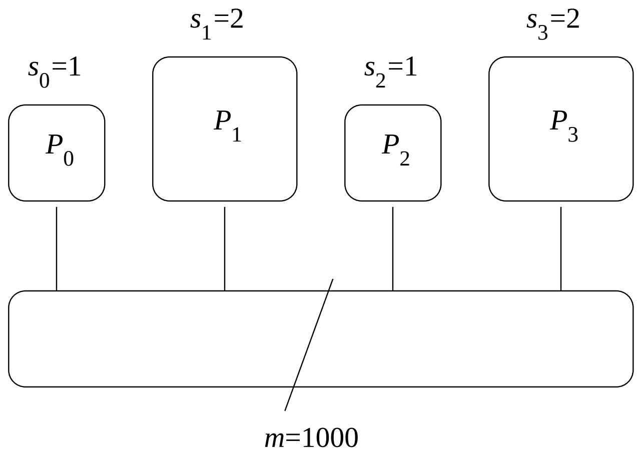

Figure 3.1: An example of an HCGM(m, p, s) with $m = 1000$, $p = 4$, $s = 6$, $P^{max} = P_1$, $s^{max} = 2$, and $P^{min} = P_0$, and $s^{min} = 1$.

That is, P^{max} is a representative fastest processor, and P^{min} is a representative slowest processor. An example of an HCGM(m, p, s) is shown in Figure 3.1.

Each processor, P_i, in an HCGM(m, p, s) has $\Theta(\max\{\frac{m}{p}, \frac{s_i}{s}m\})$ local memory. The p processors of an HCGM(m, p, s) are interconnected by a network capable of routing any all-to-all communication in which the total amount of data exchanged is $O(m)$. However, these communication operations incur a penalty in computation time. If P_i is the source (resp. destination) of b words of information, then P_i incurs a penalty in computation time of $\frac{b}{s_i}$. This represents the local computation needed to pack (resp. unpack) messages into (resp. from) buffers. For example, the computation time associated with routing an h-relation is $\max\{\frac{h}{s_i} : 0 \leq i \leq p-1\} = \frac{h}{s^{min}}$.

Because the memory and communication speeds of the processors are proportional to the processor speeds, HCGMalgorithms can take advantage of faster processors by having them process and communicate more data. If, as in the CGM(m, p) model, each processor has only $O(\frac{m}{p})$ memory, it may not be possible to improve the performance of algorithms and avoid the anomaly described in Chapter 1. Such is the case when the problem size, n, is equal to the size of the total memory, m.

We assume that the input to an HCGM(m, p, s) algorithm is initially distributed in a load balanced manner, that is, each P_i initially holds $\frac{s_i}{s}n$ input elements. At this point we note that the HCGM(m, p, s) model is equivalent to the CGM(m, p) model when $s_0 = s_1 = \cdots = s_{p-1} = 1$.

Like a CGM algorithm, the performance of an HCGM algorithm is measured in terms of computation time and the number of supersteps. Both of these quantities can be functions of n, p, s, and $s_0, \ldots, s_{p-1}$. However, rather than measuring the amount of local computation, w, it is the local computation *time*, $\frac{w}{s_i}$, that is measured. Ideally, an HCGM(m, p, s) algorithm gives a speedup of s when compared to a uniprocessor machine with unit speed running the fastest sequential algorithm for the same problem. If at all possible, this speedup should be independent of the values of $s_0, \ldots, s_{p-1}$.

One possible approach of obtaining HCGM algorithms directly from BSP and CGM algorithms is to have each processor, P_i, simulate $s_i / \gcd(s_0, \ldots, s_{p-1})$ virtual CGM processors, where $\gcd(s_0, \ldots, s_{p-1})$ is the greatest common divisor of $s_0, \ldots, s_{p-1}$. There are at least three problems with this approach.

1. The overheads associated with automatically simulating virtual processors can have a significant negative impact on real running times. These overheads can be avoided by having implementors code the simulation by hand, but this adds complexity to the already difficult task of implementing parallel algorithms.

2. In some cases the number of supersteps in a CGM algorithm is a function of the number of processors, so increasing the number of processors by creating virtual processors increases the number of supersteps.

3. Many coarse grained parallel algorithms work on some version of the coarse grained assumption (see Section 2.1), and introducing virtual processors may violate this assumption.

3.2 A Simple Example: Prefix Sum

Next, we consider the problem of computing the prefix sum of n elements, for which a CGM algorithm was presented in Chapter 2. The section begins by analyzing the algorithm presented in Chapter 2 under the HCGM(n, p, s) model. Following this (disappointing) analysis, a new algorithm is developed especially for the HCGM(n, p, s).

Recall the prefix sum algorithm of Chapter 2. Step 1 of the algorithm, computing the prefix sum locally takes $O(\frac{n}{s})$ computation time since each processor P_i initially contains $\frac{s_i}{s}n$ elements. In Step 2 of the algorithm, each processor P_i sends $p - i - 1$ elements and receives $i-1$ elements. This takes $O(p)$ work per processor, and therefore takes $O(\max\{\frac{p}{s_i} : 0 \leq i \leq p-1\}) = O(\frac{p}{s^{min}})$ time. Step 3 takes the same amount of local computation time as Step 2, and Step 4 takes the same amount of local computation time as Step 1. Thus the algorithm uses $O(\frac{p}{s^{min}} + \frac{n}{s})$ computation time and $O(1)$ supersteps on a HCGM(n, p, s).

This running time can be improved by having P^{max} do the bulk of the work in Step 3. This leads to the following algorithm.

HCGM-PREFIX-SUM()

1. Each processor locally computes the prefix sum of its $\frac{s_i}{s}n$ input elements.
2. Each processor, P_i, sends the total sum of its input elements to P^{max} in a single communication superstep.
3. P^{max} computes the prefix sum of the p elements received in Step 2.
4. For $1 \leq i \leq p-1$, P^{max} sends the $(i-1)$st element computed in Step 3 to P_i in a single communication superstep.
5. Each processor computes its final portion of the prefix sum by adding the value received in Step 4 to each of the values computed in Step 1.

This algorithm provides our first result for the HCGM model.

Theorem 1. *The prefix sum of n elements can be computed on an* HCGM(n, p, s), *using $O(\frac{n}{s})$ computation time and $O(1)$ supersteps, provided that $\frac{n}{p} \geq p$.*

Proof. In Step 1 and Step 5, each processor P_i does $O(\frac{s_i}{s}n)$ work and this work can be done in $O(\frac{n}{s})$ time. Step 2 and Step 4 each processor uses $O(1)$ computation time except P^{max} which uses $O(\frac{p}{s^{max}})$ computation time. Step 3 takes $O(\frac{p}{s^{max}})$ computation time. Since $p \leq \frac{n}{p}$, and by pigeonhole principle $s^{max} \geq \frac{s}{p}$, all steps can be performed in $O(\frac{n}{s})$ time. □

In [25], Ferreira and Ubéda describe an algorithm for computing the medial axis transform using 8 prefix sum operations and $O(\frac{n}{p})$ local computation on a CGM(n, p). By using the HCGM(n, p, s) version of the prefix sum algorithm described above, this algorithm can be adapted to an HCGM(n, p, s), yielding the following result.

Corollary 1. *The medial axis transform of a $\sqrt{n} \times \sqrt{n}$ image can be computed using $O(\frac{n}{s})$ computation time and $O(1)$ supersteps on a* HCGM(n, p, s) *with $p \leq \frac{n}{p}$.*

Corollary 1 is the first example of a technique which will be used repeatedly in this part of the thesis. Namely, to obtain HCGM algorithms from CGM algorithms, one need only find HCGM algorithms for the primitive communication operations performed by the CGM algorithm. When these operations are replaced in the CGM algorithm by their HCGM counterparts, the resulting algorithm is an HCGM algorithm.

Chapter 4

Communication Patterns

This chapter discusses common communication patterns used in coarse grained parallel algorithms and gives their implementations both in the CGM and HCGM models. The motivation for this is that by implementing HCGM versions of these patterns, we obtain a number of HCGM algorithms directly from CGM algorithms which use these patterns.

This work also has applications outside of heterogeneous parallel computing. As is well known in the field of software engineering, the study of software patterns is a field in itself (see e.g., [28]). By identifying common patterns used in coarse grained parallel algorithms we provide a good starting point for the development of libraries and frameworks supporting the implementation of such algorithms.

4.1 Random-Sample

The technique of random sampling is one of the most useful tools used in the design of randomized parallel algorithms. In random sampling, a random subset of size $O(r)$ is chosen from the n input values.

In the context of coarse grained parallel computing, random sampling involves choosing $O(r)$ samples from the input and routing them to a designated processor, usually P_0. This processor then typically performs some computation on these elements and broadcasts the results of this computation to all processors. The algorithm for the Random-Sample pattern proceeds as follows:

CGM-RANDOM-SAMPLE(r)

1. Each processor, P_i, tosses a biased coin with success probability $\frac{r}{n}$ for each of its input elements.

2. The elements for which the coin toss was successful are routed to P_0.

Using Chernoff bounds (see Appendix A), it is easily shown (see, e.g., [2]) that the number of elements which arrive at P_0 is $\tilde{O}(r)$.[1] Thus, if $r \in O(\frac{n}{p})$, the computation time used by Random-Sample pattern is $\tilde{O}(\frac{n}{p})$ and the number of supersteps is $\tilde{O}(1)$ on a CGM(n, p).

In order to modify the random sample pattern for the HCGM(n, p, s) model we need only change the designated processor to which the sample are routed. Rather than routing the samples to P_0, we route the samples to P^{max}. This results in the following implementation:

HCGM-RANDOM-SAMPLE(r)

1. Each processor, P_i, tosses a biased coin with success probability $\frac{r}{n}$ for each of its input elements.

2. The elements for which the coin toss was successful are routed to P^{max}.

Theorem 2. *The* HCGM-RANDOM-SAMPLE(r) *algorithm uses $\tilde{O}(\frac{n}{s})$ computation time and $\tilde{O}(1)$ supersteps on an* HCGM(n, p, s), *provided that $\frac{s^{max}}{s}n \geq r \geq 3\ln n$.*

Proof. In Step 1, each processor, P_i must perform $\frac{s_i}{s}n$ coin tosses and can do these in $O(\frac{n}{s})$ time. In Step 2 each processor, P_i sends at most $\frac{s_i}{s}n$ elements and can do this in $O(\frac{n}{s})$ time. Let r' be the number of elements received by P^{max} in Step 2. Then r' is a random variable following the binomial distribution $b(n, \frac{r}{n})$. Applying Theorem 15,

[1] See Appendix A for a definition of the $\tilde{O}$ notation.

Equation A.1 we get that

$$\begin{aligned}\Pr\left[r' \geq cr\right] &= \left(\frac{1}{e}\right)^{(c-1)^2 r/3} \\ &\leq \frac{1}{n^{(c-1)^2}}, \text{ for } r \geq 3\ln n.\end{aligned}$$

Therefore, during Step 2 P^{max} receives $\tilde{O}(r)$ elements and can do this in $\tilde{O}(\frac{r}{s^{max}}) \subseteq \tilde{O}(\frac{n}{s})$ time. □

4.2 Random-Assign

Random assignment is a tool used in a number of CGP algorithms to achieve load balancing. The idea behind random assignment is to assign elements of the input to processors in a random fashion. In this way, if the work performed on each element is variable, then one expects that all processors will be assigned roughly the same amount of work. This idea is realized in the following procedure.

CGM-Random-Assign()

1. Each processor, P_i, randomly assigns each of its elements to one of p buckets, $b_{i,0}, \ldots, b_{i,p-1}$ with equal probability.

2. Each processor, P_i, routes the contents of each bucket, $b_{i,j}$, to P_j.

Once again, using Chernoff bounds it is not difficult to show that the number of elements routed to any processor is $\tilde{O}(\frac{n}{p})$ (see, e.g., [2]). Thus, the algorithm uses $\tilde{O}(\frac{n}{p})$ computation time and $\tilde{O}(1)$ communication rounds.

The HCGM version of the Random-Assign pattern is similar to the CGM version except that each processor, P_i, should receive $O(\frac{s_i}{s}n)$ elements in Step 2. To achieve this, we change the probability to which elements are assigned to buckets in Step 1. The modified algorithm works as follows.

HCGM-RANDOM-ASSIGN()

1. Each processor, P_i, randomly assigns each of its elements to one of p buckets, $b_{i,0}, \ldots, b_{i,p-1}$. P_i assigns an element to bucket $b_{i,j}$ with probability $\frac{s_j}{s}$.

2. Each processor, P_i, routes the contents of each bucket, $b_{i,j}$, to P_j.

Theorem 3. *The* HCGM-RANDOM-ASSIGN() *algorithm uses $\tilde{O}(\frac{n}{s}\log p)$ computation time and $O(1)$ supersteps on an* HCGM(n, p, s)*, with $p \leq n$ and $\frac{s^{min}}{s}n \geq 3\ln n$.*

Proof. Step 1 can be accomplished by performing a binary search on the p buckets for each of the input elements, and can therefore be done in $O(\frac{n}{s}\log p)$ time.

Next we show that the number of elements received in Step 2 is $\tilde{O}(\frac{s_i}{s}n)$ for each processor, P_i. Let n_i be the number of elements received by P_i in Step 2. Then clearly n_i is a random variable which follows the binomial distribution $b(\frac{s_i}{s}, n)$. Applying Theorem 15, Equation A.1, we get

$$\begin{aligned}\Pr\left[n_i \geq c\frac{s_i}{s}n\right] &\leq \left(\frac{1}{e}\right)^{(c-1)^2\frac{s_i}{s}n/3} \\ &\leq \frac{1}{n^{(c-1)^2}}, \text{ for } \tfrac{s_i}{s}n \geq 3\ln n.\end{aligned}$$

Therefore, the probability that *any* processor, P_i, receives more than $c\frac{s_i}{s}n$ elements is bounded by

$$\begin{aligned}\Pr\left[\exists i \text{ s.t. } n_i \geq c\frac{s_i}{s}n\right] &\leq \frac{p}{n^{(c-1)^2}} \\ &\leq \frac{1}{n^{(c-1)^2-1}}, \text{ for } p \leq n.\end{aligned}$$

Therefore Step 2 can be done using $\tilde{O}(\frac{n}{s})$ computation time and the entire algorithm uses $\tilde{O}(\frac{n}{s}\log p)$ computation time. □

4.3 Linear-Partition

Let S be a be a set of keys and $\leq$ be a relation that defines a partial order on S. A linear partition of S is a partitioning of S into p disjoint subsets $S_0, \ldots, S_{p-1}$ such that $x \leq y$ for all $x \in S_i$, $y \in S_j$ and $i < j$. Linear partitioning is one of the most commonly used communication patterns in parallel computing. This is due simply to the fact that sorting is a special case of linear partitioning in which the keys are sorted locally after being partitioned.

Here we describe a randomized linear partitioning algorithm based on the sample sort algorithm described in [32]. We assume that the n keys are all distinct since if they are not, they can be made so by, e.g., concatenating their value with their processor number and memory location. The algorithm proceeds as follows.

CGM-LINEAR-PARTITION()

1. All processors take a random sample of size $O(r)$, r a constant, using the HCGM-Random-Sample algorithm and route the sample keys to P_0.
2. P_0 sorts the sample keys. Denote these keys by $sample_0, \ldots, sample_{pr-1}$ where $sample_i$ is the sample with rank i in the sorted order.
3. P_0 defines $p+1$ splitters, $splitter_0, \ldots, splitter_p$, where

$$splitter_i = \begin{cases} -\infty & \text{if } i = 0 \\ sample_{ir} & \text{if } 0 < i < p \\ \infty & \text{if } i = p \end{cases}$$

4. P_0 broadcasts $splitter_0, \ldots, splitter_p$ to all processors.
5. Each processor, P_i, places each of its keys into one of p buckets, where a key x is placed in bucket b_{ij} if and only if $splitter_j \leq x < splitter_{j+1}$.
6. Each processor, P_i, routes the contents of bucket b_{ij} to P_j for all i, j.

That this algorithm produces a valid linear partition is clear since (1) all keys are assigned to exactly one bucket, and hence one processor, (2) all keys in bucket i are strictly less than all keys in bucket j for all $i < j$. Less clear is the running time of the algorithm, since it is conceivable that some processor receives significantly more than $O(\frac{n}{p})$ elements in Step 6. Gerbessiotis and Valiant [32] showed that for properly chosen values of s, n, and p such a situation does not occur.

