CS--1994--22

Efficient Parallel Algorithms for Closest Point Problems

Peter Su

Dapartment of Mathematics
and Computer Science

Dartmouth College

Hanover, New Hampshire

and

Department of Computer Science

Duke University

Durham, North Carolina 27708-0129

June 1994

Efficient Parallel Algorithms for Closest Point Problems

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

by

Peter Su

Dartmouth College

Hanover, New Hampshire

June, 1994

Examining Committee:

____

(chairman) Scot Drysdale

____

John Reif

____

Tom Cormen

____

David Kotz

____

____

Dean of Graduate Studies

Efficient Parallel Algorithms for Closest Point Problems

by Peter Su, Ph.D.
Thesis Advisor: Scot Drysdale

Department of Mathematics
and Computer Science
Dartmouth College

ABSTRACT OF THE THESIS

This dissertation develops and studies fast algorithms for solving closest point problems. Algorithms for such problems have applications in many areas including statistical classification, crystallography, data compression, and finite element analysis. In addition to a comprehensive empirical study of known sequential methods, I introduce new parallel algorithms for these problems that are both efficient and practical. I present a simple and flexible programming model for designing and analyzing parallel algorithms. Also, I describe fast parallel algorithms for nearest-neighbor searching and constructing Voronoi diagrams. Finally, I demonstrate that my algorithms actually obtain good performance on a wide variety of machine architectures.

The key algorithmic ideas that I examine are exploiting spatial locality, and random sampling. Spatial decomposition provides allows many concurrent threads to work independently of one another in local areas of a shared data structure. Random sampling provides a simple way to adaptively decompose irregular problems, and to balance workload among many threads. Used together, these techniques result in effective algorithms for a wide range of geometric problems.

The key experimental ideas used in my thesis are simulation and animation. I use algorithm animation to validate algorithms and gain intuition about their behavior. I model the expected performance of algorithms using simulation experiments, and some knowledge as to how much critical primitive operations will cost on a given machine. In addition, I do this without the burden of esoteric computational models that attempt to cover every possible variable in the design of a computer system. An iterative process of design, validation, and simulation delays the actual implementation until as many details as possible are accounted for. Then, further experiments are used to tune implementations for better performance.

Part of this work was at atttt the Department of Computer Science, Duke University, Durham, NC 27708-0129 and was supported by ARPA/ISTO Grant N00014-91-J-1985, Subcontract KI-92-01-0182 of ARPA/ISTO prime Contract N00014-92-C-0182. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied of the Advanced Research Pojects Agency, NSF, ONR or the U.S. government.

Acknowledgements

I have noticed in my years of reading other people's dissertations that the best ones also seem to have the most interesting and entertaining acknowledgements. I can only hope that the quality of mine is close to many others that I have read and learned so much from.

To begin with, my family, especially my father, always let me pursue ideas and activities that interested me. They bought me the magazines and a telescope that introduced me to science. What they taught me gave me the ambition to continue my education into graduate school and the tools to succeed here.

I have the distinct pleasure to be able to thank three entirely separate groups of people for my eventual success as a graduate student. First, at Dartmouth, Scot Drysdale was my advisor, and taught me everything I know about Voronoi diagrams. His eye for detail, and interest in real answers made the work more comprehensive than it would have been. Finally, his knack for finding clear, simple explanations for complicated algorithms improved the presentation of almost every section of the thesis.

My fellow students at Dartmouth, all of whom graduated before me, included Barry Shaudt, Deb Bannerjee, John Vanmeter, Joe and Alisa Destefano, Jerry Quinn and of course, Larry Raab. Thanks guys, for the parties, office sharing, rooms at the "inn", help with Bent's problem sets, heated and generally worthless political discussions, and for putting up with my abuse when you asked me UNIX questions. And speaking of UNIX, Wayne Cripps, under great duress, somehow always kept the rubber bands and glue that was room 5 together and running.

Next, at Duke University, John Reif provided more advising, an endless pool of algorithmic ideas, and the monetary and machine support that I needed to finish my projects. Without John's interest in combining theoretical and experimental ideas in computational geometry, my work on parallel algorithms never would have materialized. In addition, all the folks at the North Carolina Supercomputer Center, and especially Greg Byrd, must be thanked for keeping `flyer` and `hornet` running and putting up with my constant stream of novice questions.

The whole alternative happy hour crowd quickly made me part of the Duke community, after relieving their initial suspicion that I might be a "theory weenie." Vick Khera provided numerous UNIX toys that were fun to play with, and sometimes helped me finish. Owen Astrachan wasted hours debating various religious issues, especially programming languages. Steve Tate put up with my constant badgering of theoreticians and my clueless theory questions. Mike Landis, Deganit Armon, Eric Anderson, and Chris Connelly provided friendship and great conversation.

Thirdly, old friends at CMU got me started on all of this when I was an undergrad, and gave me a place to finish the final stretch of thesis writing. Thanks to David Garlan and Rob Chandhok for all their past and future help, and thanks to Guy Blelloch, Sid Chatterjee and the whole SCANDAL group for being interested in me and for their great library of Cray magic.

Many net.hackers provided the programming tools that I used every day to get this done. Where would the world be without `emacs`, `ange-ftp-mode`, `mh`, `vm`, `gnus`, `groff`, LaTeX, `ghostview`,

`gcc`, `perl`, `tcsh` and all the rest.

Lastly, and most importantly, I have to thank Karen for the last six years of love, understanding and never-ending support. It's all over now kiddo, you might never have to hear about Delaunay triangulations again.

Table of Contents

Abstract . . . iii
Acknowledgements . . . iv
Table of Contents . . . vi
List of Tables . . . ix
List of Figures . . . x
1. Introduction . . . 1
1.1. Preliminaries . . . 2
1.1.1. Workload . . . 3
1.1.2. Models . . . 4
1.1.3. Machines . . . 6
1.1.4. Other Notation and Terminology . . . 6
1.2. Related Work . . . 6
1.2.1. Theory . . . 6
1.2.2. Practice . . . 9
1.3. Contributions . . . 9
1.4. Outline of the Thesis . . . 11
2. Practical Sequential Voronoi Algorithms . . . 12
2.1. Divide-and-Conquer . . . 12
2.2. Sweepline Algorithms . . . 18
2.3. Incremental Algorithms . . . 20
2.4. A Faster Incremental Construction Algorithm . . . 24
2.5. Incremental Search . . . 27
2.5.1. Site Search . . . 28
2.5.2. Discussion . . . 30
2.6. Empirical Results . . . 31
2.6.1. Performance of the Incremental Algorithm . . . 31
2.6.2. The Incremental Algorithm with a Quad-Tree . . . 33
2.6.3. The Incremental Search Algorithm . . . 34
2.7. Fortune's Algorithm . . . 36
2.8. The Bottom Line . . . 39
2.9. Nonuniform Point Sets . . . 40
2.10. Notes and Discussion . . . 44
2.10.1. Other Algorithms . . . 47
2.11. Principles . . . 47

3. Models and Machines . . . 49
3.1. Popular Programming Models . . . 50
3.2. What is wrong with PRAM . . . 51
3.3. Our Model . . . 52
3.4. Other Models . . . 53
3.5. Machine Descriptions . . . 54
3.5.1. The Cray Y-MP . . . 55
3.5.2. The KSR-1 . . . 56
3.6. Benchmarking . . . 57
3.6.1. The Cray . . . 57
3.6.2. The KSR-1 . . . 58
3.7. Summary . . . 61
4. Concurrent Local Search . . . 62
4.1. The Problem . . . 62
4.2. The Algorithm . . . 62
4.3. All Nearest Neighbors on the Cray . . . 63
4.4. Measurements . . . 69
4.5. Extensions and Applications . . . 71
4.6. Discussion . . . 72
4.7. Implementation for the KSR-1 . . . 73
4.8. Summary . . . 78
5. Parallel Delaunay Triangulation Algorithms . . . 79
5.1. Parallel Divide and Conquer . . . 79
5.2. Optimal Expected Time Algorithms . . . 80
5.3. Randomized Algorithms . . . 82
5.4. Preliminaries: The General Framework . . . 83
5.5. Randomized Divide and Conquer . . . 84
5.6. A More Practical Randomized Parallel Algorithm. . . . 87
5.7. Analysis . . . 91
5.8. Experiments . . . 92
5.9. Summary . . . 97
6. Practical Parallel Algorithms for Delaunay Triangulations . . . 99
6.1. Randomized Incremental Construction . . . 99
6.2. Concurrent Incremental Construction . . . 101
6.3. Simulations . . . 105
6.4. The Implementation . . . 108
6.5. Concurrent Incremental Search . . . 117
6.6. More Experiments . . . 117
6.7. Summary . . . 122
7. Summary and Future Work . . . 126
7.1. Algorithms . . . 126
7.2. Models and Experimental Analysis . . . 127

7.3. Implementations and Benchmarks 128
7.4. Future Work 128
7.4.1. Algorithms. 128
7.4.2. Programming Parallel Algorithms. 129
7.4.3. Simulation and Performance Analysis. 130
References 132

List of Tables

2.1. Nonuniform distributions. 42
3.1. Costs for vector operations on one processor of the Cray Y-MP 57
3.2. Cost for elementwise addition on the KSR-1. 59
4.1. Comparison of synthetic benchmark values to actual runtime. 75