When adapting this algorithm to the HCGM model, we change the way in which the splitters are chosen. In order to balance the work according to $s_0, \ldots, s_{p-1}$ it is necessary that $O(\frac{s_i}{s}n)$ *input keys* fall between $splitter_i$ and $splitter_{i+1}$. In order to achieve this, we choose the splitters so that $O(\frac{s_i}{s}r)$ *sample keys* fall between $splitter_i$ and $splitter_{i+1}$. This leads to the following algorithm.

HCGM-LINEAR-PARTITION()

1. All processors take a random sample of size $O(r)$, r to be defined later, using the HCGM-Random-Sample algorithm and route the sample keys to P^{max}.

2. P^{max} sorts the sample keys. Denote these keys by $sample_0, \ldots, sample_{pr-1}$ where $sample_i$ is the sample with rank i in the sorted order.

3. P^{max} defines $p+1$ splitters, $splitter_0, \ldots, splitter_p$, where

$$splitter_i = \begin{cases} -\infty & \text{if } i = 0 \\ sample_{\left\lceil \left(\sum_{j=0}^{i} \frac{s_j}{s}\right) r \right\rceil} & \text{if } 0 < i < p \\ \infty & \text{if } i = p \end{cases}$$

4. P^{max} broadcasts $splitter_0, \ldots, splitter_p$ to all processors.

5. Each processor, P_i, places each of its keys into one of p buckets, where a key x is placed in bucket b_{ij} if and only if $splitter_j \leq x < splitter_{j+1}$.

6. Each processor, P_i, routes the contents of bucket b_{ij} to P_j for all i, j.

Theorem 4. *The* HCGM-LINEAR-PARTITION() *algorithm uses* $\tilde{O}(\frac{n}{s}\log p)$ *computation time and* $O(1)$ *supersteps on an* HCGM(n, p, s)*, with* $r = \frac{s^{max}}{s}n$*,* $\frac{s^{min}}{s}r \geq 2\ln n$*, and* $p \leq n$*.*

Proof. By Theorem 2, Step 1 of the algorithm uses $\tilde{O}(\frac{n}{s})$ computation time. Step 2 of the algorithm uses $\tilde{O}(\frac{r}{s^{max}} \log r) \subseteq \tilde{O}(\frac{n}{s} \log n)$ computation time. Steps 3 and 4 clearly use $\tilde{O}(\frac{n}{s})$ computation time. Step 5 uses $O(\frac{n}{s} \log p)$ computation time.

Next we consider the possibility that some processor receives too many keys in Step 6. Let n_i be the number of keys received by P_i in Step 6. Let r_i be the number of samples chosen from the $c\frac{s_i}{s}n$ keys following $splitter_i$ in the overall sorted order. Now note that $n_i > c\frac{s_i}{s}n$ only if $r_i < \frac{s_i}{s}r$. That is, the number of keys between $splitter_i$ and $splitter_{i+1}$ can only exceed $c\frac{s_i}{s}n$ if less than $\frac{s_i}{s}r$ samples are chosen from these keys.

Since r_i is a random variable that follows the binomial distribution $b(c\frac{s_i}{s}n, \frac{r}{n})$, Theorem 15, Equation A.2 can be applied to get

$$\begin{aligned}
\Pr\left[q_i \leq \frac{s_i}{s}r\right] &\leq \left(\frac{1}{e}\right)^{(1-\frac{1}{c})^2(\frac{r}{n})(c\frac{s_i}{s}n)/2} \\
&= \left(\frac{1}{e}\right)^{(1-\frac{1}{c})^2 c(\frac{s_i}{s}r)/2} \\
&= \frac{1}{n^{(1-\frac{1}{c})^2 c}}, \text{ for } \tfrac{s_i}{s}r \geq 2\ln n.
\end{aligned}$$

Therefore, the probability that *any* processor, P_i, receives more than $c\frac{s_i}{s}n$ elements is bounded by

$$\begin{aligned}
\Pr\left[\exists i \text{ s.t. } q_i \leq \frac{s_i}{s}r\right] &\leq \frac{p}{n^{(1-\frac{1}{c})^2 c}} \\
&\leq \frac{1}{n^{(1-\frac{1}{c})^2 c-1}}, \text{ for } p \leq n,
\end{aligned}$$

and Step 6 can be done using $\tilde{O}(\frac{n}{s})$ computation time. □

Since sorting is a special case of Linear-Partition in which elements are sorted locally after partitioning, we obtain the following corollary.

Corollary 2. *Sorting n keys can be done using $\tilde{O}(\frac{n}{s} \log n)$ computation time and $O(1)$ supersteps on an* HCGM(n, p, s), *with $r = \frac{s^{max}}{s}n$, $\frac{s^{min}}{s}r \geq 2\ln n$, and $p \leq n$.*

4.4 PRAM-Simulation

PRAM simulations on the BSP model were first introduced by Valiant in [61], and by Gerbessiotis and Valiant in [32] as a means of obtaining BSP algorithms from PRAM algorithms , and it was shown that if the BSP parameter g is close to unity, the resulting BSP algorithms would be optimal. Unfortunately, this condition is not usually met in practice and the performance of the resulting algorithms is often disappointing. More recently, PRAM algorithms have been revived in the form of *clipping* [10]. Clipping involves simulating a PRAM algorithm for $O(\log p)$ rounds, stopping the algorithm (clipping it), and completing the computation with a specialized CGP algorithm.

In doing PRAM simulations on a CGM, each processor simulates $\frac{n}{p}$ EREW-PRAM processors and stores $\frac{n}{p}$ data elements. Each round of the simulation consists of a read phase and a write phase. The following algorithm performs 1 step of an EREW-PRAM simulation on a CGM(n, p):

CGM-PRAM-SIMULATION()

1. Each processor, P_i, formulates $O(\frac{n}{p})$ read requests and sends each request to the processor holding the element to be read.
2. Each processor, P_i, responds to the $O(\frac{n}{p})$ read requests received in Step 1.
3. Each processor, P_i, formulates $O(\frac{n}{p})$ write requests and sends each request to the processor holding the element to be written.
4. Each processor, P_i, responds to the $O(\frac{n}{p})$ write requests received in Step 3.

Theorem 5. *The* CGM-PRAM-SIMULATION() *algorithm uses* $O(\frac{n}{p})$ *computation time and* $O(1)$ *supersteps on a* CGM(n, p).

Proof. Determining which processor, P_i, services a read or write request for memory location j can be done in constant time using the formula $i = \lfloor j/(n/p) \rfloor$. Therefore Step 1 and Step 3 take $O(\frac{n}{p})$ time. Since an EREW-PRAM is being simulated, no processor receives more than $O(\frac{n}{p})$ requests in Step 2 and Step 4. Therefore these steps can be done in $O(\frac{n}{p})$ time, yielding the stated time bound. □

The HCGM(n, p, s) version of the PRAM-SIMULATION procedure is nearly identical to the CGM version, though it only holds for a restricted range of parameters. The algorithm proceeds as follows.

HCGM-PRAM-SIMULATION()

1. Each processor, P_i, formulates $O(\frac{s_i}{s}n)$ read requests and sends each request to the processor holding the element to be read.

2. Each processor, P_i, responds to the $O(\frac{s_i}{s}n)$ read requests received in Step 1.

3. Each processor, P_i, formulates $O(\frac{s_i}{s}n)$ write requests and sends each request to the processor holding the element to be written.

4. Each processor, P_i, responds to the $O(\frac{s_i}{s}n)$ write requests received in Step 3.

The extra restriction on the range of parameters comes from the fact that the processor, P_i, which corresponds to PRAM memory location j can not be determined using a simple formula as it is in the CGM procedure. A simple workaround to this is to use binary search to find the correct processor but this leads to an $O(\log p)$ slowdown. A more efficient method can be obtained using integer sorting.

Theorem 6. *The* HCGM-PRAM-SIMULATION() *algorithm uses* $O(\frac{n}{s})$ *computation time and* $O(1)$ *supersteps on an* HCGM(n, p, s), *provided that* $\frac{s}{s^{min}} \leq c_1 p^{c_2}$ *for some constants* $c_1, c_2 > 0$, *and* $\frac{s^{min}}{s} n \geq p$.

Proof. Clearly the proof of Theorem 5 extends to this theorem with the exception of finding the processors which service requests. Thus we need only show how this is done. By first sorting the requests locally at each processor, the relevant processors can be determined by (sequentially) scanning the sorted lists using $O(\frac{s_i}{s}n + p) = O(\frac{s_i}{s}n)$ work at each processor P_i, and can therefore be done using $O(\frac{n}{s})$ computation time. Thus we need only consider how to sort the requests.

Fact 1 (Radix Sort [42]). *It is possible to sort (sequentially)* k *integers in the range* $[0, l-1]$ *using* $O(\frac{\log l}{\log k}k)$ *computation and* $O(k)$ *memory.*

Each processor P_i must sort $\frac{s_i}{s}n$ elements in the range $[0, n-1]$. By substituting $k = \frac{s_i}{s}n$ and $l = n$ in Fact 1, we see that the sorting can be done using

$$\begin{aligned}
W_{sort} &\in O\left(\left(\frac{\log n}{\log \frac{s_i}{s}n}\right)\left(\frac{s_i}{s}n\right)\right) \\
&= O\left(\left(\frac{\log \frac{s_i}{s}n + \log \frac{s}{s_i}}{\log \frac{s_i}{s}n}\right)\left(\frac{s_i}{s}n\right)\right) \\
&= O\left(\left(1 + \frac{\log \frac{s}{s_i}}{\log \frac{s_i}{s}n}\right)\left(\frac{s_i}{s}n\right)\right) \\
&\subseteq O\left(\left(1 + \frac{\log c_1 + c_2 \log p}{\frac{s_i}{s}n}\right)\left(\frac{s_i}{s}n\right)\right), \text{ for } \tfrac{s}{s_i} \le c_1 p^{c_2} \\
&\subseteq O\left(\left(1 + \frac{\log c_1 + c_2 \log p}{\log p}\right)\left(\frac{s_i}{s}n\right)\right), \text{ for } \tfrac{s_i}{s}n \ge p \\
&= O\left(\frac{s_i}{s}n\right)
\end{aligned}$$

work at each processor, P_i, and $O(\frac{n}{s})$ time. □

For an algorithm which uses PRAM simulation to be useful in practice, the simulation must be done as efficiently as possible. If we have the extra restriction that $s/\gcd(s_0, \ldots, s_{p-1}) \in O(\frac{s^{min}}{s}n)$ we can simulate the CGM-PRAM-SIMULATION algorithm by having each processor, P_i, simulate $s_i/\gcd(s_0, \ldots, s_{p-1})$ CGM processors thereby reducing the constants in the running time of the HCGM-PRAM-SIMULATION algorithm.

If we have the weaker restriction that $\frac{s}{s^{min}} \in O(\frac{s^{min}}{s}n)$ then the same game can be played by having each processor, P_i, simulate $O(\frac{s_i}{s^{min}})$ CGM processors. Although this leads to an algorithm with running time $O(\frac{n}{s})$, the big-Oh notation hides the fact that the processors are not doing work which is exactly proportional to their speeds. This may or may not be acceptable in practice.

In many cases, PRAM algorithms operate on pointer-based data structures such as lists or graphs and the memory locations accessed by the PRAM processors are dictated by the values of pointers in the data structure. In such cases the data structure can be preprocessed so that the pointers are modified to contain a processor/address pair to allow addressing in constant time. This preprocessing can easily be done in time $O(\frac{n}{s}\log p)$, and will most likely yield the most efficient simulations in practice.

Using the techniques in [61, 32], it is possible to simulate other types of PRAMs on the HCGM model. In particular, extensions of the randomized EREW-PRAM and CRCW-PRAM simulations described in [32] to the HCGM model are possible.

4.5 Circulate

Scientific computations often use a very regular communication pattern in which a set of items is "rotated" or "circulated" through the processors in rounds, so that after p rounds, every processor has seen every item. Examples include dense matrix multiplication, in which the rows of the matrix are rotated, and solutions to the so-called n-body problem, in which the n bodies in question are rotated.

The Circulate pattern takes two ordered lists A and B of size $O(n)$ as input. The computation proceeds in p rounds. During each round each processor sends and receives some portion of B of size $\frac{n}{p}$, and performs some computation on its locally stored portions of A and B. After the p rounds, each element of B has been stored in the same processor as each element in A during exactly one round. The nature of the computation performed in each round may vary, but the running time must be of the form $O(|A_i| \cdot |B_i| \cdot n^c)$, where A_i (resp. B_i) is the sublist of A (resp. B) stored at P_i. This is captured by the following algorithm.

CGM-Circulate(A, B)

1. Repeats Steps 2 and 3 p times.
2. Each processor P_i performs computation on A_i and B_i.
3. Each processor P_i send B_i to $P_{(i+1) \bmod p}$.

The number of supersteps used by the CGM-Partition algorithm is clearly $O(p)$. Initially, A and B are distributed evenly among the processors, and so the amount of computation done during each of the p computation supersteps is

$$O\left(\frac{|A|}{p} \cdot \frac{|B|}{p} \cdot n^c\right) = O\left(\frac{n^{c+2}}{p^2}\right)$$

and the overall computation time is $O(\frac{n^{c+2}}{p})$.

To implement an HCGM version of the Circulate pattern, we need only change the way in which A and B are distributed among the processors. This leads to the following algorithm:

HCGM-CIRCULATE(A, B)

1. Distribute A and B so that P_i stores $\frac{s_i}{s}|A|$ elements of A and $\frac{|B|}{p}$ elements of B.

2. Repeats Steps 2 and 3 p times.

3. Each processor P_i performs computation on A_i and B_i.

4. Each processor P_i send B_i to $P_{(i+1) \bmod p}$.

Theorem 7. *The* HCGM-CIRCULATE(A, B) *algorithm uses* $O(\frac{n^{c+2}}{s})$ *computation time and* $O(p)$ *supersteps on an* HCGM(n, p, s)*, provided that* $\frac{n}{p} \geq p$*, and* $\lfloor \frac{s^{min}}{s} n \rfloor \geq 1$.

Proof. Redistributing A and B in Step 1 is a straightforward matter using the prefix sum algorithm of Theorem 1, and takes $O(\frac{n}{ps^{min}})$ computation time and $O(1)$ supersteps.

During each execution of Step 2, the work done by P_i is given by

$$O\left(\frac{s_i}{s}|A| \cdot \frac{|B|}{p} \cdot n^c\right) = O\left(\frac{s_i n^{c+2}}{ps}\right)$$

and this work can be done in time $O(\frac{n^{c+2}}{ps})$ time. Therefore, over the p rounds, the total computation time is $O(\frac{n^{c+2}}{s})$, and this dominates the overall computation time. $\square$

Chapter 5

HCGM Algorithms

This chapter provides a sampling of algorithms for the HCGM model, and describes some empirical results which show that these algorithms work well. These algorithm are arrived at by expressing existing CGM and BSP algorithms in terms of the communication patterns described in Chapter 4. We present algorithms for the following problems:

1. Parallel insertion and deletion operations on a priority queue illustrate the RANDOM-ASSIGN and LINEAR-PARTITION patterns.

2. Computing the lower envelope of non-intersecting line segments illustrates the LINEAR-PARTITION pattern.

3. List ranking illustrates the PRAM-SIMULATION pattern.

4. Matrix multiplication illustrates the CIRCULATE pattern.

Rather than give a superficial treatment of a large number of algorithms, we have chosen to examine a few algorithms in detail. A consequence of this approach is that we do not provide an exhaustive list of possible HCGM algorithms based on the communication patterns in Chapter 4. However, after presenting each algorithm, we make note of other BSP and CGM algorithms which could be converted to HCGM algorithms in a similar manner.

5.1 Priority Queue Operations

Priority queues are a fundamental data structure used in a large number of graph and optimization algorithms. A common example is the class of "Branch and Bound" algorithms used in combinatorial optimization.

A priority queue, Q, dynamically maintains a set of keys belonging to a linear order. In general, priority queues support a number of operations including insertion of keys, deletion of arbitrary keys, deletion of the smallest key, changing the value of a key, and concatenation of two queues. However, for the purposes of this work, we will consider only the following two operations:

1. *Insert*(k). Insert the key k into Q.
2. *DeleteMin*(). Remove the minimum key, k^{min}, from Q.

Various implementations of priority queues exist, each with different running times for different operations. We note however, that for a comparison based implementation at least one of the above operations must have a running time of $\Omega(\log n)$ where n is the number of keys in Q. Otherwise, sorting n keys could be done in $o(n \log n)$ time by inserting the keys into Q and then removing them in sorted order.

In this section, we consider the problem of performing a batch of m priority queue operations on a priority queue, Q, of size n. Throughout the following discussion, we will assume that $n > m$. The two operations we wish to support are:

1. MultiInsert($k_0, \ldots, k_{m-1}, Q$). Insert the keys $k_0, \ldots, k_{m-1}$ into Q.
2. MultiDelete(m, Q). Delete the m smallest keys from Q.

The algorithms presented in this section are a generalization of the algorithms discovered by Bäumker *et. al.* [4] and independently by Gerbessiotis and Siniolakis [31]. In these schemes, each processor, P_i maintains a local priority queue Q_i which contains $\tilde{O}(\frac{n}{p})$ of the keys in the overall priority queue. When inserting keys into the priority queues, the keys are assigned to processors at random. The MultiInsert algorithm is given below:

MultiInsert$(k_0, \ldots, k_{m-1}, Q)$

1. All processors use the Random-Assign algorithm to randomly assign the keys to processors.

2. Each processor, P_i, inserts the keys received in Step 1 into Q_i.

Theorem 8. *The algorithm* MultiInsert$(k_0, \ldots, k_{m-1}, Q)$ *runs uses* $\tilde{O}(\frac{m}{s}\log n)$ *computation time and* $O(1)$ *supersteps on an* HCGM(n, p, s)*, provided that* $\frac{s^{min}}{s}m \geq 3\ln m$*.*

Proof. By Theorem 3, Step 1 takes $\tilde{O}(\frac{m}{s}\log p)$ computation time and $\tilde{O}(1)$ supersteps. Furthermore, each processor, P_i, receives $\tilde{O}(\frac{s_i}{s}m)$ keys. Therefore, Step 2 takes $O(\frac{m}{s}\log n)$ computation time and dominates the computation time. □

Assigning the keys to processors using the Random-Assign pattern not only ensures load balancing during insertion, but also ensures load balancing during deletion. This is because each local queue, Q_i, contains no more than $c\frac{s_i}{s}m$ of the m smallest keys, with high probability. Thus, the strategy used when deleting keys is to delete too many keys from Q_i and then reinsert those keys which should not have been deleted. This leads to the following algorithm.

MultiDelete(m, Q)

1. Each processor P_i removes the minimum $c\frac{s_i}{s}m$ keys from Q_i.

2. All processors globally sort the keys removed in Step 1.

3. The keys with rank $< m$ in the sorted order are deleted, while the keys with rank $> m$ in the sorted order are reinserted using the MultiInsert algorithm.

Note that the MultiDelete(m, Q) algorithm is a Monte Carlo algorithm as well as a Las Vegas algorithm. It is possible that some Q_i contains more than $c\frac{s_i}{s}m$ of the m smallest keys, in which case the algorithm produces an incorrect result. However, as Theorem 9 shows, this is highly unlikely.

Theorem 9. *Let m_i be the number of the m smallest keys which are stored in Q_i. Then*

$$\Pr\left[m_i \geq c\frac{s_i}{s}m\right] \leq \frac{1}{m^{(c-1)^2}}$$

provided that $\frac{s_i}{s}m \geq 3\ln m$.

Proof. Note that any key is assigned to Q_i with probability $\frac{s_i}{s}$ and that this probability is independent of any other key being assigned to Q_i. Therefore the value of m_i follows the binomial distribution $b(m, \frac{s_i}{s})$. Applying Equation A.1, we get that

$$\Pr\left[m_i \geq c\frac{s_i}{s}m\right] \leq \left(\frac{1}{e}\right)^{(c-1)^2\frac{s_i}{s}m/3} \leq \frac{1}{m^{(c-1)^2}}$$

□

Theorem 10. *The algorithm* MultiDelete(m, Q) *uses $\tilde{O}(\frac{m}{s}\log n)$ computation time and $O(1)$ supersteps on an* HCGM(n, p, s)*, provided that $r = \frac{s^{max}}{s}n$, $\frac{s^{min}}{s}r \geq 2\ln n$, and $p \leq n$.*

Proof. In Step 1, each processor P_i deletes $\frac{s_i}{s}m$ items from Q_i, and can do this using $O(\frac{m}{s}\log n)$ computation time. By Corollary 2, Step 2 can be done using $O(1)$ supersteps and $O(\frac{m}{s}\log m)$ computation time. By Theorem 8, Step 3 can be done using $O(1)$ supersteps and $O(\frac{m}{s}\log n)$ computation time. Thus, all steps can be completed in the stated resource bounds. □

The Monte Carlo MultiDelete(m, Q) algorithm described above can easily be made into a strictly Las Vegas algorithm, with some additional verifications. This can be accomplished by ensuring that after Step 2, the minimum item in each Q_i is greater than the key with rank m. If this does not hold, then items should be removed from Q_i until it does. Note that this does not affect the analysis of the algorithm since, with high probability, no extra items need to be deleted from any Q_i.

A number of BSP and CGM algorithms exist which use the Random-Assign pattern. These include the randomized sorting algorithm of Bader *et. al.* [2], the tree multisearch algorithms of Bäumker *et. al.* [5, 6, 7, 3], and the DAG multisearch algorithms of Gerbessiotis and Siniolakis [29, 30].

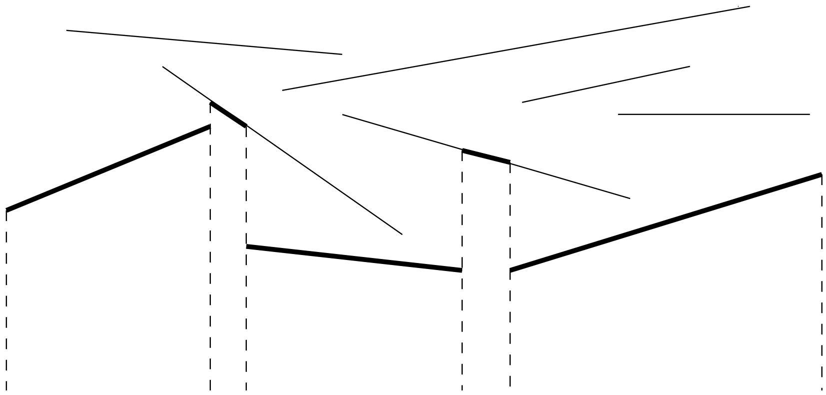

Figure 5.1: The lower envelope of a set of line segments. The portions of the line segments which form the lower envelope are shown in bold.

5.2 Lower Envelopes

Computing the lower envelope of non-intersecting line segments is a classic problem in computational geometry. Given a set of non-intersecting line segments in the plane, the lower envelope problem is to determine which portions of these segments are visible to a viewer standing at $(0, -\infty)$ (see Figure 5.1). A number of (more realistic) visibility problems can be reduced to the lower envelope problem through standard geometric transformations. The sequential complexity of the lower envelope problem is $O(n \log n)$.

The algorithm we describe relies on the following property of the lower envelope of line segments.

Observation 2 (x-Monotonicity). *Any lower envelope, L, is x-monotone, i.e., any vertical line intersects L at most once.*

Dehne, Fabri, and Rau-Chaplin [21, 22] describe a CGM algorithm for the lower envelope problem which uses $O(1)$ supersteps and $O(\frac{n \log n}{p})$ computation time. The algorithm we describe is arrived at by simply replacing the global sort operation used in [21, 22] with the LINEAR-PARTITION algorithm in Chapter 4.

The algorithm works by first computing p individual lower envelopes. Next, the plane is partitioned into p vertical *slabs*, where each slab intersects at most $O(\frac{n}{p})$ of the segments of the lower envelopes computed above. Finally, the lower envelope of the segments in each slab is computed, and the overall lower envelope is the union of the lower envelopes of all slabs. The algorithm is described in detail below, and it's execution is illustrated in Figure 5.2.

LowerEnvelope(S)

1. Each processor P_i computes the lower envelope of it's locally stored segments, call this L_i.

2. All processors globally perform a linear partition of the L_i computed in Step 1, using the x-coordinate of the right endpoint of each segment as the key.

3. Each processor, P_i, determines the vertical line, l_i, through the rightmost segment received in Step 2, and broadcasts this line to all other processors.

4. Each processor, P_i, sends the segments $s \in L_i$ to P_j if and only if s intersects l_j.

5. Each processor, P_i, computes the lower envelope of the segments received in Steps 2 and 4.

Theorem 11. *The lower envelope of n non-intersecting line segments can be computed using $\tilde{O}(1)$ supersteps and $\tilde{O}(\frac{n}{s}\log n)$ computation time on an* HCGM(n, p, s) *provided that $r = \frac{s^{max}}{s}n$, $\frac{s^{min}}{s}r \geq 2\ln n$, and $\frac{s^{min}}{s}n \geq p$.*

Proof. The correctness of the algorithm follows from the correctness of the algorithm in [21, 22]. At the beginning of Step 1 each processor, P_i, contains $O(\frac{s_i}{s}n)$ segments and can therefore compute the lower envelope of these segments in $O(\frac{n}{s}\log n)$ time. By Theorem 4, Step 2 can be done in $\tilde{O}(1)$ supersteps and $\tilde{O}(\frac{n}{s}\log p)$ computation time. Step 3 consists of routing an h-relation with $h = p$. As noted in [22], during Step 4, each processor sends and receives at most $p \leq \frac{s^{min}}{s}n$ segments. This is due to Observation 2, since each L_i intersects each l_j at most once. At the beginning of

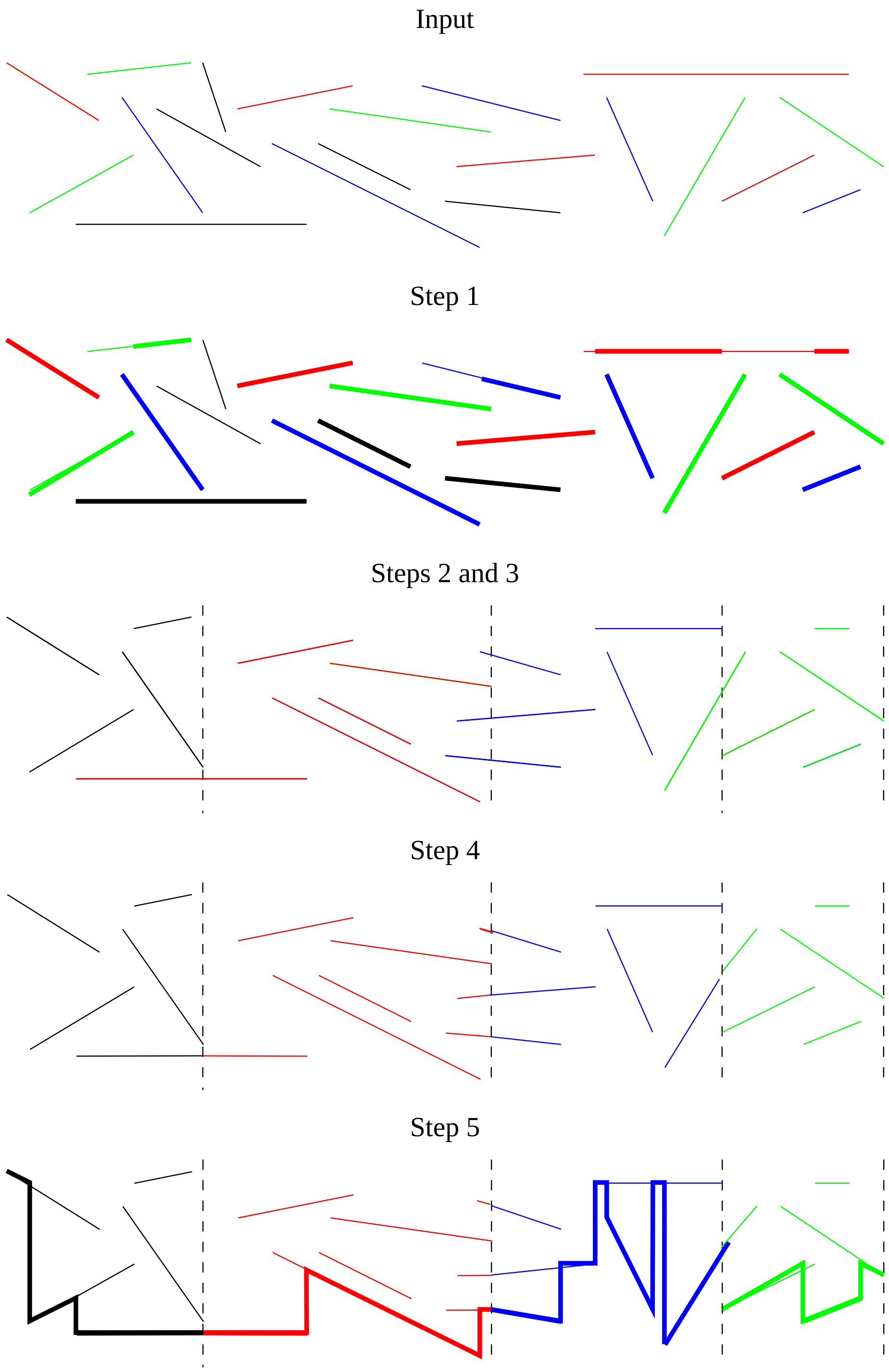

Figure 5.2: Example execution of LowerEnvelope algorithm. Line segments are colored by the processor which holds them.

Step 5 each processor, P_i, contains $\tilde{O}(\frac{s_i}{s}n)$ segments and can therefore compute the lower envelope of these segments in $\tilde{O}(\frac{n}{s}\log n)$ time. □

Dehne, Fabri, and Rau-Chaplin [22] also discuss an extension of the algorithm to the case of possibly intersecting line segments and show that with a small amount of extra memory, this problem can also be solved. Similar comments hold regarding our generalized version of the algorithm.

The Linear-Partition algorithm is extremely useful in adapting CGM and BSP algorithms to the HCGM model. The only communication operation used by the algorithms in [21, 22] is global sorting. A first step in generalizing almost any CGM and BSP algorithms to the HCGM model is to replace all calls to global sort with calls to Linear-Partition. In some cases, the resulting algorithm is in fact more efficient than the original since global sort is often used when a linear partition will suffice. (This is the case with the LowerEnvelope algorithm if a sequential algorithm is used in Step 5 that does not require sorting.)

5.3 List Ranking

The list ranking problem takes as input a linked list and returns as output the distance of each list element to the last element of the list. This list ranking problem has been studied extensively for the PRAM model and optimal randomized and deterministic algorithms have been devised [59].

The simplest (near-optimal) solution to the list ranking problem uses a recursive doubling technique known as pointer jumping (see, e.g., [41]. In pointer jumping, each element is initially assigned a rank of 1, with the exception of the last list element which is assigned a rank of 0. Each element then gets assigned the pointer of its successor, and adds to its rank, the rank of its successor. After repeating this procedure $\log n$ times, each element has the last list element as its successor and is correctly ranked.

In [10], Cáceres *et. al.* describe a CGM algorithm for the list ranking problem. The algorithm begins by finding a p^2-*ruling set* of size $O(\frac{n}{p})$. This is a subset of the original

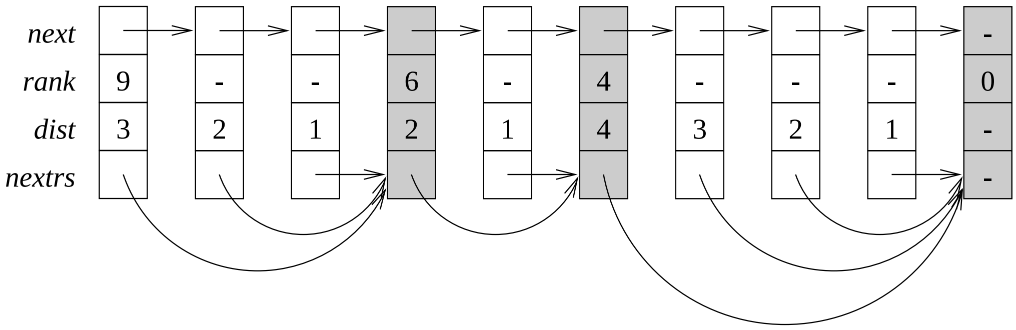

Figure 5.3: An example of the list ranking algorithm of Cáceres *et. al.* Ruling set elements are shown in gray.

list elements such that no two consecutive elements are further than distance p^2 apart. At the same time, the algorithm determines for each list element x, $nextrs(x)$ the first ruling set element which occurs after x in the list, and $dist(x)$ the distance between x and $nextrs(x)$. The ruling set elements are then broadcast to all processors and are ranked sequentially. Elements not in the ruling set are then ranked using the formula $rank(x) = rank(nextrs(x)) + dist(x)$. This is expressed by the following algorithm. See Figure 5.3 for an example.

CGM-List-Rank() [10]

1. All processors compute a p^2 ruling set of size $O(\frac{n}{p})$. At the same time, all processors compute $dist(x)$, and $nextrs(x)$, for all list elements x. This can be done in $O(\log p)$ supersteps and $O(\frac{n}{p}\log p)$ computation time [10].