List of Figures

2.1. The quad-edge data structure. 13
2.2. The quad-edge representation of a simple subdivision. 14
2.3. The merge procedure in Guibas and Stolfi's divide and conquer algorithm. 15
2.4. The rising bubble. 16
2.5. The merge loop. 17
2.6. Code for processing events in Fortune's algorithm 18
2.7. Invalidating circle events. 19
2.8. Fortune's algorithm. 21
2.9. Pseudo-Code for the incremental algorithm. 22
2.10. The incremental algorithm. 23
2.11. The incremental algorithm with spiral search for point location. 25
2.12. Spiral search for a near neighbor. 26
2.13. Algorithm IS. 27
2.14. Code for Site Search. 29
2.15. Example bounding boxes that the site search algorithm examines. 29
2.16. Site search in Algorithm IS 30
2.17. Comparison of point location costs. 32
2.18. Circle tests per site for two algorithms. 33
2.19. Performance of the quad-tree algorithm. 34
2.20. Performance of Algorithm IS 35
2.21. The cost of Fortune's algorithm. 36
2.22. Circle events cluster near the sweepline. 37
2.23. The cost of Fortune's algorithm with a heap. 38
2.24. Delaunay edges imply empty regions of the plane. 39
2.25. Comparison of runtimes. 41
2.26. The incremental algorithm on non-uniform inputs, n is 10K. 43
2.27. Fortune's algorithm on non-uniform inputs, n is 10K. 43
2.28. Dwyer's algorithm on non-uniform inputs, n is 10K. 44
2.29. Algorithm IS is very sensitive to bad inputs. 45
2.30. Algorithm IS the "cluster" and "corners" distributions. 45
2.31. Runtimes on non-uniform inputs. 46
3.1. Schematic diagram of the Cray Y-MP. 55
3.2. Schematic diagram of the KSR-1 56
3.3. Performance of elementwise vector addition on the KSR-1. 58
3.4. Performance of parallel prefix sum on the KSR-1. 59
3.5. Performance of unstructured permute on the KSR-1. 60
3.6. Performance of reverse on the KSR-1. 61

4.1. Parallel loop to bucket points. . . . 64
4.2. The bucket data structure. . . . 64
4.3. Handling write collisions. . . . 66
4.4. Spiral search to find a near neighbor. . . . 67
4.5. The outer loop of the spiral search algorithm. . . . 67
4.6. The second outer loop of the spiral search algorithm. . . . 68
4.7. The inner loop of the spiral search algorithm. . . . 68
4.8. Distance computations per site for each algorithm. . . . 70
4.9. Runtime of the spiral search algorithm. . . . 70
4.10. Runtime on the Cray 71
4.11. Cray vectorized runtime on clusters 72
4.12. Threads code for bucketing points. . . . 74
4.13. Inner loop of the spiral search. . . . 75
4.14. Performance of the synthetic benchmark. . . . 76
4.15. Performance of the concurrent local search algorithm on the KSR-1. . . . 76
4.16. Speedup of the parallel spiral search algorithm. . . . 77
4.17. The parallel algorithm compared to a Sparcstation. . . . 78
5.1. Objects, regions and the conflict relation. . . . 84
5.2. The divide step in Ghouse and Goodrich's algorithm. . . . 86
5.3. The routine `find-circles`. . . . 89
5.4. The triangle ABC fails a circle test, so p goes to A's subproblem. . . . 90
5.5. The Voronoi region of a point r, and the "flower" of circles. . . . 91
5.6. Load balancing results for 32 processors. . . . 93
5.7. Load balancing for 128 processors. . . . 94
5.8. Bucket expansion with bad distributions for 32 processors. . . . 94
5.9. Total subproblem size for Algorithm RDDT. . . . 96
5.10. Total subproblem size by distribution for 32 processors. . . . 96
6.1. The concurrent incremental algorithm. . . . 103
6.2. Load balancing in the concurrent incremental algorithm. . . . 106
6.3. Concurrency Profiles for various machine and problem sizes. . . . 107
6.4. Circle tests per site for Algorithm CRIC. . . . 109
6.5. Edge tests per site for Algorithm CRIC. . . . 110
6.6. Interchanging loops to use GSS. . . . 112
6.7. The effect of large batch sizes for 16 and 32 processors. . . . 113
6.8. Circle tests per site for Algorithm CRIC. . . . 114
6.9. Edge tests per site for Algorithm CRIC. . . . 115
6.10. Speedup of Algorithm CRIC vs. One KSR node. . . . 115
6.11. Algorithm CRIC vs. a Sparc 2. . . . 116
6.12. Code for a concurrent edge dictionary. . . . 118
6.13. Code for the concurrent edge queue. . . . 119
6.14. Distance calculations per site for the concurrent incremental search algorithm. . . . 120
6.15. Buckets examined per site for the concurrent incremental search algorithm. . . . 121
6.16. Speedup of Algorithm CIS relative to one KSR node. . . . 121
6.17. Speedup of Algorithm CIS relative to the SparcStation 2. . . . 122
6.18. Algorithm CIS vs. a Sparc 2. . . . 123

6.19. Speedup of Algorithm CIS relative to Algorithm CRIC on 32 procs. 123

Chapter 1
Introduction

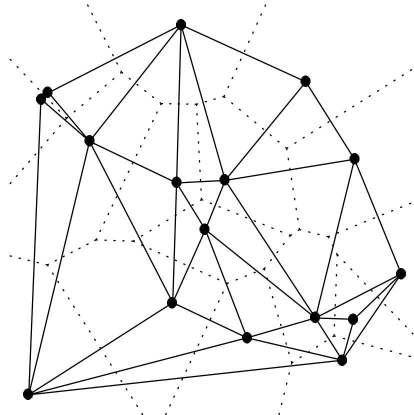
In creating, the only hard thing
is to begin; A grass-blade's no
easier to make than an oak.
—James Russell Lowell

This thesis develops parallel algorithms for closest point problems in computational geometry. Specifically, we will concentrate on the all-nearest-neighbors problem and constructing the Voronoi diagram. These problems are easy to state:

All-nearest-neighbors. Given a set S of n points in space, and a set query points Q, for each $p \in Q$, find the point $s \in S - \{p\}$ that is closest to p.

Voronoi diagram. Let S be a set of n points (or **sites**). in space. For each $s \in S$ define $V(s)$ to be a collection of points such that for all $x \in V(s)$ and $y \in S - \{s\}$, $dist(x, s) \leq dist(x, y)$. The Voronoi diagram of S is the collection $\{V(s) \mid s \in S\}$.

The straight-line dual of the Voronoi diagram is called the Delaunay triangulation. The following picture shows the Voronoi diagram and the Delaunay triangulation of 16 points in the plane:



In d dimensions, our algorithms will assume that no set of more than $d + 1$ sites are co-spherical. In this case, the Delaunay triangulation partitions the convex hull of S into simplices, and the circumsphere of each simplex contains no site. For example, in the planar case, we assume that no four sites are co-circular, so the convex hull of the point set is divided into triangles, and each triangle has an empty circumcircle.

This dissertation will present four case studies that study the performance of sequential and parallel algorithms for the planar version these problems. These case studies will provide an experimental comparison of known sequential algorithms, and will also present new parallel algorithms for both problems.

Algorithms for closest point problems make up a large part of the literature in computational geometry. Preparata and Shamos [PS85] provide an excellent overview of this field in their textbook. Theoretical research in this area has produced many algorithms that are not only optimal in a theoretical sense but also efficient in practice.

These problems have also interested many outside the field of theoretical computer science. Voronoi diagrams have been used in many applications such as statistical classification, crystallography, data compression, and finite element analysis. Nearest neighbor search appears as a problem in database systems, clustering algorithms, image processing, and combinatorial optimization. Aurenhammer [Aur91] and Okabe, et al [OBS92] survey algorithms and applications related to these problems.

Parallel computational geometry is the branch of computational geometry that is concerned with developing geometric algorithms that are suitable for computers with multiple processors. Since parallel computers can, in theory, allow users to solve large problems much more quickly than conventional machines, parallel algorithms for geometric problems are of interest as potential applications get larger. Over the last decade or so, dozens of parallel algorithms for various kinds of machines have been proposed, but very few have been successfully implemented. Moreover, few implementations have delivered reasonable performance. This is primarily due to the use of unrealistic computational models and high constant factors hidden by asymptotic analysis [Nat90].

This relative lack of concern for practical issues in parallel algorithms has opened a gap between researchers in theoretical computer science and practicing programmers. Theoretical computer scientists tend to argue that incorporating architecture specific parameters into their models will make a general theory difficult to obtain. Practitioners in parallel computing argue that the computational models (i.e. PRAM) used in theoretical computer science are unrealistic and thus irrelevant.

The thesis motivating this dissertation is that neither of these two views is correct. An understanding of both theoretical insights and machine details is necessary to develop efficient and practical parallel algorithms for important problems. This dissertation will show how to use the tools of theory (analysis of algorithms and probability theory) and systems (compilers, programming languages, operating systems and architecture) to design parallel algorithms for closest point problems on a variety of architectures.