2. All processors perform an all-to-all broadcast of the ruling set elements.

3. Each processor, P_i ranks the ruling set elements.

4. Each processor, P_i, ranks the elements it contains which are not in the ruling set using the formula $rank(x) = rank(nextrs(x)) + dist(x)$.

Clearly the running time of this algorithm is dominated by Step 1. Therefore the algorithm uses $O(\frac{n}{p}\log p)$ computation time and $O(\log p)$ supersteps.

When adapting this algorithm to the HCGM model there are two problems to overcome. The first is the all-to-all broadcast in Step 2 and the resulting computations in Step 3. This is easily handled gathering the ruling set elements in P^{max} and having P^{max} do the work of ranking these elements sequentially. The second consideration is what to do with the ranked ruling set elements after they have been ranked by P^{max}. They can not be broadcast, since then some slow processors would receive $O(\frac{n}{p})$ elements and become bottlenecks for the algorithm. The answer is to simply replace the elements in the processors to which they were originally assigned. Afterwards, another $O(\log p)$ rounds of pointer jumping allows all processors to determine the ranks of these elements.

HCGM-List-Rank()

1. All processors compute a p^2 ruling set of size $O(\frac{n}{p})$ using the EREW-PRAM simulation procedure described in [10].
2. All processors gather the ruling set elements into P^{max}.
3. P^{max} ranks the ruling set elements.
4. P^{max} sends each (now ranked) ruling set element back to the processor from which it was received.
5. All processors simulate pointer jumping to determine, for each list element x, the values $nextrs(x)$, $dist(x)$, and $rank(nextrs(x))$.
6. Each processor, P_i, ranks the elements it contains which are not in the ruling set using the formula $rank(x) = rank(nextrs(x)) + dist(x)$.

Theorem 12. *The list ranking problem can be solved on an* HCGM(n, p, s) *using* $O(\frac{n}{s}\log p)$ *computation time and* $O(\log p)$ *supersteps.*

Proof. In a preprocessing step, each processor, P_i, can preprocess its subarray in $O(\frac{n}{s}\log p)$ time, so that rather than containing indices, the subarray contains processor/index pairs. In this way, the PRAM simulation in Step 1 and the pointer jumping

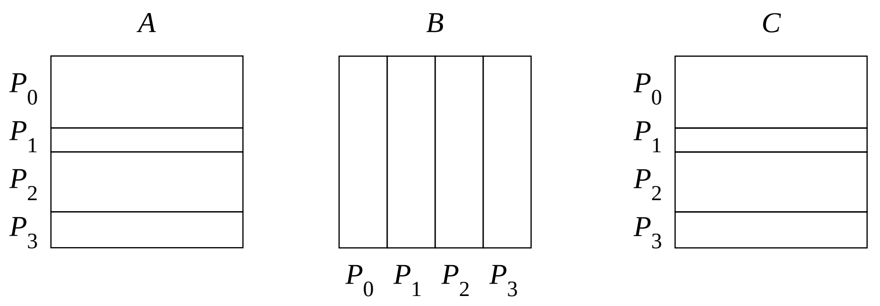

Figure 5.4: Partitioning matrices A, B, and C in a 4 processor system.

in Step 5 can be done in $O(\frac{n}{s}\log p)$ time. Steps 2–4 can clearly be done using $O(1)$ supersteps and $O(\frac{n}{s})$ computation time. □

List ranking is used as a subroutine in a number of parallel tree and graph algorithms. In [10], Cáceres *et. al.* describe a number of CGM graph algorithms based on existing PRAM algorithms. Using the PRAM simulation technique of Chapter 4, most of these algorithms can be adapted to the HCGM model.

5.4 Matrix Multiplication

Matrix multiplication is perhaps one of the most common operations used in large-scale scientific computing. Given two $n \times n$ matrices A and B, we define the matrix $C = A \times B$ as

$$C_{i,j} = \sum_{k=0}^{n-1} A_{k,i} B_{j,k}.$$

In this section, we show how to implement matrix multiplication using the CIRCULATE pattern. We assume that the matrix A is partitioned among the processors so that each processor, P_i holds $\frac{s_i}{s}n$ rows of A and $\frac{n}{p}$ columns of B. At the completion of the computation, P_i will hold $\frac{s_i}{s}n$ rows of C (see Figure 5.4). We denote the parts of A, B, and C held by P_i as A_i, B_i, and C_i respectively.

The matrix multiplication algorithm consists of circulating the columns of B among the processors. Note that when P_i receives column j of B, it can compute column j of C_i. Thus, once P_i has seen all columns of B, it will have computed all of C_i. Although, by now the operation of the algorithm should be obvious, we include it here for the sake of completeness.

HCGM-Matrix-Multiply(A, B)

1. Apply the HCGM-Circulate algorithm, where the A set consists of the rows of A, and the B set consists of the columns of B. When processor P_i receives some columns of B, it computes the corresponding columns of C_i.

Theorem 13. *The* HCGM-Matrix-Multiply(A, B) *algorithm computes the matrix C using $O(\frac{n^3}{s})$ computation time and $O(p)$ supersteps on an* HCGM(n^2, p, s), *provided that $\frac{n}{p} \geq p$, and $\lfloor \frac{s^{min}}{s} n \rfloor \geq 1$.*

Proof. The correctness of the algorithm follows from the fact that the Circulate pattern ensures that every processor P_i, sees every column of B exactly once, thereby enabling it to correctly compute C_i. The running time follows from Theorem 7 and from the fact that the work done by P_i during each round is of the form $O(|A_i| \cdot |B_i| \cdot n)$ □

5.5 Empirical Results

The algorithms for the communication patterns described in Chapter 4 and some of the algorithms in this chapter have been implemented as part of the PLEDA library, an ongoing project whose goal is to supply a portable library of efficient parallel data structures and algorithms [47]. This work builds on the LEDA library of sequential data structures and algorithms [49]. The library is written in C++ and uses MPI for message passing.

Timing results are presented for a sorting algorithm, which uses the Random-Sample and Linear-Partition patterns (see Corollary 2), and for a parallel version of the Floyd-Warshall all pairs shortest path algorithm (see, e.g., [13]), which is based on the Circulate pattern. These results were obtained on a dedicated cluster

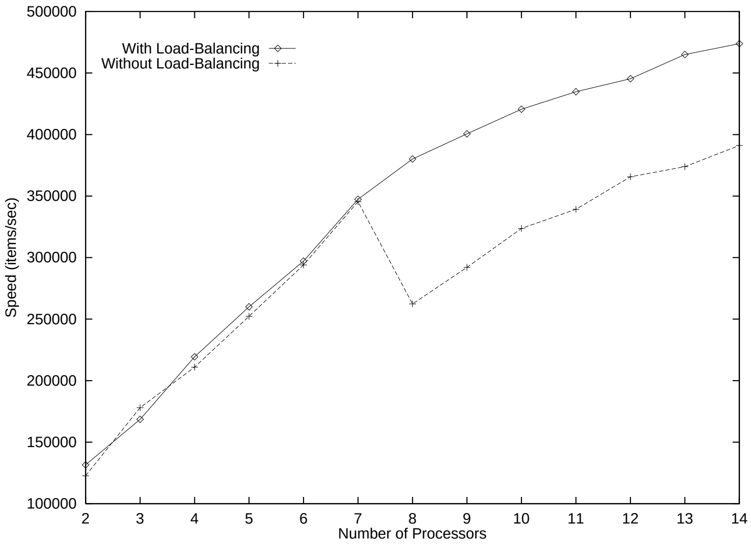

Figure 5.5: Performance of CGM and HCGM versions of Sample Sort.

of workstations consisting of 14 166MHz Pentium processors interconnected by a 100MHz Ethernet switch, running Linux, and using the LAM MPI implementation. In order to simulate slow processors, a crippling process was launched on those processors in order to reduce their effective speed. Crippling processes do nothing but spin in a tight loop performing useless calculations, effectively reducing the speed of the processor to $\frac{1}{2}$ its usual speed. For these tests up to 14 processors were used. P_0 through P_6 were run at the regular speed, while P_7 through P_{13} were crippled.

Figure 5.5 compares the results of using the HCGM Linear-Partition algorithm and then sorting locally against the results obtained by standard Sample Sort [32]. The test sorts a list of $2.5 \cdot 10^6$ integers, using the LEDA implementation of quicksort as the local sorting function. In both cases, the input is initially distributed in a load balanced manner. It is clear from Figure 5.5 that the HCGM version (labelled "With Load-Balancing") of the algorithm performs much better than the standard version (labelled "Without Load-Balancing") when slow processors are introduced into the system.

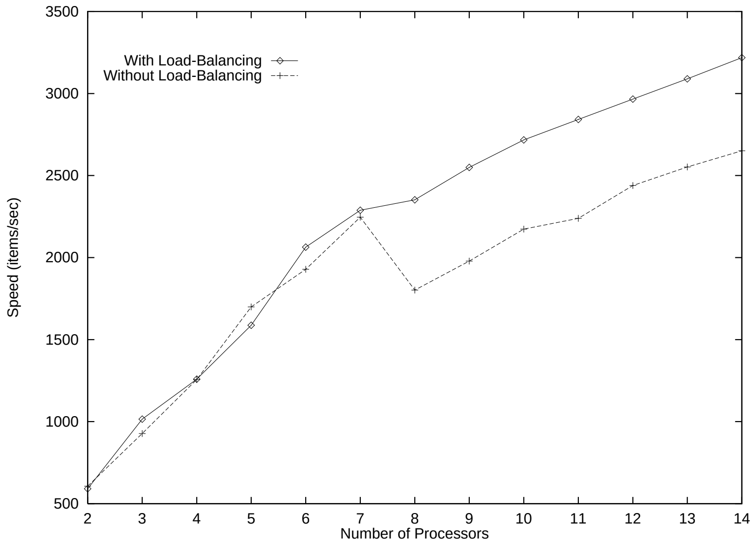

Figure 5.6: Performance of CGM and HCGM versions of Floyd-Warshall algorithm

In order measure the performance of another class of HCGM algorithms we implemented a CGP version of the Floyd-Warshall all pairs shortest path algorithm which uses the Circulate pattern on the columns of the adjacency matrix. The results of running this test with $n = 1.0 \cdot 10^3$ are shown in Figure 5.6. As we would expect, the HCGM version of the algorithm performs much better. With the CGM version it is faster to run the application with 7 fast processors than it is to run it with 7 fast processors and 4 slow processors, while with the HCGM version the performance improves each time a processor is added to the cluster.

Chapter 6

Conclusions

This part of the thesis has defined a model for parallel computing on heterogeneous systems, and described a number of algorithms for this model. Although this model is very simple, the empirical results in Chapter 5 suggest that it captures the most important aspects of actual systems, and algorithms which the model predicts as being fast tend to work well in practice.

There are several directions in which this work could be extended. Although the HCGM model appears to be a fairly good model for developing algorithms, it is not (nor is it intended to be) an accurate predictor of exact running times. One could argue that the asymptotic analyses used in this thesis are not strong enough, since in practice processor speeds do not vary by more than constant factors. To this end, it might be useful to define an HBSP model which incorporates the BSP parameters g and L. Tests like those done in [63] could then be performed to determine whether the resulting model is an accurate predictor of performance.

We note that such extensions to the BSP model are not entirely trivial. The question of whether to apply a bandwith limitation locally, by e.g., charging a cost of $\frac{g}{s_i}$ for each word sent or received by P_i, or globally as is done in the HCGM model is not an easy one. In a recent paper, Adler *et. al.* [1] have shown that a model in which bandwidth is restricted globally has significant theoretical advantages over a model in which bandwidth is restricted locally. Empirical testing is necessary to determine which is more appropriate, and it may be the case that the choice of global versus local bandwidth restriction depends on the actual parallel machine being used.

Another direction for future work is a direct comparison of the algorithms described in this thesis with algorithms based on the overparitioning approach described in Chapter 1. The difficulty with this is that although algorithms which use the overpartitioning approach exist for some problems, most of the algorithms described in Chapter 5 do not have counterparts based on overpartitioning. Thus, regardless of the results of the outcome of such tests, most of the algorithms in Chapter 5 are the only algorithms currently available for such problems.

One advantage which algorithms based on overpartitioning may have over HCGM algorithms is that they can adjust to changing load in systems in which processors are shared among users. Although the HCGM model could be extended to incorporate dynamically changing processor speeds, the algorithms for this model would have to be very different than those presented in this thesis. This is due to the fact that HCGM algorithms attempt to minimize the number of supersteps, which means maximizing (with reason) the amount of computation performed between communication operations. Therefore, if processor speeds change frequently and drastically, most of the computation will be non-optimal. It seems that the CGP paradigm of algorithm design is simply not suited for a scenario in which processor speeds change dynamically.

The communication patterns of Chapter 4 are also interesting in their own right. Since most existing CGP algorithms can be expressed in terms of these patterns, they are an excellent starting point for a software framework which is to support the development of CGP algorithms. This is the direction being pursued in the PLEDA project [47]. Ongoing work in this area includes the implementation and testing of more algorithms, as well as keeping the list of communication patterns up-to-date as new algorithms are developed which use different patterns. On a more theoretical note, it may be interesting to develop algorithms for the patterns of Chapter 4 which do not rely on randomization.

Bibliographic Note

The work described in this part of the thesis has been accepted for publication in the proceedings of the 12th Annual ACM Symposium on Applied Computing (SAC'98) [48].

Part II

Multiresolution Surface Modeling

Chapter 7

Introduction

Triangle meshes are used in a number of fields for modelling the surfaces of real world objects. A triangle mesh is a graph which is isomorphic to a planar triangulation (c.f. [57]), each of whose vertices is assigned a position in $\Re^3$. An example of a triangle mesh which models a human head is shown in Figure 7.1. In the field of geographic information systems (GIS), triangle meshes are used as to model the surface of the earth. In computer graphics, triangle meshes are used to model the surfaces of objects in virtual worlds. In CAD systems triangle meshes are used to model the surfaces of machine parts.

Often meshes are generated from data obtained using electronic imaging equipment such as laser range finders, remote sensing equipment, and satelite imaging units. As this equipment increases in fidelity, the size of the resulting meshes increases as well. One method of dealing with this dramatic increase in data size has been *mesh simplification*. Mesh simplification involves reducing the number of vertices in a mesh in an intelligent manner, so that accuracy of the model is maintained as much as possible.

Mesh simplification is a time consuming process, and in many cases, manual intervention is necessary in order to preserve features of the mesh which may be important, but can not be recognized by a computer. Additionally, mesh simplification involves simplifying a mesh to a given level of detail (number of vertices or error tolerance).

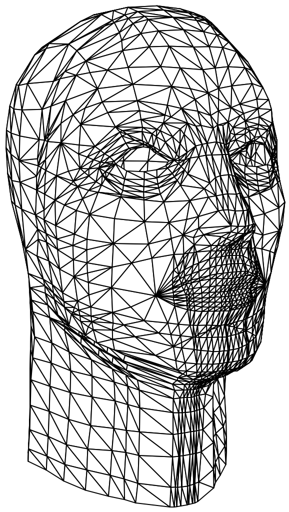

Figure 7.1: An example of a triangle mesh which models a human head.[1]

For some applications, this level of detail may be inappropriate. If the level of detail is insufficient, then the application may produce poor or even incorrect results. If the level of detail is too high, then the application may become computationally inefficient or even infeasible.

It is this latter problem of mesh simplification that is addressed by the field of multiresolution surface modelling. Essentially, multiresolution surface modelling involves performing mesh simplification using a series of localized simplification operations to produce a sequence of progressively less detailed meshes $M_n, M_{n-1}, \ldots, M_0$. While the simplification takes place, the simplification operations are recorded. In this way, the operations can be inverted, and the original mesh can be recreated. In fact, given the mesh M_i, any of the of the meshes $M_n, M_{n-1}, \ldots, M_0$ can be recreated. Thus, the level of detail can be varied according to the requirements of the application. For illustration, Figure 7.2 shows an example of a multiresolution terrain model at 3 different levels of resolution.