1.1 Preliminaries

The goal of this dissertation is to present efficient algorithms for finding all-nearest-neighbors and constructing Voronoi diagrams on conventional and parallel architectures. By *efficient*, we do not mean asymptotically efficient in some abstract model of parallel computation, nor do we mean efficient in terms the rate at which the algorithms perform floating point operations, or simply the speedup achieved by the parallel algorithms when using multiple processors. Ideally, the algorithms should achieve runtimes that are competitive with known practical solutions to these problems, and at the same time make effective use of multiple processors when they are available.

In order to achieve this goal, we will use a broad range of tools from the areas of algorithm design, sequential and parallel programming methodology, performance analysis and experimental

computer science. Some of the specific techniques that will show up in the course of this dissertation include:

- The study of algorithms with efficient *expected* runtimes when the input to the program is assumed to come from some known probability distribution.
- The use of randomization in sequential and parallel algorithms design.
- The use of high level programming models and data structures to abstract algorithms away from machine architectures.
- The combination of high level programming models with explicit cost models to keep algorithm designs realistic.
- The use of simulation and animation of sequential and parallel algorithms to systematically analyze an implementation's behavior before time-consuming programming takes place on a large-scale machine.

Chapter 2 discusses these principles in more detail in the context of sequential algorithms, while Chapters 5 and 6 apply the techniques in the context of parallel algorithms design.

The key idea that this dissertation tries to emphasize is the use of a systematic methodology to study parallel algorithms from both a theoretical and experimental viewpoint. Both Bentley [Ben91] and McGeoch [McG86] have presented similar methodologies for the experimental study of sequential algorithms. This thesis will present case studies that combine careful experiments with new or known theoretical results into a coherent understanding of the performance of proximity algorithms on sequential, vector, and parallel architectures.

The experimental framework consists of three parts: workloads, models, and machines.

1.1.1 Workload

Workload is a term used in experimental computer science to describe a suite of programs, real or synthetic, that are used to test the behavior of existing systems and guide the design new ones.

In the context of algorithms analysis, we will use the term to refer our model of the expected input to our algorithms. We will use the following terminology to describe the inputs to our algorithms, and the results that the algorithms produce:

- A *site* is a point in the plane. Generally, we will use the terms point and site interchangeably.
- As in the above definitions, S is a set of n sites, and Q is a set of m sites. For the all-nearest-neighbors problem, we will also assume Q and S are the same set, though this is not the case in general.
- For a set of sites S, Vor(S) will denote the Voronoi diagram of S, DT(S) will denote the Delaunay triangulation of S. For each site $s \in S$, $V(s)$ will denote the Voronoi region of s and DN(s) will denote the set of sites in S that are connected to s in DT(S).

Many of our algorithms are designed to be efficient when the sites are independently chosen from a probability distribution. For example, most of the experiments in the thesis were done with the sites being generated from a uniform distribution in the unit square: $x = U(0,1)$, and $y = U(0,1)$. We will use the generic term "uniform inputs" to describe this class of input sets.

Such inputs include the uniform distribution in the unit square and circle, and distributions whose values are bounded above and below by constants over their entire domain.

When we say that an algorithms runs in expected time $T(n)$, we mean one of two things. If the algorithm is deterministic (e.g. it makes no random choices), $T(n)$ is the average runtime over some distribution of inputs, with each input equally likely.

On the other hand, we may allow the algorithm to make random choices. In this case $T(n)$ is the average runtime over some distribution of sequences of random choices, with each sequence equally likely.

The sequential quicksort algorithm provides a good example of each of these terminologies. In its original form, the algorithm has an expected runtime of $O(n \log n)$ as long as we assume every input permutation is equally likely. If we modify the algorithm to choose partition elements at random, then the algorithm will run in $O(n \log n)$ expected time for any input distribution, as long as every sequence of splitter choices is equally likely.

We will say that a randomized algorithm runs in time $T(n)$ with *high probability* if the probability that the algorithm fails to run within the time bound is less than $1/n^k$ for some $k > 1$.

This thesis will concentrate primarily on algorithms that are provably efficient on uniform inputs. This is because these inputs are usually an accurate model of data sets occurring in actual applications. In addition, we will also conduct tests to study the robustness of the algorithms on non-uniform inputs. These distributions are outlined in Chapter 2.

1.1.2 Models

One of the main difficulties facing the designer of parallel algorithms is choosing an appropriate model of computation. On the one hand, the model must be abstract enough to allow algorithm designers to analyze and reason about algorithms in a straightforward, machine independent way. On the other hand, it must be realistic enough to accurately model the performance of the target machines. Because current parallel architectures still vary so much from machine to machine, this task is even more difficult.

The RAM model is the standard model of complexity for sequential algorithms. It assumes a machine with a central processor and an infinite random-access memory. In a single machine cycle, the RAM machine can fetch a word, perform an operation in the processor and write the result back to memory. The operations that are allowed vary from machine to machine, but we will assume that real arithmetic, bit-wise operations and the floor function are all constant time. For practical problem sizes ($n < 2^{32}$), most machines provide this functionality.

People who design sequential algorithms have long used the RAM model as a basis for their analysis. While the model ignores many relevant machine details (size of memory, paging, number of registers, etc.), asymptotic analysis of RAM algorithms has largely been representative of real performance. Asymptotics has been abused from time to time, but for the most part careful analysis has yielded good answers.

The natural extension of the RAM model to parallel computing is just to replicate many RAM processors in a single machine where all processors share a common memory. The result of this replication is the PRAM model. The PRAM machine is made up of n processors that share a central memory. In each cycle, each processor is allowed to perform the same functions as a normal RAM processor. Different PRAM machines allow a different amount of concurrent memory access. EREW machine allow no concurrent access, CREW machines allow concurrent reads and CRCW machines allow concurrent writes as well. In addition, various versions of the CRCW model specify

different ways to resolve multiple writes to the same memory location. For example, the *arbitrary* CRCW model picks an arbitrary processor as the "winner" in such cases, while the *priority* CRCW model picks the winner according to a pre-assigned set of priorities.

The efficiency of PRAM algorithms is measured using two metrics. The first is $t(n)$, the number of parallel steps needed to solve the problem at hand. The second is $p(n)$, the number of processors needed to achieve the parallel runtime. The total *work* done by the algorithm is $t(n) \cdot p(n)$. A PRAM algorithm is *work efficient* if $t(n) \cdot p(n) = T(n)$ where $T(n)$ is the runtime of the best possible or best known sequential algorithm. A PRAM algorithm is *optimal* if $t(n) = \log^k n$ for some positive integer k (*polylog*(n) time) and it is work efficient.

NC or "Nick's Class" is the class of problems that have solutions running in *polylog*(n) time on a polynomial number of processors. In theoretical computer science, these problems are considered to have efficient parallel solutions. Thus, algorithms that run in *polylog*(n) time and only do *polylog*(n) extra work are thought of as efficient.

The PRAM model has many of the same weaknesses as the RAM model, since it ignores parameters such as machine size, and memory bandwidth, that are crucial to the performance of an algorithm. But, algorithms designed in the PRAM model have not been as successful in practice as those designed for a RAM machine. In fact, most programmers of parallel machines simply write off PRAM algorithms as nothing more than theoretical curiosities.

The problem with the PRAM model really isn't in the model itself. In fact, the PRAM model is remarkably similar to the data parallel programming models [SH86] that have recently come into wide use. Data parallel models are based on primitive operations that operate on large, aggregate data structures such as vectors and arrays. Examples of these primitives include elementwise arithmetic, permutation routing, and parallel prefix sums (also called *scans*). In his thesis, Blelloch [Ble90] showed that most PRAM algorithms can be translated into such a model. More recent work has shown that Blelloch's set of vector primitives can be efficiently implemented on a wide variety of machines [CBZ90, CBF91]. Thus, it is apparent that the PRAM model can be used as a basis for designing parallel algorithms that are efficient in practice.

The real problem with PRAM algorithms has to do with why they are developed and how they are analyzed. For the most part, the goal behind a PRAM algorithm is not to provide a practical method for solving a problem. PRAM algorithms are more likely to be vehicles for studying questions about complexity theory. As a result, asymptotic analysis is often abused, and an unrealistic importance is placed on obtaining polylogarithmic runtimes. In practice, asymptotic analysis hides important details, and working hard to achieve $polylog(n)$ time encourages the use of overly esoteric and expensive algorithms and data structures where simpler methods would be sufficient for practical problems. Kruskal, Rudolph and Snir [KRS90] have also criticized traditional notions of efficiency in parallel algorithms, and propose a more realistic approach to parallel complexity theory.

In this thesis, we will use both PRAM and data parallel models to express our algorithms. But, since we are concerned with practical computational methods, we augment the models with explicit costs for each primitive operation. Algorithms are then analyzed in terms of these costs, and can be easily reanalyzed for different machines. Such explicit cost models allow algorithms to be designed at a high level while retaining some contact with realistic machine constraints. In addition, we will be most interested in the performance of algorithms where the size of the problem, n, is large, but the size of the machine, p, is fixed. Thus, the question of interest is how to take advantage of the machine's parallelism, not how to exploit all of the inherent parallelism in a particular algorithm.

1.1.3 Machines

No practical study of algorithms would be complete without machines to run programs on. In this thesis, we will use three principal machines in our experiments: a Sparcstation 2, a Cray-YMP, and a KSR-1.

The baseline machine will be SparcStation 2 workstation with a 40MHz clock rate and 64MB of memory. Most of the simulation work and preliminary program development for this thesis was done on this machine. In addition, it provides the main cost/performance comparison point for the parallel algorithm implementations that we will study.

The Cray Y-MP represents the "traditional" large scale parallel architecture. It uses a combination of large, heavily pipelined CPUs and a large, high bandwidth memory to provide effective support for program written in a vector-based data parallel style.

The KSR-1 represents a newer style of parallel architecture. It supports many more, much less powerful CPUs and a relatively low bandwidth distributed memory system. The memory system is enhanced through the use of local caches. The caches are kept globally coherent through the use of extra hardware and a ring-based communications network. Thus, the KSR-1 provides a shared memory programming model on top of a message passing architecture.

Chapter 3 will cover the machines, and machine models in more detail.

1.1.4 Other Notation and Terminology

Below is a list of notation and terminology that we will use throughout the rest of this dissertation:

- We will use "log" to mean the natural logarithm, and "lg" to mean the base 2 logarithm. We will never use "ln." We will use H_n to denote the nth harmonic number: $H_n = \sum_{i=1}^{n} 1/i$.
- A thread is a program counter and a runtime environment within some address space. Traditional operating systems provide a process abstraction that is usually made up of an address space, various operating system data structures and one thread. Operating systems for parallel machines often allow processes to hold many threads of control within one address space. This allows applications to exploit parallelism at a fine grain, since threads are cheap to create while processes are more expensive. This notion is also similar to the idea of virtual processors or VPs, that some SIMD machines use to multiplex physical hardware. We will use both terms synonymously.

1.2 Related Work

In this section, we will survey the literature on sequential and parallel algorithms for closest point problems. The section is divided into two parts, one devoted to theoretical results, and one to practical implementations.

1.2.1 Theory

Researchers in computational geometry have been very successful in designing algorithms for closest point problems that are elegant, theoretically efficient and practical. Bentley, Weide and Yao [BWY80] describe "cell" algorithms for nearest-neighbor search and Voronoi diagram construction that use buckets as their main data structure and run in expected time $O(n)$ when the input points are chosen from the uniform distribution in the unit square. These algorithms use a technique called "spiral search" which they show to be both simple and efficient.

Clarkson's thesis [Cla84] presents results on closest point problems in higher dimensions. He developed algorithms for various kinds of nearest-neighbor searching and for constructing minimum spanning trees. While his algorithms have not been tested in a practical situation, they are relatively simple, and use generalizations of the quad-tree to subdivide space and make searching efficient. Bentley's k-d tree data structure can also be use to solve the all-nearest-neighbors and other closest point problems in higher dimensions [Ben80]. Bentley uses this data structure in the plane to answer fixed-radius nearest-neighbor queries in his traveling salesman programs [Ben89].

Constructing planar Voronoi diagrams has also received a large amount of attention. Guibas and Stolfi [GS85] present a unified framework and ideal data structure for computing and representing the diagram and its dual. Their algorithms then construct Delaunay triangulations rather than Voronoi diagrams. We also will use this approach because the resulting algorithms are simpler and more elegant.

Guibas and Stolfi show how to implement two algorithms for constructing the Delaunay triangulation. The first is a divide and conquer algorithm, and the second constructs the diagram incrementally. Each of these algorithms had previously been presented in the literature [GS77, LS80, SH75], but Guibas and Stolfi's implementations are simpler and more elegant than the earlier algorithms. These algorithms run in $O(n \log n)$ and $O(n^2)$ worst-case time respectively. Fortune [For87] examines how to construct an optimal sweepline algorithm for constructing the Voronoi diagram. The algorithm uses a transformation of the plane to allow a straightforward sweepline algorithm to correctly compute either the Voronoi diagram or the Delaunay triangulation of the input.

Many proposed algorithms have good expected-case runtimes. These algorithms come in two flavors. First, there are algorithms whose expected runtime depends on the input distribution. These algorithms tend to use some sort of bucketing scheme to take advantage of smooth input distributions. Bentley, Weide and Yao's cell algorithms fall into this class [BWY80]. Maus [Mau84] presents a simple algorithm for constructing the Delaunay triangulation that is similar to gift-wrapping algorithms for convex hulls [PS85]. The algorithm repeated discovers new Delaunay triangles using a search process that is similar to spiral search. The algorithm uses bucketing to speed up its inner loop and runs in expected time $O(n)$. Dwyer's thesis [Dwy88] shows that a generalization of this algorithm to higher dimensions also runs in linear expected time on uniform inputs. Dwyer [Dwy87] also shows that a modified version of the divide and conquer runs in $O(n \log \log n)$ expected time. Finally, Ohya, Murota and Iri [OIM84] describe an incremental algorithm that uses a combination of bucketing and quad-trees to achieve linear expected time.

A second class of fast expected time algorithms uses randomization to achieve good expected performance on any input. Clarkson and Shor [CS89] describe an incremental algorithm that inserts the points in a random order and runs in $O(n \log n)$ expected time. Knuth, Guibas and Sharir [GKS92] give a refined analysis of a modified randomized incremental algorithm, and Sharir and Yaniv [SY91] refine the analysis further and discuss an implementation of this algorithm. These algorithm use a persistent tree structure to perform point location in the current diagram. Devillers, Meiser and Teillaud [DMT90] use similar data structures in their dynamic algorithms. Finally, Clarkson, Mehlhorn and Seidel [CMS92] show how to extend Clarkson's original results to develop dynamic algorithms for higher-dimensional convex hulls, that can be used to construct the Voronoi diagram.

Many of these studies have produced algorithms that are both theoretically efficient and practical. Chapter 2 will describe the incremental, divide and conquer and sweepline algorithms in

more detail and present detailed a performance analysis of their implementations. This discussion is important because good parallel algorithms are often based on ideas derived from good serial algorithms. Therefore, a thorough understanding of known conventional algorithms is needed before beginning to design parallel methods.

Parallel computational geometry has lagged behind its sequential counterpart in the design of practical algorithms. However, a large amount of theoretical work has been done over the last ten years or so. Most of this work has been in the form of algorithms for one of the PRAM models [KR90]. Recall that PRAM machines come in several flavors depending on how they control concurrent access to memory locations. In parallel computational geometry, the CREW PRAM, which allows concurrent reads but not concurrent writes is the most popular. However, many algorithms also use the CRCW PRAM, which allows both concurrent reads and writes.

Aggarwal, et al. [ACG+88] summarize much of this work. In particular, they describe a parallel algorithm for constructing planar Voronoi diagrams that runs in $O(\log^2 n)$ parallel time on $O(n)$ CREW processors. The algorithm is a parallelization of the standard divide and conquer algorithm, and the main problem is executing the merge step in parallel. Goodrich, Cole, and O'Dunlaing [CGO90] present an even more complicated scheme that achieves $O(\log^2 n)$ time on $O(n/\log n)$ processors. In order to achieve these time bounds, each of these algorithms use elegant, but very complex data structures. This makes it unlikely that the algorithms could be efficiently implemented on current machines.

Several algorithms in the literature achieve optimal expected run times. Venmuri, Varadarajan and Mayya [VVM92] present an $O(\log n)$ expected time, $O(n)$ processor divide and conquer algorithm. It uses a combination of bucketing and parallel dividing chain construction to achieve its runtime. The CRCW algorithm of Levcopoulos [LKL88] uses a hierarchical bucketing scheme, and runs in $O(\log n)$ expected time on $O(n/\log n)$ processors. This algorithm is a parallelization of Bentley Weide and Yao's scheme. Mackenzie and Stout [MS90] present another version of this algorithm that runs in $O(\log\log n)$ expected time on $O(n/\log\log n)$ processors. These methods are mainly of theoretical interest because the constants inside the big-O's are large.

Many recent algorithms achieve optimal performance with high probability by using some kind of randomized divide and conquer. Reif and Sen [RS92] show how to use randomized divide and conquer to construct the convex hull of a set of points in three dimensions. This results in an algorithm for constructing the Voronoi diagram that runs in $O(\log n)$ time on $O(n)$ CRCW processors with high probability. While the published algorithm is extremely complex, a simplified version for constructing the Voronoi diagram seems practical. Chapter 6 will discuss this algorithm in more detail. Goodrich and Ghouse [GG91] also use random sampling to construct three-dimensional convex hulls. Their algorithm is a parallel version of the convex hull algorithm of Edelsbrunner and Shi [ES91]. Goodrich and Ghouse use some additional probabilistic machinery to improve the confidence bounds on their run time. They also used the Reif/Sen algorithm as a subroutine, and so it is also unlikely that their method is practical as published.

Finally, many algorithms have also been proposed for two-dimensional mesh-connected computers. Dehne [Deh89] presents a simple iterative algorithm to construct the Voronoi diagram on a raster screen. Lu [Lu86] describes a divide and conquer algorithm that runs in $O(\sqrt{n}\log n)$ time on an n processor mesh. Jeong and Lee [JL90] improve this to $O(\sqrt{n})$ time on an $\sqrt{n} \times \sqrt{n}$ processor mesh.

1.2.2 Practice

Researchers in scientific computation have done the most practical work in implementing parallel algorithms. Most of this work discusses parallel algorithms for various numerical problems such as solving linear systems. Hundreds of papers document the performance of these codes, and the dominant style of presentation is to compare different methods in benchmarks using various metrics such as speedup, or MFLOPS. Fox's book [FJL$^+$88] describes work done by the scientific computing group at Caltech, and gives the flavor of this kind of research.