In [38], Hoppe defines some desirable features of a multiresolution surface model

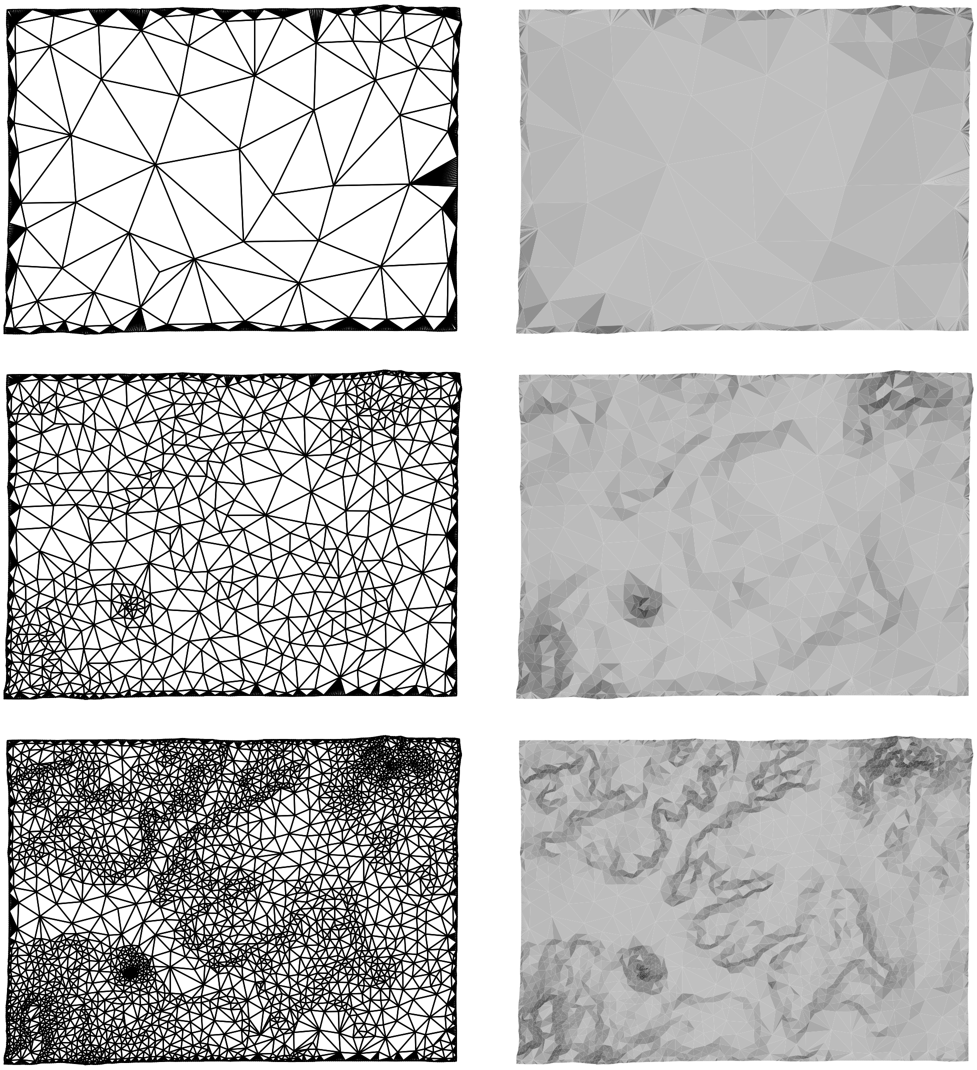

Figure 7.2: A triangle terrain at 3 different levels of resolution (left) and it's corresponding shaded image (right).

- *Level-of-Detail (LOD) Approximation* allows for viewing and manipulating the mesh at varying levels of detail (number of vertices).
- *Progressive transmission* allows for the transmission of a mesh across a network, so that the receiver can immediately display an approximation of the mesh, and refine this display as more data is received.
- *Selective refinement* is the ability to view and manipulate part of the mesh at a very high level of detail while the remainder of the mesh remains at a low level of detail.

In this part of the thesis we are interested in multiresolution surface models and their applications to other fields, with special emphasis on geographic information systems. The main technical contribution of this portion of the thesis is a solution to the problem of efficient selective refinement of progressive meshes posed by Hoppe in [38].[2] Using the ideas behind this solution, a number of applications are presented, including point location, elevation queries, visibility queries, isoline/contour extraction (conversion to contour lines). Empirical results are presented for some of these algorithms which show that they are of considerable practical relevance. Furthermore, we sketch representations a multiresolution surface model in external memory.

The remainder of this part of the thesis is organized as follows: Chapter 8 discusses other approaches to multiresolution terrain modelling and reviews the progressive mesh representation upon which our work is based. Chapter 9 describes our solution to the problem of efficient selective refinement in progressive meshes. Chapter 10 discusses some applications of this solution. Finally, Chapter 11 summarizes and outlines directions for future work.

[2] A selective refinement scheme similar to the one described in Chapter 9 has been independently and concurrently proposed by Hoppe [39].

Chapter 8

Survey of Existing Work

The problem of level of detail approximation in triangulated meshes has received a significant amount of attention in recent years. In this chapter, we give a brief critical survey of this work. We broadly classify these schemes into three categories: (1) the *tree-based* schemes of De Floriani *et. al.* [17, 19, 18], (2) the *DAG-based* scheme of Dobrindt and de Berg [16] and Puppo [58], and (3) the *Progressive Mesh* scheme of Hoppe [38].

8.1 Tree-Based Schemes

The approach to multiresolution modelling taken in [19, 18] is to generate a mesh consisting of nested triangles, and to arrange these triangles into a tree shaped hierarchy (see Figure 8.1). At the top level of the hierarchy, the mesh consists of a fixed number of very large triangles. To refine (all or part of) the mesh, the hierarchy is searched in a top-down manner, and large (parent) triangles are replaced by groups of small (child) triangles. This process is repeated until a sufficient level of detail has been achieved.

Since the triangles of the hierarchy are arranged as a tree with small fixed degree, this scheme tends to be very efficient in practice. However, it does have some disadvantages. As is already evident in the three level hierarchy of Figure 8.1, triangles at lower levels in the hierarchy tend to become elongated, due to the fact that the long

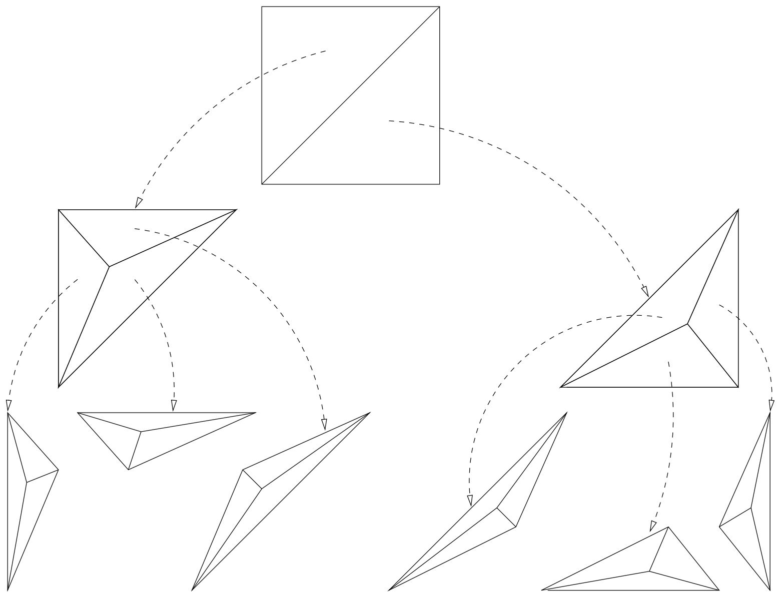

Figure 8.1: An example of a tree-based hierarchy.

edges of the low-resolution mesh are also present in the high-resolution mesh. Such long triangles can lead to numerical instabilities in computations, and aliasing effects in graphics applications. Methods which avoid these elongated triangles by splitting long edges are described in [18], but these can lead to vertical faces (discontinuities) in the TIN.

8.2 DAG-Based Schemes

Dobrindt and De Berg [16], and De Floriani [17] address the problem of the thin triangles which occur in tree-based schemes by using a very different approach. By repeatedly deleting an independent set of TIN vertices, and retriangulating the resulting holes, a hierarchy of triangulations which has a depth of $O(\log n)$ and a total size of $O(n)$ is achieved (see Figure 8.2). In this case, refinement is achieved by replacing a *group* of triangles from a higher level in the hierarchy, with a group of triangles from a lower level in the hierarchy. Using this method, Dobrindt and de Berg show

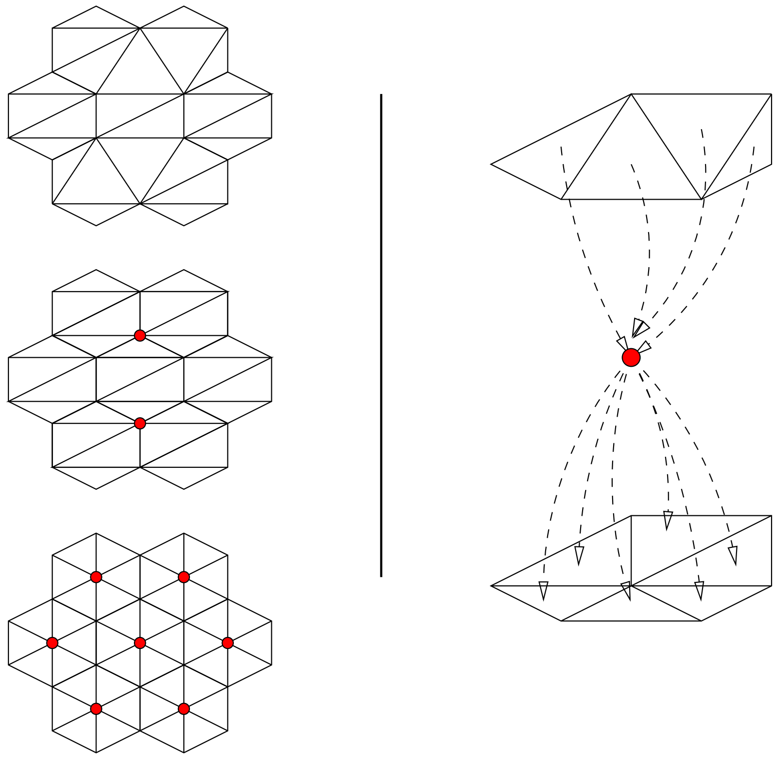

Figure 8.2: An example of a DAG-based hierarchy.

that the Delaunay triangulation of the data can be maintained at all levels of the hierarchy.

While this approach solves the problem of thin triangles, it tends not to be as efficient as tree based schemes. The storage requirements of this scheme are much larger since, on average, every vertex which is deleted generates an additional 10 pointers in the hierarchy (see Figure 8.2). Furthermore, although the hierarchy has depth $O(\log n)$, there are constant factors hidden in the big-O notation (Dobrindt and De Berg report hierarchies of with depths varying between $1.7 \log_2 n$ and $2.4 \log_2 n$ depending on the method used to select vertices for deletion).

Puppo [58] takes the DAG-based approach one step further by describing a general DAG-based framework called the *MultiTriangulation*. In this scheme, a group of triangles can be replaced by another group of triangles as long as the boundaries of

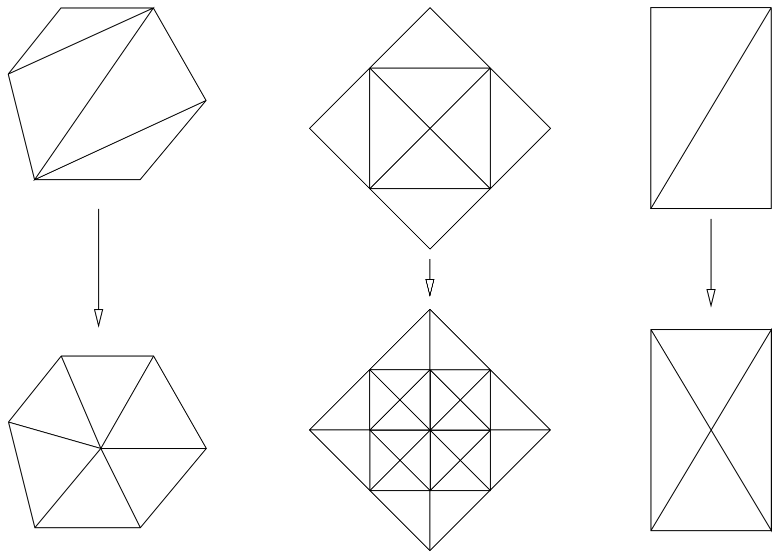

Figure 8.3: Some possible transformations in a MultiTriangulation.

the two groups match (see Figure 8.3). In a companion paper [26], it is shown that most existing schemes, including those in [16] and [19], can be expressed in terms of the MultiTriangulation (although sometimes the MultiTriangulation is less efficient than the original scheme). Although the MultiTriangulation is quite powerful and expressive, current implementations still seem to exhibit the same high overheads associated with the DAG-base scheme of Dobrindt and de Berg [27].

8.3 The Progressive Mesh Representation

Hoppe's progressive mesh (PM) representation [38] is based on the *edge collapse* transformation and its inverse the *vertex split*. An edge collapse involves collapsing an edge by identifying its two incident vertices, v_1 and v_2, into a single aggregate vertex v. The two adjacent faces (v_1, v_2, v_l), and (v_1, v_2, v_r) vanish in the process. An edge collapse and its inverse vertex split are illustrated in Figure 8.4.

In the progressive mesh representation, a mesh M, is represented as a pair, $(M_0, \mathit{vsplits})$, where M_0 is a coarse-grained mesh and *vsplits* is a list of vertex splits

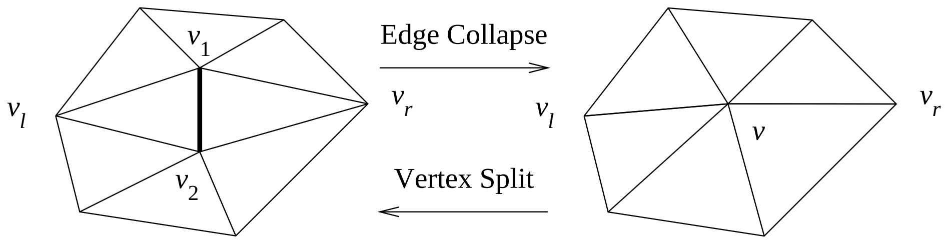

Figure 8.4: The edge collapse transformation and its inverse, the vertex split

which will reproduce the original mesh when applied in order. A vertex split transformation *vsplits*$(v, v_1, v_2, v_l, v_r, A)$ splits the aggregate vertex v into two vertices v_1 and v_2 and adds the edges (v_1, v_l), (v_1, v_r), (v_2, v_l), and (v_2, v_r). A stores attribute information for the neighbourhood of the transformation, including, but not limited to, the positions of the two new vertices.

We call v the *parent* of v_1 and v_2, and we call v_1 and v_2 the *children* of v. We say that a vertex u is an *ancestor* of a vertex v if $u = v$ or if u is an ancestor of the parent of v. In this case we call v a *descendent* of u. We denote by M_i the mesh obtained by performing the first i elements of *vsplits* on the coarse grained mesh M_0. The original mesh, M, can be obtained by applying all the elements of *vsplits* in order, i.e., $M_{|vsplits|} = M$.

The progressive mesh construction algorithm takes a mesh, M, performs a series of edge collapse operations to get the mesh M_0, and sets *vsplits* to the list of vertex split operations that invert the edge collapses performed. The order in which the edge collapses are performed is determined by an application specific fitness function (e.g., minimizing the geometric error). The edges of the mesh are placed in a priority queue based on their fitness, and are removed one at a time as they are collapsed. Edges in the neighbourhood of a collapse may have their priorities updated. A fitness function which considers the geometry, discrete attributes, and scalar attributes of the mesh is described in [38]. Sufficient requirements for an edge collapse to be valid are described in [40].

Chapter 9

Selective Refinement of Progressive Meshes

In Chapter 7, we saw that it may be desirable to work with a mesh at a lower level of resolution than the original data. More generally, we may wish to work with parts of the mesh at a high level of resolution, and the rest of the mesh at a lower level of resolution. For example, if the mesh is a terrain representing a city, the user may only be interested in a particular part of the city. Therefore, it is not necessary (and inefficient) to display the uninteresting parts of the city at a high level of resolution. The *selective refinement* problem is that of extracting a mesh which is at a high level of resolution in a query region q while leaving the mesh outside of q at a low level of resolution.

In this chapter we are concerned with the selective refinement problem on progressive meshes. In particular, we devise an algorithm to extract all the triangles of of M which intersect a query region q while leaving the mesh outside the query region relatively untouched.

In [38], a brute force method of selective refinement is described which examines every vertex split record to determine whether or not it should be split. There are two disadvantages of this algorithm. The first is that the running time is $O(|vsplits|)$ regardless of the number of splits actually performed. The second is that it is difficult to perform an exact selective refinement, that is, to extract all the triangles of M

which intersect a specified query region, q. The reason for this is that there may be vertex splits which lie outside of q, but which affect the triangles within q.

The following sections describe a new efficient method of performing selective refinement in progressive meshes. Our technique uses information about dependencies in the *vsplits* list to achieve better efficiency. The method is simple, robust, and easy to implement. This method is the main technical contribution of this part of the thesis.

Our approach can be summarized as follows: In a preprocessing phase, we associate with each vertex split record a *region of influence* outside of which splitting the vertex has no effect. By using the parent/child relationships between vertex split records, these regions of influence can be organized as a forest of rooted binary trees. When the records are organized in this manner it is possible to efficiently identify all the vertex split records whose regions of influence intersect a given query region q. Once the relevant vertex split records have been identified, they can be sorted and applied in order.

9.1 Computing the Region of Influence

We treat the *vsplits* list as a dependency graph in which a vertex v_1 depends on a vertex v if v is split and one of the resulting vertices is v_1. Observe that the dependency graph is a forest of rooted binary trees whose roots are the vertices of the coarse TIN, M_0, and whose leaves are the vertices of the original mesh, M (see Figure 9.1 for an illustration).

With each node v in the dependency graph we associate an axis aligned 3-Dimensional bounding box, denoted $roi(v)$ which represents the region outside of which this vertex split has no influence. We define $roi(v)$ recursively as follows: if v is a leaf then $roi(v)$ is the smallest box which encloses all the neighbours of v in M, otherwise $roi(v)$ is the smallest box which encloses $roi(v_1)$ and $roi(v_2)$ where v_1 and v_2 are the two children of v. These boxes, i.e., the roi's can be computed in a bottom-up fashion during the progressive TIN construction procedure. Alternatively, these boxes can be constructed using a post order traversal of the forests.

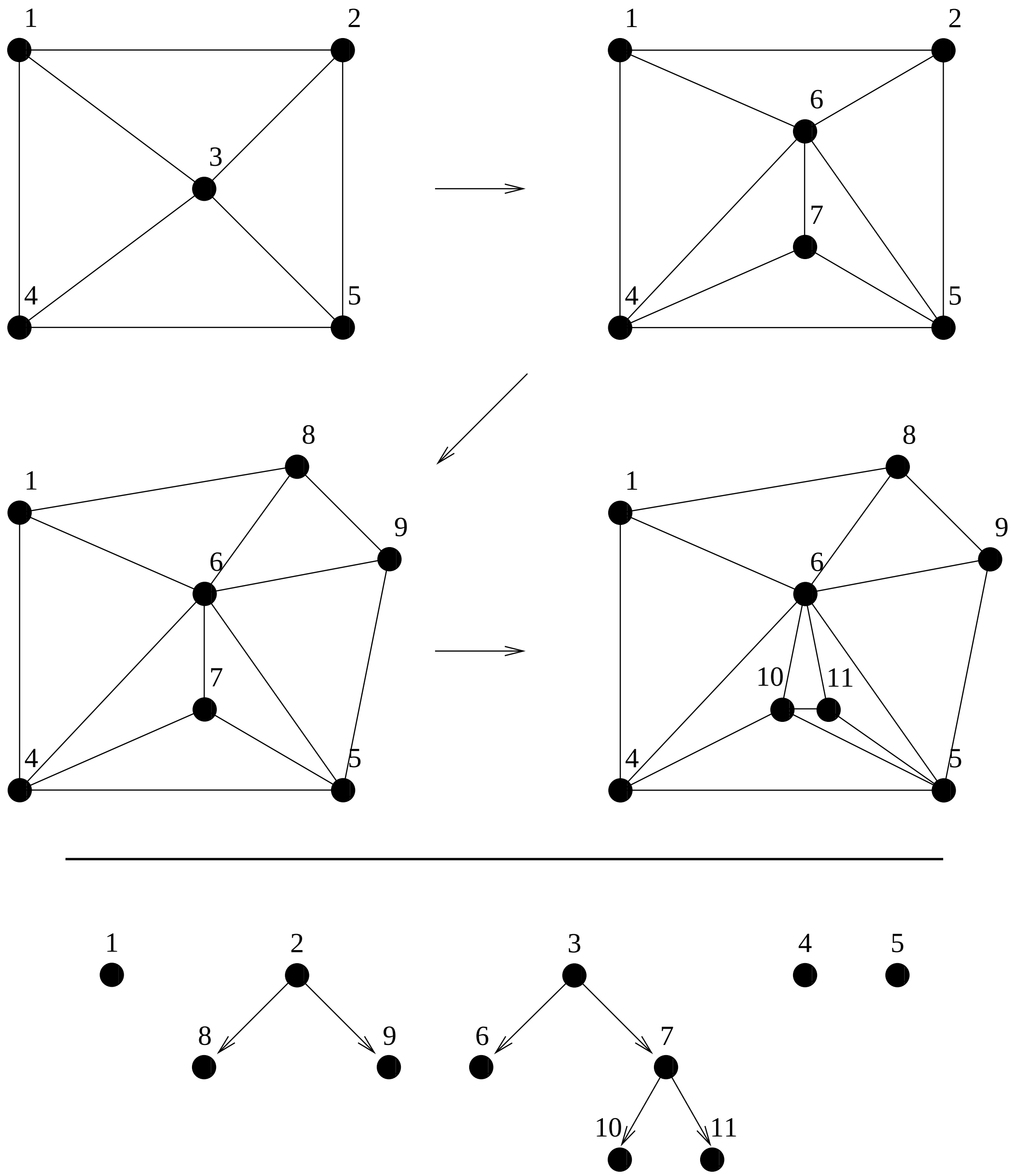

Figure 9.1: A sequence of vertex splits and its associated dependency graph.

The following lemma shows us that if we apply to M_0 every vertex split v such that $roi(v)$ intersects q, then all triangles of M which intersect q will be reconstructed.

Lemma 1. *Let u be a vertex incident to a triangle in M which intersects the query region q. Then, for all ancestors v of u, $roi(v)$ intersects q.*

Proof. The proof is by induction on the level of v in the dependency graph. The base case is when $v = u$, and is true since $roi(u)$ contains all triangles incident to u. Next, w.l.o.g., suppose v is an internal vertex which was formed by collapsing the edge (v_1, v_2). Note that one of v_1 or v_2 is an ancestor of u, and by the inductive hypothesis one of $roi(v_1)$ or $roi(v_2)$ intersects q. This implies that $roi(v)$ intersects q as well, since $roi(v)$ contains $roi(v_1)$ and $roi(v_2)$. $\square$

For completeness, the following pseudocode shows how to construct the *roi* information. The notation $neighb(v)$ is used to denote the neighbourhood of the vertex v in M, i.e., $neighb(v) = \{v\} \cup \{u : (u, v) \in E(M)\}$. The notation $bb(X)$ is used to denote the bounding box of the set X, i.e., the smallest axis-aligned box which contains X.

BUILD-ROI()
forall $v \in V(M)$ **do**
    $roi(v) \leftarrow bb(neighb(v))$
forall $v \in V(M_0)$ **do**
    BUILD-ROI-2(v)
return

BUILD-ROI-2(v)
if $vsplits[v].v_1 \neq nil$ **then**
    BUILD-ROI-2($vsplits[v].v_1$)
    BUILD-ROI-2($vsplits[v].v_2$)
    $roi(v) \leftarrow bb(roi(vsplits[v].v_1), roi(vsplits[v].v_2))$
return

Lemma 2. *The running time of procedure* BUILD-ROI() *is* $O(n)$*.*

Proof. Lines 1 and 2 of BUILD-ROI() take $O(n)$ time, since each vertex of M is examined once, and each edge is examined twice (once for each end point). Lines 3 and 4, including the call to BUILD-ROI-2, take $O(n)$ time, since they perform a post-order traversal of the dependency graph, which is a forest of size $O(n)$. □

9.2 Retrieving the Vertex Splits

Once the *vsplits* list is augmented with the *roi* information, selective refinement can be done in the following manner: For each vertex of M_0 we search the tree rooted at M_0 and retrieve all vertex splits, v, for which $roi(v)$ overlaps the query region, q. It is not necessary to search the children of v if $roi(v)$ does not intersect q, since the *roi* of the children of v are contained in $roi(v)$.

```
GET-SPLITS(q)
splits ← λ
forall v ∈ V(M0) do
    splits ← splits ∪ GET-SPLITS-2(v, q)
return splits
```

```
GET-SPLITS-2(v, q)
if v = nil then
    return λ
if q intersects roi(v) then
    return {v} ∪ GET-SPLITS-2(vsplits[v].v1, q)
        ∪ GET-SPLITS-2(vsplits[v].v2, q)
return λ
```

Lemma 3. *The procedure* GET-SPLITS(q) *returns exactly the vertex splits* v *such that* $roi(v)$ *intersects* q*.*

Proof. Clearly, a vertex split, v, such that $roi(v)$ does not intersect q is never reported. Furthermore, any vertex split which is examined in Line 3 and has a region of influence which intersects v is reported in Line 4 of GET-SPLITS-2. Thus we need only show that all such vertices are examined in Line 3. Note that the only reason a vertex split, v, is not examined in Line 3 is if one of its ancestors, v' had a region of influence that did not intersect q. But in this case, $roi(v)$ can not intersect q, since $roi(v)$ is contained in $roi(v')$, which does not intersect q. Therefore, every vertex split which is not examined in Line 3 of GET-SPLITS-2 does not have a region of influence which intersects q. This completes the proof. □

Lemma 4. *The procedure* GET-SPLITS(q) *has running time* $O(|M_0| + k)$ *where* k *is the number of vertices,* v*, such that* $roi(v)$ *intersects* q.

Proof. We count the number of calls to the GET-SPLITS-2 procedure, since this clearly bounds the total running time. The procedure is called at most $|M_0|$ times in the GET-SPLITS procedure. The number of recursive calls is at most $3k$, since each vertex which is reported results in calling the procedure for at most two unreported vertices. □

9.3 Sorting and Applying the Vertex Splits

Once the *vsplits* records are retrieved they need to be sorted and applied in order. The sorting could be done by a standard sorting algorithm, but this would take $O(k \log k)$ time. A more efficient method can be obtained by observing that the order in which the vertex splits can be applied is not necessarily unique. In fact, the only dependencies between vertex splits come from the parent-child relationship on which our data structure is based, and the *left-neighbour*, and *right-neighbour* relationships. I.e., using the notation in Section 8.3 the vertices v_1 and v_2 are dependent on the vertex v, and v is dependent on the vertices v_l and v_r. Both these relationships can be represented as a directed acyclic graph of size $O(k)$ and thus a feasible order for the edge collapses can be obtained in $O(k)$ time by performing a topological sort (c.f. [13]).

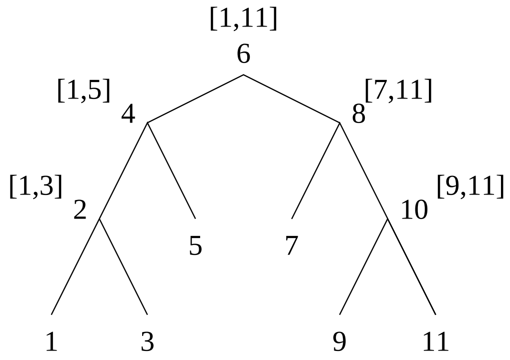

Figure 9.2: An example of the *rank* and *range* numbering.

When splitting a vertex v, it is possible that v_l (or v_r) may not exist, since it was not split. Then the dependency is with the nearest ancestor of v_l which appears in the retrieved *vsplits* records. This requires that we locate the nearest living ancestor of a vertex. In order to do this we use an in-order numbering of the vertices in the dependency graph, and associate with each node v, a number, $rank(v)$ which is the in-order traversal number of v, and an interval, $range(v)$ which contains the ranks of the descendents of v as well as v itself. This information is computed in a precomputation phase. With this information, it is possible to find the nearest living ancestor of v_l by searching the neighbourhood of v for a vertex, v_l', such that $rank(v_l) \in range(v_l')$. See Figure 9.2 for an example of such a numbering.

The following functions produce the *rank* and *range* information used for finding ancestors. After these functions are run, one can determine if a vertex u is an ancestor of v by checking if $rank(v) \in range(u)$.

Build-Ranges()
$next \leftarrow 0$
forall $v \in V(M_0)$ **do**
  $next \leftarrow$Build-Ranges-2$(v, next)$
return

```
BUILD-RANGES-2(v, next)
marker ← next
if vsplits[v].v1 ≠ nil then
    next ← BUILD-RANGES-2(vsplits[v].v1, next)
    rank(v) ← next
    next ← BUILD-RANGES-2(vsplits[v].v1, next + 1)
else
    rank(v) ← next
    next ← next + 1
range(v) ← [marker, next − 1]
return next
```

Lemma 5. *The running time of procedure* BUILD-RANGES() *is* $O(n)$.

Proof. The functions implement a post-order traversal of the dependency graph, which has size $O(n)$. □

The SELECTIVE-REFINE function is the main entry point for the selective refinement algorithm. It first finds all the vertex splits whose regions of influence intersect the query region, and then applies these splits in an order which respects all dependencies.

```
SELECTIVE-REFINE(q)
splits ← GET-SPLITS(q)
forall v ∈ splits do
    if v ∈ M0 then
        APPLY-SPLITS(v, splits)
return
```

```
APPLY-SPLITS(v, splits)
if v ≠ nil and v ∈ splits then
    SPLIT(v, vsplits)
    APPLY-SPLITS(vsplits[v].v_1, vsplits)
    APPLY-SPLITS(vsplits[v].v_2, vsplits)
return
```

The SPLIT function is what performs a vertex split operation. Lines 1–2 ensure that we don't split a vertex that was already split. Lines 3–6 ensure that the left and right neighbours or their nearest ancestor are present before we do the split. Line 7 actually performs the split as discussed above. The predicate *active*(v) is true when vertex v appears in the mesh at its current state of refinement and false otherwise. The *activeancestor*(u, v) function answers the active ancestor of vertex v which is adjacent to vertex u in the current mesh. This is determined by searching the neighbourhood of u for the vertex v' which satisfies $rank(v) \in range(v')$.

```
SPLIT(v, splits)
1  if not active(v) then
2      return
3  while [not active(vsplits[v].v_l)] and [activeancestor(v, vsplits[v].v_l) ∈ splits]
4      SPLIT(activeancestor(v, vsplits[v].v_l), splits)
5  while [not active(vsplits[v].v_r)] and [activeancestor(v, vsplits[v].v_r) ∈ splits]
6      SPLIT(activeancestor(v.vsplits[v].v_r), splits)
7  PERFORM-SPLIT(vsplits[v])
8  return
```

Lemma 6. *The running time of the* SELECTIVE-REFINE *procedure is* $O(|M_0| + kd)$*, where* k *is the number of vertices whose regions of influence intersect* q*, and* d *is the average degree of these vertices.*

Proof. The running time of the GET-SPLITS procedure has been shown to be $O(|M_0| + k)$ in Lemma 3. Therefore, we need only prove the running time of the APPLY-SPLITS procedure. Note that the number of calls to the APPLY-SPLITS procedure is at most

3 times the number of calls to the SPLIT procedure. In turn, the number of calls to the SPLIT procedure is at $2k$, since for each $v \in splits$, SPLIT is called at most once from APPLY-SPLITS and once recursively. Disregarding recursive calls, the running time of APPLY-SPLITS is constant, and the running time of *Split* is $O(d')$ where d' is the degree of the vertex being split. Amortizing this value over all calls to SPLIT we get a running time of $O(kd)$. This completes the proof. □

9.4 Analysis and Comments

Finally, we summarize with a theorem describing our selective refinement scheme and comment on other practical aspects of the scheme.

Theorem 14. *Given a mesh in the progressive mesh representation, the selective refinement scheme uses $O(n)$ preprocessing time and $O(|M_0| + kd)$ query time, where $|M_0|$ is the size of M_0, k is the number of vertex splits whose regions of influence intersect the query region, and d is the average degree of the vertices retrieved.*

Proof. The preprocessing resources follow from Lemma 2 and Lemma 5. The query time follows from Lemma 6. □

At this point we note that it is also possible to generalize this scheme to perform selective refinement on any mesh $M_t \in \{M_0, \ldots, M_{|vsplits|}\}$. In this case, the query region is returned at exactly the level of detail at which it appears in M_t. To achieve this generalization, we simply "prune" the search when a vertex split record is reached who's index is greater than t. In analyzing the running time of the generalized algorithm, the value of k is defined in the same manner as above, but with respect to the vertex split sequence $v_1, \ldots, v_t$.

From the pseudocode, it is easy to see that the preprocessing and selective refinement procedures are quite simple, and hide only small constants in the big-Oh notation (see Section 9.6 for empirical results). Another merit of this scheme is that since it uses only axis aligned bounding boxes, the search procedure can be implemented using only comparison operations and is therefore not subject to the rounding errors inherent in floating point arithmetic computations.