There has been comparatively little work on implementations of combinatorial or geometric algorithms. Sorting has received most of the attention in this area. Fox [FJL$^+$88] describes hypercube implementations of quicksort, shellsort, and bitonic sort. He also compares them to a hypercube version of sample sort. Blelloch, et al. [BLM$^+$91] describe sorting algorithms for the Connection Machine, and surveys most of the known theoretical and practical parallel sorting algorithms. The paper also analyzes CM-2 implementations of bitonic sort, radix sort and sample sort. One feature of the paper is a carefully constructed analytical model that accurately predicts the performance of the sorting algorithms over a large range of input sizes. This style of analysis is also used by Blelloch and Zagha [ZB91] in their paper on vectorized radix sort, and by Hightower, Prins, and Reif [PHR92] in their paper on sample sort for mesh-connected machines.

In parallel computational geometry, Blelloch's thesis [Ble90] describes algorithms for planar-convex hulls, building k-d trees, finding the closest pair, and other simple geometric problems. However, he does not give a detailed performance analysis of these algorithms. Cohen, Miller, Sarraf, and Stout [CMSS92] report on hypercube algorithms for various problems. In addition, some practical work has been done on algorithms for constructing three dimensional Delaunay triangulations for use as finite element meshes. Merriam [Mer92] and Teng, et al. [TSBP93] describe algorithms for the Intel machines and the Connection Machine respectively. Most of the other implementation work comes from application areas such as image processing and computer graphics, and it is beyond the scope of this thesis to survey that literature here.

In this dissertation, we use implementations in two ways. First, they will demonstrate that the proposed algorithms actually achieve good performance on a real machine. These results will be presented in a standard benchmark style. In addition, implementations, or simulations of them, can be used to gain a more detailed understanding of the algorithms themselves. McGeoch's thesis on experimental algorithms analysis [McG86] and recent work by Bentley on the traveling salesman problem [Ben89] presents a framework for experimental analysis of algorithms. In this thesis, we will apply these methods to the design and analysis of parallel algorithms. Chapter 2 presents these ideas in more detail.

1.3 Contributions

This dissertation will describe several case studies in the design, analysis and implementation of parallel algorithms for computational geometry. The case studies will concentrate on closest point problems, but I will try to outline general principals that are applicable to any problem. This work makes the following contributions:

Methodology. I will describe and apply a systematic methodology for the implementation and experimental analysis of algorithms, both sequential and parallel. The method utilizes high level models for designing and analyzing parallel algorithms that are similar PRAM [KR90] and to the Blelloch's vector models [Ble90]. These models are based on primitive operations, such as routing, element-wise arithmetic, and parallel prefix sums, that have efficient

implementations on current parallel architectures. In order to keep my analysis realistic, I augment the models in two ways. First, the models are parameterized by the cost of the primitives on the target machines. Second, the models themselves may be modified to include higher level operations, so long as these primitives are chosen carefully, and their cost in an implementation are not excessive. Finally, the methodology depends on a pragmatic mix of mathematical and experimental analysis. Mathematical analysis provides a high level view of performance, while experimental analysis effectively guides the design process towards the bottlenecks in an implementation.

Algorithms. The case studies will describe the design and analysis of several algorithms for the all-nearest-neighbors problem and for constructing Voronoi diagrams in two dimensions. The case studies will provide a practical guide to efficient sequential and parallel algorithms for Voronoi diagram construction. In addition, they will illustrate how parameterized models and careful experimentation can accurately predict the performance of parallel programs in a machine-independent manner. Finally, the parallel algorithms are novel results by themselves. Not only are they efficient in realistic models of computation, but they also exhibit good performance when carefully implemented on real parallel machines. This combination of theoretical analysis, pragmatic machine models and careful programming is the main focus of the dissertation.

Benchmarks. Since the algorithms have all been implemented, they can be used as additional benchmarks to study the performance characteristics of parallel architectures. Algorithms for closest point problems have a more dynamic and irregular flavor than the benchmark programs normally used to analyze machine performance. In addition, the algorithms are more sophisticated than some irregular benchmarks that are already in use [Cha91].

The dissertation does not address any of the following concerns:

Full applications. The algorithms that I study fall into the class of "kernel" programs. Thus, my results will only reflect the performance of a small percentage of any large application. However, experience from numerical computing suggests that having high performance libraries of kernel algorithms available is very valuable [LHKK79] .

Parallel I/O. The algorithms in this dissertation do not attempt to take advantage of disk arrays or other parallel I/O devices. I assume that the algorithms operate on data sets that have already been loaded into main memory and results are stored in main memory. Optimizing the algorithms to deal with I/O issues is beyond the scope of this work. Aggarwal and Vitter [ACG+88] and Vitter, Shriver and Nodine [VS92, NV91] have obtained some basic theoretical results in this area. Womble, et. al. [WGWR93] and Cormen's thesis [Cor92] examine systems issues, including the implementation of I/O efficient algorithms and virtual memory for data parallel computation.

Compilers. In this work, I use compilers to generate efficient machine code from a relatively high level description of my algorithms. The use of restructuring compilers and parallelizing compilers is the subject of much current research. Wolfe [Wol89] gives a good overview of basic methods and provides more detailed references. This dissertation does not address the effect of more sophisticated compilers on algorithm development.

Degeneracy and roundoff error. The design of robust geometric algorithms has been the subject of much theoretical research [DS90, For89, For92, Mil88, Yap90]. While handling these details is important, a general discussion of these issues is beyond the scope of this thesis. My algorithms assume non-degenerate input and are for the most part naive about numerical issues.

Generalized abstract models. Finally, this work does not address the extension of standard theoretical models to deal with asynchrony, memory latency, or hierarchical memory systems [ACF90, ACS89, ACS90, CZ89, CKP+92, Gib89, HR91]. In this thesis, we will be concerned with these issues only when they become real bottlenecks in the performance of our algorithms. Incorporating low level machine parameters such as network bandwidth, cache size, and the cost of synchronization into the analysis of every parallel algorithm is no more appropriate than incorporating parameters such as processor speed, the number of available registers, cache size, or bus bandwidth into the analysis of sequential algorithms. All of these parameters are potentially important, but not in every situation.

1.4 Outline of the Thesis

First, Chapter 2 compares the performance of a large number of known sequential algorithms for constructing the Delaunay triangulation. In addition, it discusses a new incremental algorithm for constructing planar Delaunay triangulations. This algorithm uses both randomization and bucketing to achieve speed while retaining simplicity. After a discussion of parallel programming models in Chapter 3, Chapter 4 investigates the application of bucketing techniques to parallel algorithms. The result of this investigation is a high performance parallel algorithm for the all-nearest-neighbors problem. In chapters 5 and 6, we discuss how these ideas fit together into fast parallel algorithms for constructing the Delaunay triangulation. Finally, Chapter 7 concludes the thesis and outlines areas of future work.

Chapter 2
Practical Sequential Voronoi Algorithms

Every man is the center of a
circle, whose fatal
circumference he cannot pass.
—John James Ingalls

Sequential algorithms for constructing the Voronoi diagram come in three basic flavors: divide-and-conquer [Dwy87, GS85], sweepline [For87], and incremental [CS89, GS77, Mau84, OIM84].

This chapter presents an experimental comparison of these algorithms. In addition, it describes a new incremental algorithm that is simple to understand and implement, but still competitive with the other, more sophisticated methods on a wide range of problems. The algorithm uses a combination of dynamic bucketing and randomization to achieve both simplicity and good performance.

The experiments in this chapter also illustrate basic principles in the design and experimental analysis of algorithms. They provide a detailed characterization of the behavior of each algorithm over a wide class of problem instances. We achieve this through a combination of high level primitives (*abstraction*), explicit cost models, algorithm animation and careful design and experimentation. In later chapters, we will apply these methods to the study of parallel algorithms.

2.1 Divide-and-Conquer

Divide-and-conquer algorithms work by dividing the original input set into smaller subsets and solving the sub-problems recursively. Then, answers for the sub-problems must be merged together to form the final answer for the entire problem. Guibas and Stolfi [GS85] present a conceptually simple method for implementing this idea, and an elegant data structure for representing both the Delaunay triangulation and the Voronoi diagram at the same time. The "quad-edge" data structure represents a subdivision of the plane in terms of the edges of the subdivision. Each edge is a collection of pointers to neighboring edges in the subdivision or its dual.

In Figure 2.1, the solid edges are from the subdivision, while the dotted edges are from the dual. Each edge `e` has an origin, `e.Org`, a destination, `e.Dest`, and a left and right face. The edge `e.Sym` is the same as `e` but directed the opposite way. The edge `e.Rot` is the edge dual to `e` and directed from right to left. The edges `e.Onext` and `e.Lnext` are the next edges counterclockwise around the same origin and left face respectively. Similarly, `e.Oprev` and `e.Lprev` are the edges clockwise around `e.org` and the right face of `e`. Finally, `e.Dnext`, `e.Dprev`, `e.Rnext` and `e.Rprev` are defined analogously for the destination of `e` and the right face of `e`. Guibas and Stolfi show that each of these edges is computable from a constant number references to `e.Sym` and `e.Rot`. The proof of this, and the other algebraic properties of these abstractions is beyond the scope of this chapter.

Edges are represented using a record of four pointers and four data fields. This data represents the edges `e`, `e.Rot`, `e.Sym` and `e.Rot.Sym`, and the pointers represent the `Onext` edge of each of

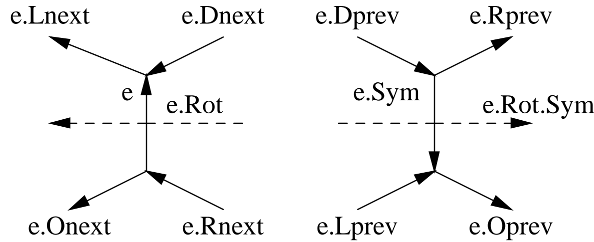


Figure 2.1: The quad-edge data structure.

these edges. In general, if e is an edge record, then $e[i]$ represents the edge $e.Rot^i$. As an example, Figure 2.2 shows a triangle and its quad-edge representation.

Faces and vertices in the subdivision are represented implicitly by circular linked lists of edges. In the picture, bold lines connect nodes in a vertex ring while thin lines connect nodes in a face ring. To add and delete edges from the subdivision, we use an operator called `splice(a,b)` , which is similar to linked list insertion. A call to `splice(a,b)` has the effect of splicing the ring of the edge `b` together with the ring of the edge `a`. If `a` and `b` are part of the same ring, then `splice` splits that ring into two pieces. Using splice, the `connect(e,f)` routine connects two edges `e` and `f` together so that they share a common left face. Guibas and Stolfi give a more detailed description of how this works, and also discuss the topological and algebraic foundations of this representation in great detail. To understand the algorithms in this chapter, it should be sufficient just to keep the above picture in mind when reading the pseudo-code.

Guibas and Stolfi's algorithm uses two additional geometric primitives. First, it uses a two-dimensional orientation test, `CCW(a,b,c)` that returns whether the points `a`, `b` and `c` form a counter-clockwise turn. The second is `in-circle(a,b,c,d)`, which determines whether `d` lies within the circle formed by `a`, `b` and `c`, assuming that `CCW(a,b,c)` is true. These primitives are defined in terms of two and three dimensional determinants. Fortune [For89, For92] shows how to compute these accurately with finite precision.

With these primitives in place, Guibas and Stolfi's algorithm proceeds using a standard divide-and-conquer scheme. The points are sorted by x-coordinate, and split until three or fewer points remain in each sub-problem. Then, the sub-problems are merged together using the routine shown in Figure 2.3

The `Merge` procedure must accomplish two tasks. First, it must delete any edges in L and R that are not valid. Second, it must create a set of the valid Delaunay edges, called *cross-edges*, connecting L to R. The process begins at the base of the convex hull of $L \cup R$. The `Merge` procedure then walks upward along a vertical line between L and R, finding cross-edges as it goes. In the code, the variable `bl` keeps track of the current cross-edge. The next cross-edge will either connect the left end-point of `bl` to some point in R that is above `bl`, or it will connect the right end-point of `bl` to some point in L. Conceptually, the algorithm finds this new cross-edge by expanding the empty circle incident on `bl` into the half-plane above `bl`. The portion of this "bubble" that lies below `bl` stays empty, since it must shrink as the circle expands into the other half-plane. The portion of the bubble above `bl` will eventually encounter a new site, either in L or in R. The endpoints of `bl` and the new site will then define a new Delaunay triangle, since the circle growing from `bl` is empty, and this triangle will contain the next cross-edge. The algorithm then repeats the process inductively from the new cross-edge. This whole process has the effect of iteratively weaving the

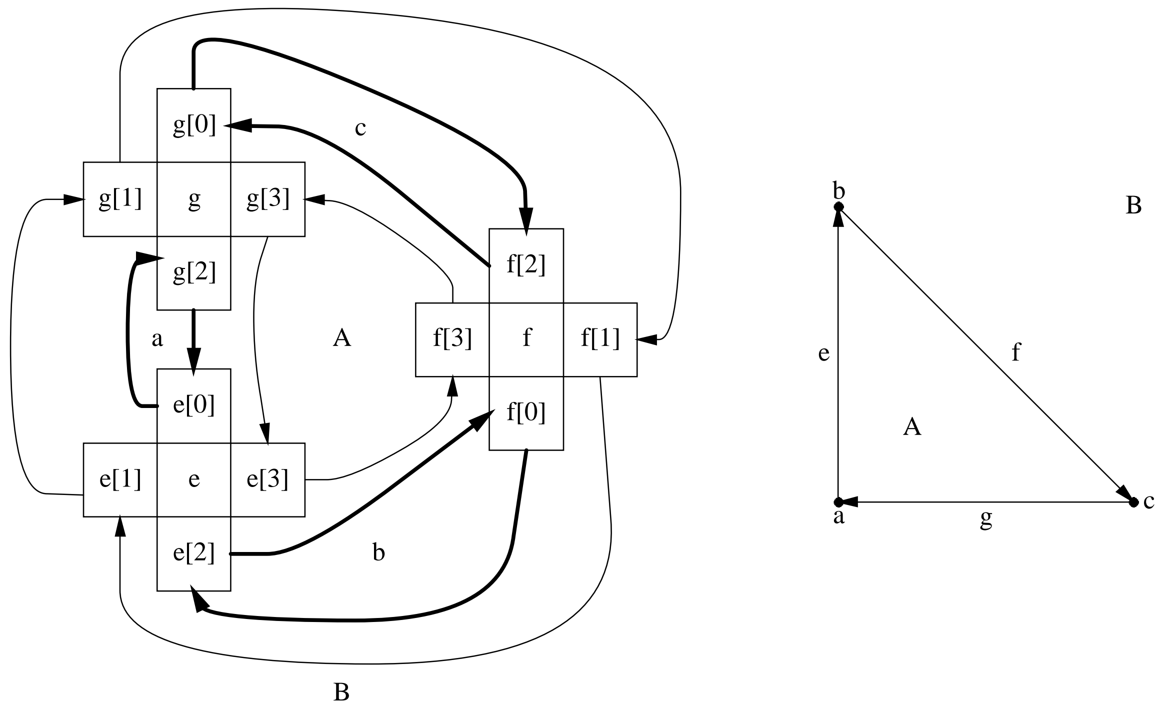


Figure 2.2: The quad-edge representation of a simple subdivision. Each block of four on the left represents one edge on the right. For example, the pointer `e[0]` to `g[2]` indicates that the edge counterclockwise from `e` around `a` is the edge from `a` to `c` which is the symmetric copy of `g`. Thus, the vertex `a` is represented implicitly by the ring from `e[0]` to `g[2]`

```
Merge(L, R)
{
  Find the lower common tangent between R and L;

  /* bl is the ‘base edge’ */
  bl := lower tangent from R to L
  while not (found upper tangent) {
    lcand := bl.Sym.Onext; lvalid := CCW(lcand.Dest, bl.dest, bl.org);
    if (lvalid) {   /* remove bad edges from L */
      while
        (in-circle(lcand.dest,lcand.Onext.dest, lcand.org, bl.org)
          t := lcand.Onext; Delete lcand ; lcand := t;
        }
    }
    rcand := bl.OPrev; rvalid := CCW(rcand.Dest, bl.dest, bl.org);
    if (rvalid) {   /* Remove bad edges from R */
      while
        (in-circle(rcand.Dest,rcand.Oprev.Dest, rcand.org, bl.dest)) {
          t := rcand.Oprev; Delete rcand ; rcand := t;
        }
    }
    if ((!lvalid) && (!rvalid)) { /* found upper tangent! */
      return;
    }
    /* Now connect the new cross-edge. */
    if (!lvalid || (rvalid
         && in-circle(lcand.Dest, lcand.org, rcand.Org, rcand.Dest)))
      bl := connect(rcand, bl.Sym);
    } else {
      bl := connect(bl.Sym, lcand.Sym); bl := bl.Sym;
    }
  }
}
```

Figure 2.3: The merge procedure in Guibas and Stolfi’s divide and conquer algorithm.

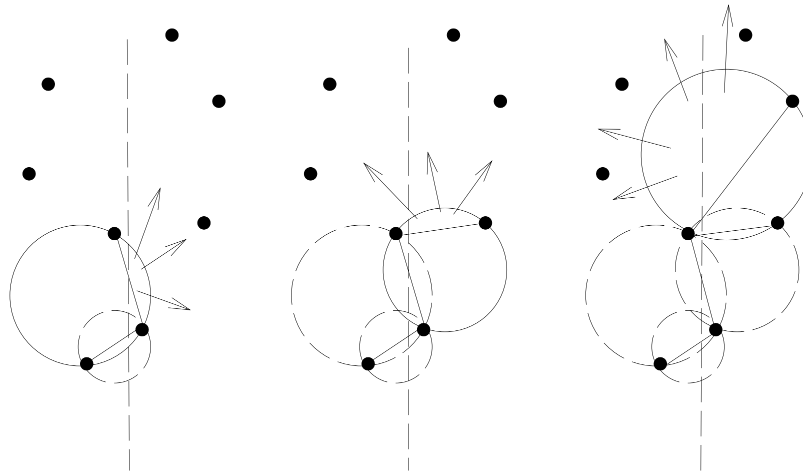


Figure 2.4: The rising bubble.

circular bubble up between L and R, finding cross-edges each time the bubble encounters a site (see Figure 2.4).