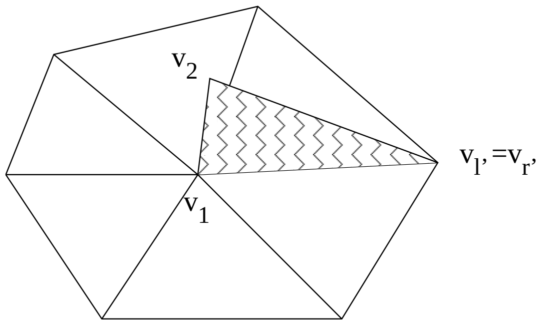

Figure 9.3: An example of the case where $v_l' = v_r'$.

9.5 Extensions and Implementation Notes

9.5.1 Dealing with Missing Neighbours

In Section 9.3, it was noted that when splitting a vertex v, it may be the case that one (or both) of the vertices v_l or v_r may not be present in the mesh. In this case we suggest using the nearest living ancestor of v_l or v_r, call these v_l' and v_r', respectively. An important point to note when implementing this is that it may be the case that $v_l' = v_r'$. In this case, the two resulting triangles (v_1, v_2, v_l) and (v_1, v_2, v_r) are identical, and are only connected to the mesh by a single edge (see Figure 9.3 for an example). Depending on the application, these types of triangles may or may not cause problems. For display purposes, a reasonable solution is to simply flag these special triangles and not render them.

Another approach to solving this problem, which is proposed in [39] is to force splits of v_l' and v_r' and their descendents until v_l and v_r are recreated, at which point the vertex v can be split. This approach seems to work well, and prevents extremely abrupt changes in resolution which may be visually unpleasant. Unfortunately, it may also significantly increase the number of vertices which are split thereby increasing the running time of the selective refinement algorithm.

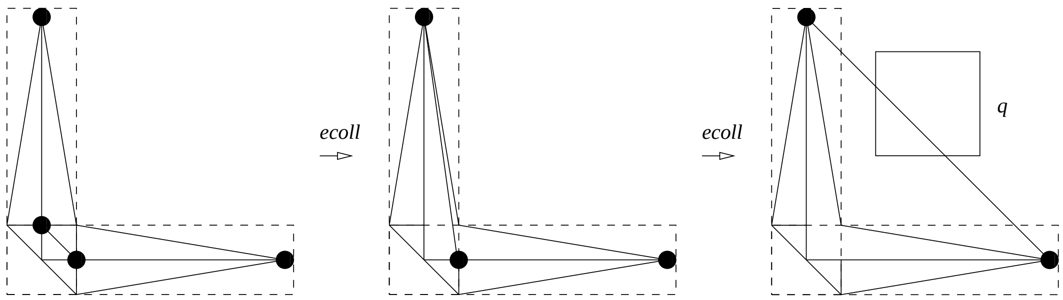

Figure 9.4: An example in which a triangle not in M intersects q.

9.5.2 Invalid Triangles in Query Regions

Although the selective refinement scheme described in this chapter guarantees that all the triangles of M which intersect the query region q are extracted, it does not guarantee that there are not other triangles not in M which intersect the query region. Figure 9.4 shows an example in which this occurs. The figure shows two edge collapses, and the resulting mesh with the *roi* boxes shown as dashed lines. The query region q intersects a triangle which is not in M, but does not intersect any of the *roi* boxes.[1]

For the applications described in Chapter 10 this is not a problem. However, for some applications it may be necessary to avoid this situation. In order to do so, we must ensure that every triangle $\triangle abc$ that is in an intermediate mesh is completely covered by the $roi(a) \cup roi(b) \cup roi(c)$. In this way, if the triangle intersects q, then so does one of $roi(a)$, $roi(b)$, $roi(c)$, and the triangle will not remain in the selectively refined mesh since one of the vertices of the triangle will have been split.

A simple way to achieve this is to augment the definition of *roi* so that for a vertex v, $roi(v)$ contains all triangles incident to v. If this approach is combined with Hoppe's method of forcing vertex splits (see Section 9.5.1), then the query region q will not intersect any triangle not in M, since every triangle not in M will be covered by the *roi* of each of its vertices. Unfortunately, this increases the running time of the selective refinement procedure on two counts. Firstly, since the size of the *roi*

[1]The same situation can occur in the selective refinement scheme described by Hoppe in [39].

increases, the likelihood that a query region will intersect them also increases, which increases the number of vertex splits which are reported. Secondly, using Hoppe's method of forcing vertex splits results in increased running times as well.

9.6 Empirical Results

In order to verify the viability of the bounding box approximation to the regions of influence used in our selective refinement algorithm, some empirical tests were performed on triangulated terrains (TINs). Tests were performed on random TINs as well as two real TINs, containing 10000 and 20000 points, each. Random TINs were generated by choosing uniformly distributed points in the unit square and then computing a Delaunay triangulation of these points. We measured and compared our results on random TINs with those on the two TINs, with the same number of vertices, and found that the results obtained were almost identical.

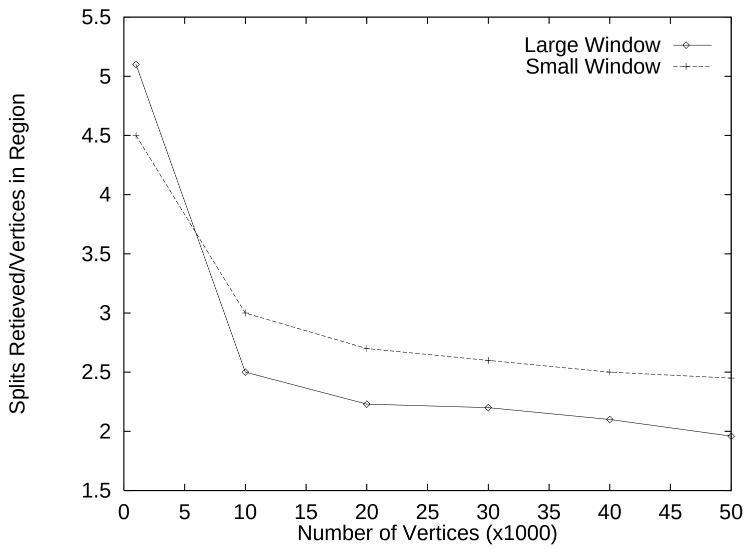

Figure 9.5: Performance of selective refinement algorithm for medium and large query regions.

Figure 9.5 shows the ratio between the number of vertices of the TIN, T, in a query region and the number of vertex splits which are retrieved when selectively refining the region. The tests take a query window of a fixed size and places it at 2500 regularly spaced locations on the TIN and perform selective refinement at each location. Both small (1/25 of the TIN's surface area) and large (1/4 of the TINs surface area) query windows were tested. The main results of these tests is that the ratio between the number of splits performed and the number of vertices in the query window converges to a small constant (< 3) as n increases.

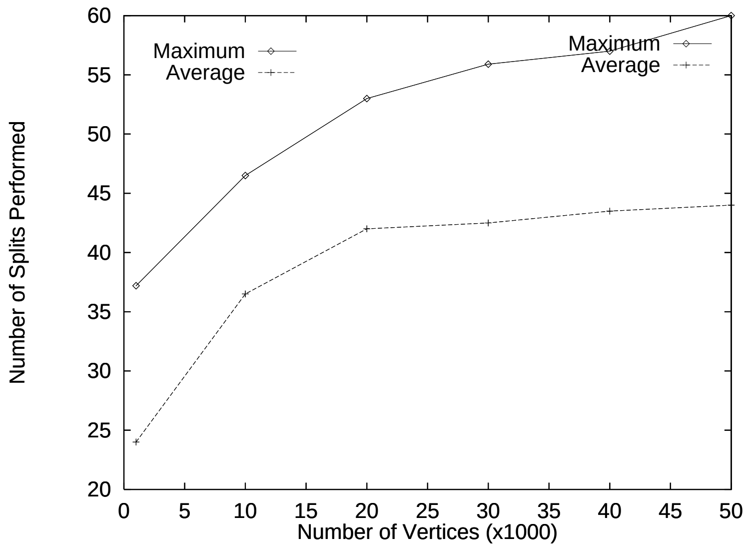

Figure 9.6: Performance selective refinement algorithm for small query regions.

Figure 9.6 shows results for a query region consisting of a single point on the surface of the TIN. Again, the query point is placed at 2500 regularly spaced locations on the TIN and selective refinement is performed. Figure 9.6 shows the results for TINs with up to 50000 vertices, and shows that even the worst-case running times tend to be logarithmic in the size of T. Also of interest are the absolute values in Figure 9.6 since these show that the constants are quite small. In no case does the number of vertex splits performed exceed 70.

When taken together, these two sets of experiments would suggest that the running time of the selective refinement algorithm is of the form $O(\log n + k)$, where k is the complexity of the TIN in the query region, and n is the number of vertices in the

TIN. Thus, at least empirically, the selective refinement procedure exhibits optimal behaviour up to small constant factors.

Chapter 10

Applications

The triangulated irregular network (TIN) is one of the basic models for representing geographical data where a triangulated set of points is stored together with its elevation. The TIN is essentially a mesh with the added constraint that no two points on it's surface have the same projection on the (x, y) plane. TINs have been introduced in 1978 (see [55, 56, 33]) and they are a fundamental data structure in GIS and related areas.

In this section, we describe some applications of our selective refinement scheme, with particular attention paid to applications to TINs. We show that meshes stored in the progressive mesh representation naturally supports a number of operations common in both computational geometry and geographic information systems. Because of the small constants involved in the PM representation, the algorithms presented in this section are competitive with existing algorithms which operate on meshes in a standard representation. The advantage of these algorithms over these existing algorithms are:

1. these algorithms work on meshes in the PM representation, maintaining all the advantages of the PM representation,

2. these algorithms require little or no additional preprocessing or storage beyond the progressive mesh construction process, making them, in some cases, more efficient than existing solutions, and

3. by using the "pruning" technique described in Section 9.4, all the algorithms presented here can be used to efficiently solve approximate versions of the problem in question, i.e., the problem can be solved on any of the meshes $M_0, \ldots, M_{|vsplits|}$.

Throughout the remainder of this chapter we use T when discussing a TIN or a planar triangulation in the same manner as we have previously used M to denote an arbitrary mesh.

10.1 Point Location

Point location in a triangulation is a well studied problem in computational geometry, and a number of algorithms exist. The point location problem consists of determining, given a planar triangulation, T, the face, f, in which a query point lies. Although theoretically optimal algorithms ($O(n)$ preprocessing time and $O(\log n)$ query time) exist for the point location problem, these algorithms have large constants hidden in the big-Oh notation which makes them less useful in practice. This is witnessed by the fact that most real world implementations use schemes that are less than optimal but which work well in practice [34, 50].

The selective refinement procedure described in Chapter 9 can be used as an efficient method for locating a point in a triangulation. The motivation for this application is twofold: (1) the selective refinement algorithm is fast in practice, and thus should yield a fast point location algorithm, and (2) if a triangulation is already in the progressive mesh representation then point location can be performed without any additional preprocessing.

To perform point location on a progressive mesh, we perform selective refinement in which the query region is a vertical line, namely the vertical line which passes through the query point q (see Figure 10.1). The selective refinement algorithm is run and the triangle in which q lies is found. The running time of this algorithm is clearly $O(|M_0| + kd)$ where k and d are defined as in Chapter 9.

If our objective is to do largely point location queries in a triangulation, then the fitness function that can be used in the construction algorithm to obtain a progressive

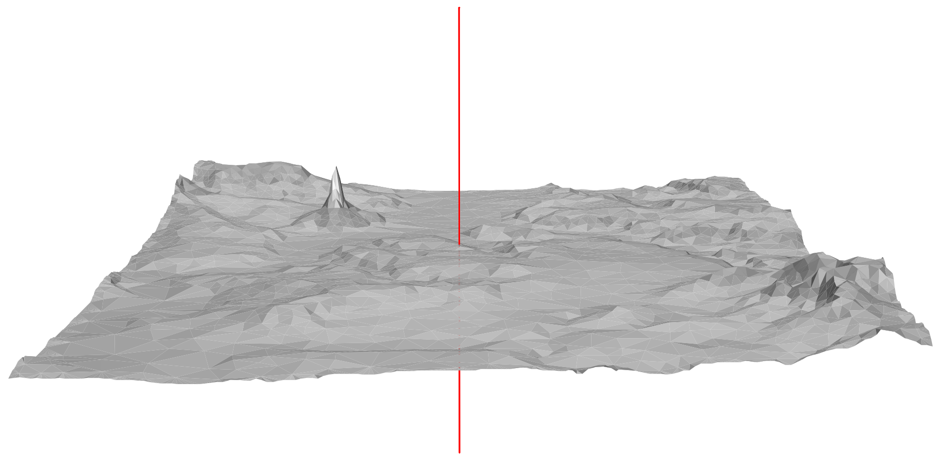

Figure 10.1: Expressing a point location query as a selective refinement query.

mesh is to collapse the edge (v_1, v_2) such that the resulting region of influence is the smallest among all possible edge collapses. A similar heuristic is used in the construction of R-Trees [37].

10.2 Isoline Extraction

The problem of isoline or contour line extraction is stated as follows: given a query elevation h, return all triangles of T which occur at elevation h. Given the triangles which occur at elevation h, the polygons and polygonal chains which form the isolines can be found in $O(k)$ time (where k is the number of triangles) by starting at an arbitrary triangle and walking along triangles until a triangle already visited is reached, or the border of T is reached. When this occurs, another unvisited triangle is chosen, and the same procedure is applied.

In, [62] van Kreveld presents an algorithm for isoline extraction based on the interval tree [24, 46]. An interval tree stores intervals on the real line and can answer queries of the form: find all intervals which overlap the query point q. By storing the minimum and maximum height of each triangle in an interval tree (along with a pointer to the triangle itself), isoline extraction reduces to interval tree queries.

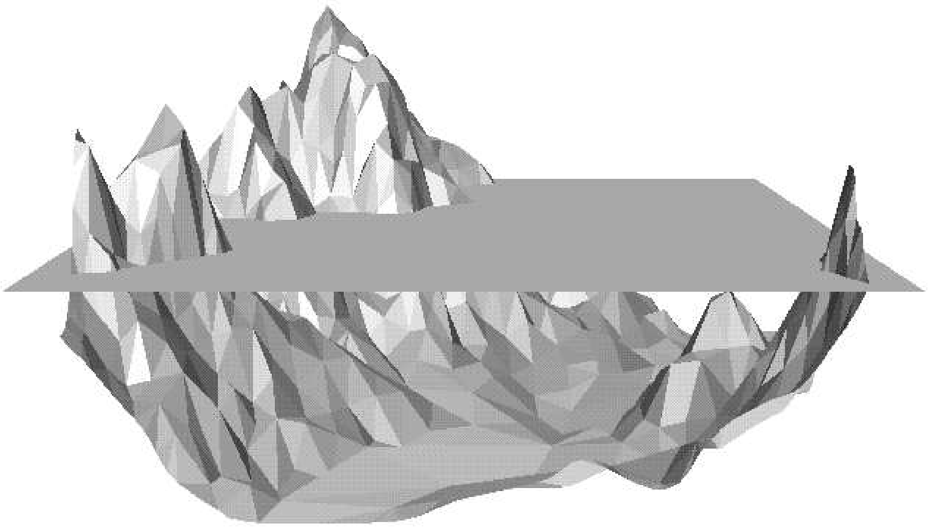

Figure 10.2: Expressing an isoline query as a selective refinement query.

Construction of the interval tree takes $O(n \log n)$ time, and queries take $O(\log n + k)$ time giving the preprocessing time and query time, respectively, for isoline extraction on a TIN represented in a standard representation.

For answering isoline queries on a progressive TIN, we simply perform selective refinement in which the query region is a level plane of height h, and report all triangles of T which intersect this plane. An example of this approach is shown in Figure 10.2. The running time of this algorithm is $O(|M_0| + kd)$ where k and d are defined as in Chapter 9.

From this, it follows that the edge collapse sequence can be optimized for isoline extraction by always collapsing the edge which produces the smallest interval in the resulting supervertex. This heuristic is similar to that suggested in Section 10.1 for optimizing the edge collapse sequence for point location.

Progressive TINs also support progressive transmission of isolines. To perform progressive transmission of isolines, we note that a vertex split operation can only affect isolines in the neighbourhood of the split vertex, and so maintaining isolines under the vertex split operation is a straightforward localized matter. Therefore, progressive transmission of isolines involves first transmitting the coarse grained TIN T_0 and then transmitting the vertex split records which affect the elevation(s) in question. At the receiving end, extracting the isolines from T_0 can be done using a linear time brute force method, and these isolines can be maintained using only local operations as new vertex splits records are received.

10.3 Visibility Queries

Two points on a TIN T are said to be visible if the line segment joining them does not intersect with T. The visibility query problem is the following: given two query points p and q, is p visible from q?

A straightforward solution to the visibility query problem is as follows. Locate the triangle in which p lies and then walk along triangles in the direction of q until either (1) an edge is crossed which occludes q in which case the answer is negative, or (2) the triangle containing q is reached in which case the answer is positive. The preprocessing required by this algorithm is the same as that for point location. The query time is the time to locate q plus the number of edges, k, which intersect the segment $\overline{pq}$.

On a progressive TIN, visibility between a pair of points can be answered by selectively refining the TIN along the (3 dimensional) line segment $\overline{pq}$. In many cases, this will lead to a faster query time. If p and q are visible, then the line segment $\overline{pq}$ does not properly intersect T. In this case $\overline{pq}$ is not likely to intersect many of the regions of influence of the vertex splits, and the selective refinement procedure will finish very quickly indeed. In p and q are not visible, then $\overline{pq}$ may intersect many of the regions of influence of vertex splits, but the selective refinement procedure can be stopped the first time a triangle of T is found which intersects $\overline{pq}$. See Figure 10.3 for an example.

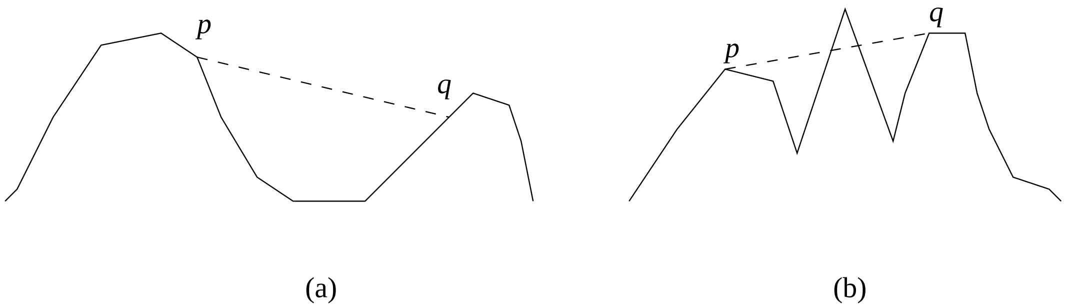

Figure 10.3: An example of two visbility queries on a TIN. In case (a), the selective refinement procedure will not do any refinement in the valley between p and q. In case (b), the selective refinement procedure will determine very quickly that visbility is blocked by the peak between p and q.

10.4 External Memory Progressive Meshes

External memory methods for meshes are of significant practical relevance to the field of GIS, as the amount of geographic data currently available exceeds the capacity of internal memories. This motivates the development of an external memory storage scheme for progressive meshes. With such a scheme, the coarse grained mesh, M_0, could be stored in internal memory, and user could perform selective refinement to refine a small portion of this mesh and with it. A number of external memory spatial data structures exist which can be used to store progressive TINs in the external memory. We believe that the R-Tree [37], or one of it's variants, is a data structure well suited for this application. R-trees are designed specifically for storing axis aligned boxes. (A variety of other possible spatial index structures exist see e.g. [60, 35, 36, 43] and also consult the new survey article by Nievergelt and Widmayer [53].) In particular, one variant of the R-tree, the packed R-tree constructs an R-tree in a bottom up fashion from a static set of axis aligned boxes.

Thus in order to construct an external memory representation of a progressive mesh, build a packed R-tree on the list of vertex splits, where the box associated with a vertex split v is $roi(v)$. When performing selective refinement we use the packed R-tree to extract the relevant vertex splits and then proceed in the manner described in Chapter 9 to sort and apply the splits.

Using this representation, the algorithms for point location, elevation queries, and visibilities queries can all be applied to TINs stored in the PM representation in external memory.

Chapter 11

Conclusions

In this part of the thesis, we have extended Hoppe's progressive mesh representation so that it efficiently supports selective refinement. Using this selective refinement scheme a number of algorithms for fundamental problems in computational geometry and geographic information systems have been devised.

Perhaps as important as the refinement scheme itself are some of the ideas behind it. Augmenting the DAG-based hierarchies of Dobrindt and de Berg [16] and Puppo [58] with the *roi* information described in Chapter 9 could be beneficial to these schemes as well. For one, it would allow the type of exact selective refinement described in Chapter 9 (as opposed to the heuristic computer graphics oriented methods currently used), and could improve the performance of these schemes by replacing complicated point-in-triangle tests with the much simpler point-in-rectangle tests. Of course, an empirical study would be needed to determine whether the increase in performance due to the simpler tests is enough to offset the increase in work caused by the bounding box approximation.

It is also worth noting that algorithms in Chapter 10 which were described for the progressive mesh could also be used with other multiresolution surface models. For this to work, these models would need a method of exact selective refinement like the one described in Chapter 9. Since such methods can be designed by using analagous techniques, namely the *roi* information, the algorithms in Chapter 10 actually solve a number of problems on most multiresolution surface models.

The field of multiresolution surface modelling is full of interesting challenges and opportunities. Any problem related to surface models can be attempted in multiresolution models as well. It may be possible to use the multiresolution aspect of the model to advantage. One obvious approach is to provide faster approximation algorithms by simply running the standard algorithm on a mesh with a lower level of detail. In this case, it is important to bound the quality of the resulting approximation, either theoretically or empirically. One could also imagine using these models to develop interruptible algorithms, i.e., algorithms that can be interrupted at any time to produce an approximate result but which will eventually converge to an exact solution given enough time. This could be done by finding an exact solution on the coarse mesh M_0 and improving it by increasing the resolution of the mesh in certain areas.

Bibliographic Note

The work described in this part of the thesis is joint work with Jörg-Rüdiger Sack and Anil Maheshwari and has appeared in a preliminary form in the proceedings of the 5th ACM Workshop on Geographic Information Systems [44].

Bibliography

[1] M. Adler, P. B. Gibbons, Y. Matias, and V. Ramachandran. Modelling parallel bandwidth: Local vs. global restrictions. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–105, 1997.

[2] D. Bader, D. Hellman, and J. JáJá. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, 1996.

[3] A. Bäumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 233–242, 1996.

[4] A. Bäumker, W. Dittrich, F. Meyer auf def Heide, and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Proceedings of Euro-Par '96*, pages 27–29, 1996.

[5] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the European Symposium on Algorithms*, pages 17–30, 1995.

[6] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. Technical report, University of Paderborn, 1996.

[7] A. Bäumker, W. Dittrich, and A. Pietracaprina. The deterministic complexity of parallel multisearch. In *Proc. 5th SWAT*, 1996.

[8] A. Beguelin, J. Dongarra, A. Geist, B. Manchek, and V. Sunderam. Solving computational grand challenges using a network of heterogeneous supercomputers. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 596–601, 1991.

[9] L. Boxer, R. Miller, and A. Rau-Chaplin. Some scalable parallel algorithms for geometric problems. In *Proceedings of 8th International Conference on Parallel and Distributed Computing and Systems (PDCS '96)*, 1996.

[10] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Proceedings of International conference on Automata, Languages, and Programming*, 1997.

[11] A. L. Cheung and A. P. Reeves. High performance computing on a cluster of workstations. In *Proceedings of 1st International Symposium on High Performance Distributed Computing*, pages 152–160, 1992.

[12] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *Proceedings of 4th International Symposium on High Performance Distributed Computing*, 1995.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[14] P. E. Crandall and M. J. Quinn. A decomposition advisory system for heterogeneous data-parallel processing. In *Proceedings of 3rd International Symposium on High Performance Distributed Computing*, pages 114–121, 1994.

[15] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Snatos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *ACM Symposium on Principles and Practices of Parallel Programming*, pages 1–12, 1993.

[16] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *ACM Symposium on Computational Geometry*, pages ??–??, 1995.

[17] L. de Floriani. A pyramidal data structure for triangle based surface description. *IEEE Computer Graphics and Applications*, 9:67–78, 1989.

[18] L. de Floriani, P. Magillo, E. Puppo, and M. Bertolotto. Variable resolution operators on a multiresolution terrain model. In *4th ACM Workshop on Advances in Geographic Information Systems*, pages 123–130, 1996.

[19] L. de Floriani, D. Mirra, and E. Puppo. Extracting contour lines from a hierarchical surface model. In *Eurographics '93*, pages 249–260, 1993.

[20] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Kokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, 1995.

[21] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proceedings of ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[22] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International journal on Computational Geometry*, 6(3):379–400, 1996.

[23] A. Diallo, A. Ferreira, A. Rau-Chaplin, and S. Ubéda. Scalable 2d convex hull and triangulation algorithms for coarse grained multicomputers. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

[24] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical report, Technical University of Graz, 1980.

[25] A. Ferreira and S. Ub'eda. Computing the medial axis transform with 8 scan operations. In *IEEE International Conference on Image Processing*, 1995.

[26] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a terrain at variable resolution. Technical Report PDISI-97-02, University of Genova, 1997.

[27] L. De Floriani, P. Magillo, and E. Puppo. Variant - processing and visulazing terrains at variable resolution. In *Proceedings of the 5th International Workshop on Geographic Information Systems (ACM-GIS'97)*, pages 15–19, 1997.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[29] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications to computational geometry. In *Proceedings of EuroPar'96*, 1996.

[30] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications to computational geometry. Technical report, Oxford Computing Laboratory, 1996.

[31] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the bulk-synchronous parallel model with applications to priority queues. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96)*, 1996.

[32] A. V. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. In *3rd Scandinavian Workshop on Algorithm Theory*, pages 1–18, 1992.

[33] C. M. Gold. The practical generation and use of geographic triangular element data. *Harvard Papers on Geographic Information Systems*, 5:??–??, 1978.

[34] M. T. Goodrich, M. Orletsky, and K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *ACM Symposium on Discrete Algorithms*, pages ??–??, 1997.

[35] O. Guenther. The design of the cell tree: an object-oriented index structure for geometric databases. In *Proceedings of the 5th IEEE Conference on Data Engineering*, pages 598–615, 1989.

[36] R. H. Güting. An introduction to spatial database systems. *The VLDB Journal*, 3:357–399, 1994.

[37] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference 1984*, pages 47–57, 1985.

[38] H. Hoppe. Progressive meshes. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 99–108, 1996.

[39] H. Hoppe. View-dependent refinement of progressive meshes. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, page to appear, 1997.

[40] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuezle. Mesh optimization. *siggraph95*, pages 247–254, 1995.

[41] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[42] D. Knuth. *The Art of Computer Programming – Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[43] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, London, 1992.

[44] A. Maheshwari, P. Morin, and J.-R. Sack. Progressive tins: Algorithms and applications. In *Proc. 5th ACM Symposium on Geographic Information Systems*, 1997.

[45] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[46] E. McCreight. Priority search trees. *SIAM Journal on Computing*, 14:257–276, 1985.

[47] P. Morin. *The PLEDA User's Guide*. Carleton University, 1.0 edition, 1997.

[48] P. Morin. Coarse grained parallel computing on heterogeneous systems. In *To appear in Proc. 13th Annual ACM Symposium on Applied Computing (SAC'98)*, 1998.

[49] S. Näer. The LEDA manual. Technical Report MPI-I-93-109, Max-Planck Institut für Informatik, 1993.

[50] S. Näher. The leda manual, version r-3.4.1. Technical report, Max-Planck Institut, 1996.

[51] N. Nedeljković and M. J. Quinn. Data parallel programming on a network of heterogeneous workstations. *Concurrency: Practice and Experience*, 5(4):257–268, June 1993.

[52] M. V. Nibhanupudi, C. D. Norton, and B. K. Szymanski. Plasma simulation on networks of workstations using the bulk-synchronous parallel model. In *International Conference on Parallel and Distributed Techniques and Applications*, 1995.

[53] J. Nievergelt and P. Widmayer. Spatial data structures: concepts, design, and choices. In J. Urrutia and J.-R. Sack, editors, *Handbook of Computational Geometry*. Elsevier Sciences, to appear.

[54] S. Orlando and R. Perego. A template for non-uniform parallel loops based on dynamic scheduling and prefetching techniques. In *Proceedings of the 10th ACM International Conference on Supercomputing*, 1996.

[55] T. K. Peucker. Data structures for digital terrain models. *Harvard Papers on Geographic Information Systems*, 5:??–??, 1978.

[56] T. K. Peucker, R. J. Fowler, J. J. Little, and D. M. Mark. The triangulated irregular network. In *Proceedings DTM Symposium American Society of Photogrammatry - Amercian Congress on Survery and Mapping*, pages 24–31, 1978.

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

[58] E. Puppo. Variable resolution terrain surfaces. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, 1996.

[59] J. Reif, editor. *Synthesis of parallel algorithms.* Morgan Kaufmann, 1993.

[60] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[61] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[62] M. van Kreveld. Efficient methods for isoline extraction from a digital elevation model based on triangulated irregular networks. Technical report, University of Utrecht, 1994.

[63] H. A. G. Wijshoff and B. H. H. Juurlink. A quantitative comparison of parallel computation models. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 13–24, 1996.

[64] M. H. Willebeck-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[65] Y. Yan, X. Zhang, and Y. Song. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, 1996.

[66] X. Zhang and Y. Yan. Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 25–34, 1995.

Appendix A

High Probability Bounds

When discussing randomized algorithms of any type it is usually not enough to give the expected running time of the algorithm. The reason for this is that, while the expected running time may be good, any particular execution of the algorithm may take significantly longer.

This where high probability bounds are used. The idea behind such bounds is to show that the execution time of an algorithm rarely differs significantly from its expected running time, at least in the case where the input size is large. The definition of high probability is usually applied to a random variable. However, in order to emphasize its application, we describe it in terms of the running time of a randomized algorithm.

Definition 1. *We say that a randomized algorithm $\mathcal{A}$ has running time $O(f(n))$* with high probability*, denoted $\tilde{O}(f(n))$ if*

$$\Pr\left[\textit{Execution time of } \mathcal{A} \geq cf(n)\right] \leq \frac{1}{poly(n)}$$

where $poly(n)$ is a polynomial whose degree depends on c.

At this point it is worth noting that no assumption is made about the input to $\mathcal{A}$, and so $\mathcal{A}$ must rely strictly on the randomization to achieve this running time.

A.1 The Binomial Distribution

A *Bernoulli trial* is any experiment with only two possible outcomes, success or failure. In such a trial we define q as the probability of success and $1-q$ as the probability of failure. If we denote success by 1 and failure by 0 we see that a Bernoulli trial X is defined by

$$\Pr[X=1]=q \text{ and } \Pr[X=0]=1-q.$$

If we define $X(n)$ as the number of successes during n Bernoulli trials with success probability q then we have

$$\Pr[b(n,q)=k]=C(n,k)q^k(1-q)^{n-k}.$$

We say that $b(n,q)$ follows a *binomial distribution* with success probability q and number of trials equal to n.

Binomial distributions occur frequently when giving high probability bounds for randomized algorithms. The nice property of the binomial distribution is that if the running time of an algorithm follows the distribution $b(n,q)$, then for certain values of n and q, the running time of the algorithm is $\tilde{O}(nq)$. In the next two sections we present some tools that help in proving such results.

A.2 Chernoff Bounds

This section discusses Chernoff bounds for bounding a binomial distribution from above and below, and discusses their applications to high probability bounds. Because it is unclear from just looking at the definition, how Chernoff bounds apply to high probability bounds, an abstract high probability bound is derived as an illustration.

Theorem 15 (Chernoff Bounds). *Let X be a random variable which follows the distribution $b(n,q)$, then*

$$\Pr[X\geq cqn]\leq\left(\frac{1}{e}\right)^{(c-1)^2qn/3}, \textit{ for } c>1 \tag{A.1}$$

and

$$\Pr[X\leq cqn]\leq\left(\frac{1}{e}\right)^{(1-c)^2qn/2}, \textit{ for } c<1. \tag{A.2}$$

Consider how Equation A.1 might be used to prove high probability bounds for a randomized parallel algorithm $\mathcal{A}$. Usually q is the probability that an input element is assigned to a specific processor, P_i, the value n corresponds to the problem size, and the value c corresponds to the constant in the $\tilde{O}$ notation. Thus we wish to prove that X, the number of elements received by P_i, is $\tilde{O}(qn)$.

In order to achieve this we must place some restrictions on the values of q, and n. In particular, we choose $qn \geq 3 \ln n$ and Equation A.1 becomes

$$\Pr\left[X \geq cpn\right] \leq \frac{1}{n^{(c-1)^2}}, \text{ for } c > 1, \text{ and } qn \geq 3 \ln n, \tag{A.3}$$

giving us the desired result. We say that $X = \tilde{O}(qn)$, for $qn \geq 3 \ln n$.