In the code, the two while loops perform the task of finding the next cross-edge. The first while loop examines edges incident on the left end-point of `bl` in a counterclockwise fashion. The loop uses `in-circle` to find and delete edges that are invalidated by `bl.org`. The second while loop performs the symmetric operation on the right endpoint of `bl`. The two variables `lvalid` and `rvalid` keep track of whether the rising bubble will in fact hit some edge in L or R respectively. After the two loops are finished, there are two cases, either both `lcand` and `rcand` hold a valid candidate edge to which the algorithm can connect the new cross-edge, or there is only one edge available. In the first case, the final circle test determines whether the new cross-edge should be connected to the right or left endpoint of `bl`, otherwise, no such test is needed. The process then continues until it reaches the upper convex hull edge of $L \cup R$.

Figure 2.5 shows the operation of the `Merge` loop, starting from the lower common tangent of the left and right diagrams. In the frames, bold circles show successful `in-circle` tests, dotted edges are being tested for validity. The current the value of `bl`, which is the most recent cross-edge to be computed will be called the "base edge", and is shown in bold. Cross-hairs mark candidate points for connecting the new cross-edge.

Reading the frames left to right and top to bottom, the first frame shows that the merge process has proceeded to the bold edge. The next five frames show the action of the two while loops. No edges in the left diagram are deleted, while one in the right diagram is. The eighth frame shows the circle test that determines which way to connect the new cross-edge. Since the circle incident on the left edge is empty, the cross edge is connected from the right endpoint of the base to the left subproblem. This new edge is shown in bold in the last frame, and will become the new base edge in the next iteration of `Merge`.

Guibas and Stolfi provide a proof that this scheme only deletes invalid edges and only adds valid ones, so the final diagram is the Delaunay triangulation of the whole set. They also show that the merge step takes $O(n)$ steps in the worst case, so the whole algorithm runs in $O(n \log n)$ worst-case time. In comparison-based models of computation, this is the optimal runtime for any algorithm that constructs the Voronoi diagram because the problem can be reduced to sorting.

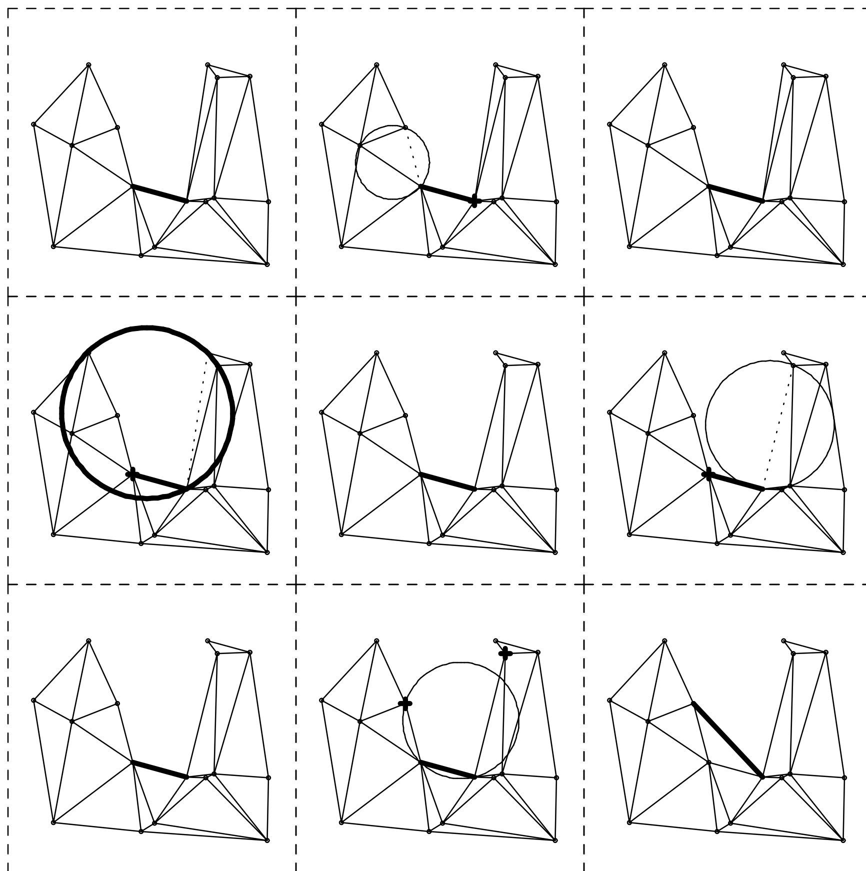


Figure 2.5: The merge loop.

```
Do_Event(e) {
  if e is a site {
    Find the site T such that the circle inscribed through
      e and T and tangent to the sweepline is empty.
    Add the edge (e,T) to the frontier;
    Update the event queue with any new circle events;
    Delete invalid circle events from the event queue;
  } else {
    Let (a,b,c) be the sites lying on the circle;
    Remove the edges (a,b) and (b,c) from the frontier;
    Add the edge (a,c) to the frontier;
    Update the event queue as above;
  }
}
```

Figure 2.6: Code for processing events in Fortune's algorithm

However, if our computational model includes a unit-time floor function, then better algorithms are possible for some special cases.

Dwyer [Dwy87] showed that a simple modification to this algorithm runs in $O(n \log\log n)$ expected time on uniformly distributed points. Dwyer's algorithm splits the point set into vertical strips of width $\sqrt{n/\log n}$, constructs the DT of each strip by merging along horizontal lines and then merges the strips together along vertical lines. His experiments indicate that in practice this algorithm runs in linear expected time. Another version of this algorithm, due to Katajainen and Koppinen [KK87] merges square buckets together in a "quad-tree" order. They show that this algorithm runs in linear expected time for uniform points. In fact, their experiments show that the performance of this algorithm is nearly identical to Dwyer's.

2.2 Sweepline Algorithms

Fortune [For87] discovered another optimal scheme for constructing the Delaunay triangulation using a sweepline algorithm. The algorithm keeps track of two sets of state. The first is a list of edges called the *frontier* of the diagram. These edges are a subset of the Delaunay diagram, and form a tour around the outside of the incomplete triangulation. The algorithm also keeps track of a queue of events containing *site* events and *circle* events. Site events happen when the sweepline reaches a site, and circle events happen when it reaches the top of a circle formed by three adjacent vertices on the frontier. In the discussion below, we will say that a circle event associated with the triple (a, b, c) is *incident* on the frontier edges (a, b) and (b, c). Events are ordered in the y direction, so the next event at any time is event in the queue with minimum y coordinate. The algorithm sweeps a line up in the y direction, processing each event that it encounters (see Figure 2.6).

Conceptually, Fortune's algorithm constructs valid Delaunay edges, one at a time, in an upward sweep over the point set. At each site event, the algorithm searches for a valid Delaunay edge between the new site, s, and the points on the frontier. It does this by finding the site t on the frontier such that the circle inscribed through s and t and tangent to the sweepline is empty. This can be accomplished using a binary search over the edges in the frontier. The edges (s, t) and (t, s) are guaranteed to be Delaunay edges, so they are added to the frontier.

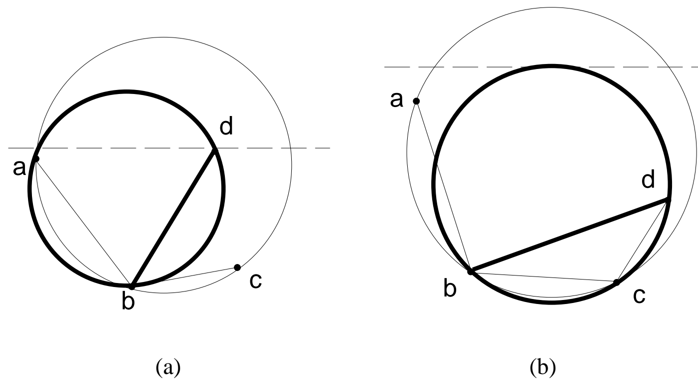


Figure 2.7: In (a) a new site (d) creates a new frontier edge (the bold edge) and a new circle event (the bold circle), invalidating an existing circle event (the light circle). In this case a new circle incident on (b, d) might also be created, but is not shown to make the picture more clear. In (b), a circle event (bold circle) creates a new edge (bold line) and invalidates an existing circle event (light circle). In this situation, a new circle event incident on (a, b) and (b, d) would also be created.

When a new edge is added to the frontier, it may generate new circle events. Circle events are added to the event queue when a new frontier edge makes an externally convex angle with either of its neighboring edges, that is, whenever the edge could be one edge in a Delaunay triangle that will be completed later on. The priority of the new event is defined to be the y-coordinate of the top of the circle.

In addition, adding a new edge to the frontier may invalidate an existing circle event. Suppose that three sites, a, b, and c form two edges (a, b) and (b, c) that are currently on the frontier and have a circle event C associated with them. That is, the two edges form an externally convex angle and thus could be two edges in a Delaunay triangle. If a new site d is swept and is connected to b, the event C becomes invalid, and is replaced by a circle event associated with with the edges (a, b) and (d, b) (see Figure 2.7a).

When the sweepline reaches a circle event, it will have reached the y-coordinate of the top a circle through three sites, a, b and c, where (a, b) and (b, c) are adjacent edges on the frontier and form an externally convex angle. The circle is guaranteed to be empty, since any site in the circle would have been encountered by the sweepline at some earlier time and invalidated the event [For87]. To process the event, the algorithm removes (a, b) and (b, c) from the frontier (but *not* from the diagram!), and replaces them with the edge (a, c). If either (a, b) or (b, c) were associated with another circle event, it is invalidated and replaced with a circle incident on (a, c). If the edge (a, c) forms an externally convex angle with either of its neighbors, a new circle event is generated and added to the event queue (see Figure 2.7b).

In his paper, Fortune shows that this scheme only adds valid Delaunay edges to the diagram, since the edges added when processing site events are valid, and circle events that are reached by the sweepline represent empty circles. Using standard tree-based data structures for the priority queue and the frontier, each new event can be processed in $O(\log n)$ time, so the whole algorithm uses $O(n \log n)$ time in the worst case.

In Fortune's implementation, the frontier is a simple linked list of half-edges, and point location is performed using a bucketing scheme. The x-coordinate of a point to be located is used as a hash

value to get close to the correct frontier edge, and then the algorithm walks to the left or right until it reaches the correct edge. This edge is placed in the bucket that the point landed in, so future points that are nearby can be located quickly. This method works well as long as the query points and the edges are well distributed in the buckets. A bucketing scheme is also used to represent the priority queue. Members of the queue are bucketed according to their priorities, so finding the minimum involves searching for the first non-empty bucket and pulling out the minimum element. Again, this works well as long as the priorities are well distributed.

Figure 2.8 shows the operation of the algorithm on a small point set. In the pictures, the sweepline is shown as a dotted horizontal line. Sites are large dots, and circle events shown as "+" signs marking the top of the corresponding circle.

Reading the frames from left to right and top to bottom, the first few frames show how a circle event is processed. In the first frame, the sweepline has reached the top of the bold circle. The algorithm first removes this event from the priority queue. Then, it removes from the frontier the two edges incident on the empty circle (frames 3 and 4). Frame 5 shows the algorithm completing the new triangle. The bold circle shown in frame 6 becomes invalid at this point because it was incident on one of the edges just removed from the frontier (the rightmost one). Frame 7 shows its replacement. In frame 9, the sweepline reaches a new site. Frames 10 to 12 show the search for the correct frontier site, and frame 13 shows the new frontier edge. Finally, in frames 14 to 16, the algorithm adds a new circle event incident on the new frontier edge, and this event becomes the next one in line to be processed.

2.3 Incremental Algorithms

The third, and perhaps simplest class of algorithms for constructing the Delaunay triangulation is incremental algorithms. We will study two styles of incremental algorithms: incremental *construction* and incremental *search*. Incremental construction algorithms add sites to the diagram one by one and update the diagram after each site is added. Guibas and Stolfi [GS85] present a basic incremental construction algorithm at the end of their paper. The algorithm consists of two main subroutines: `Locate` locates a new site in the current diagram by finding the triangle that the point lies in, and `Insert` inserts a new site into the diagram by calling `Locate` and then iteratively updating the diagram until all edges invalidated by the new site have been removed (see Figure 2.9).

The `Locate` routine works by starting at a random edge in the current diagram and walking along a line in the direction of the new site until the right triangle is found. The algorithm is made simpler by assuming that the points are enclosed within large triangle. The `Insert` routine works by calling `Locate` and connecting the new point into the diagram with one edge to each vertex of the triangle found by `Locate`. Then, `Insert` updates the diagram by finding invalid triangles around the outside of the polygon containing the new site. These triangles are found using the `in-circle` test. Let ABC be such a triangle with the point A opposite the new site p. The edge flipping procedure replaces the edge BC with an edge from A to p, creating two new triangles ApC and ABp. The loop then adds the edges AC and AB to a queue of edges to be checked. It keeps track of this queue implicitly through links in the quad-edge data structure. When the loop comes back to the original starting edge, this queue is empty and the routine stops. Guibas and Stolfi show that when the loop terminates, all of the edges in the diagram are guaranteed to be valid.

Figure 2.10 shows the operation of the `Insert` routine. In the first frame, we are inserting the site p marked by the "+" symbol. The bold edges show the path that `Locate` takes through the diagram to find the new site. The second frames shows the three new edges that connect p into the

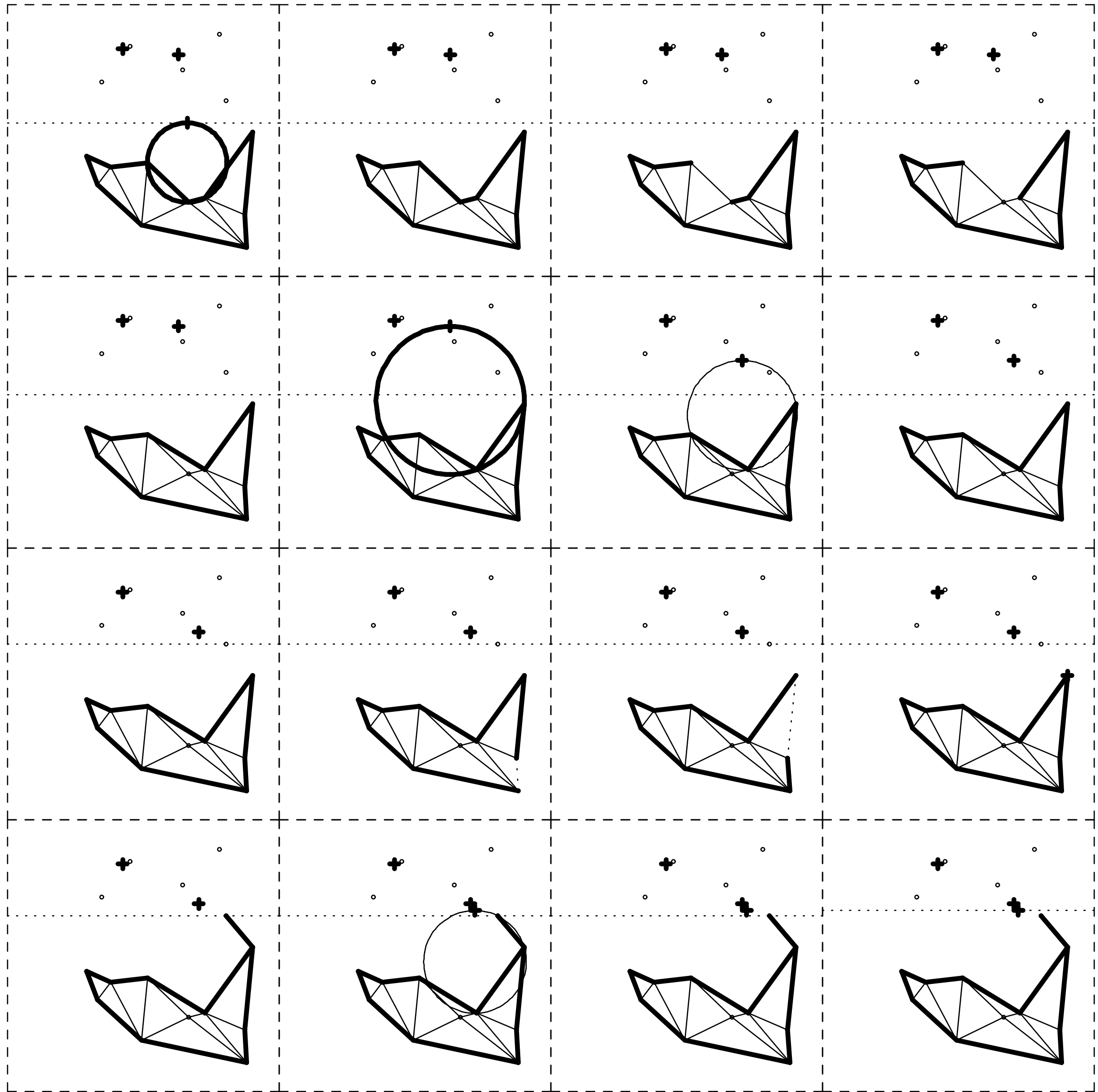


Figure 2.8: Fortune's algorithm.

```
Locate(start, p)
{
  e := start;
  while(true) {
    if (p == e.Dest || p == e.Org) return e;
    if (right_of(p,e.Org, e.Dest)) e := e.Sym;
    else {
      t1 := e.Onext;
      if (CCW(p,t1.Org,t1.Dest)) e := t1;
      else {
        t1 := e.Dprev;
        if (CCW(p,t1.Org, t1.Dest)) e := t1;
        else { return e; } } } }
}
Insert(p)
{
  e := Locate(last,p);
  if (p == e.Org || p == e.Dest) return;
  if (is_on(p,e.Org, e.Dest)) { t = e.Oprev; delete(e); e = t; }
  base = make_edge(e.Org, p); first = e.Org; splice(base,e);
  do {  /* Connect new point */
    t = base.Sym;
    base = connect(e, t);
    e = base.Oprev;
  } while (e.Dest != first);
  e = base.Oprev;
  while(true) {   /* Flip invalid edges until done */
    t = e.Oprev;
    if (not CCW(t.Dest, e.Org, e.Dest) &&
        in-circle(e.Org,t.Dest,e.Dest, p)) {
      swapedge(e); e = e.Oprev;
    } else if (e.Org == first) { last = e; return; }
    else { e = e.Onext.Lprev; } }
}
```

Figure 2.9: Pseudo-Code for the incremental algorithm.

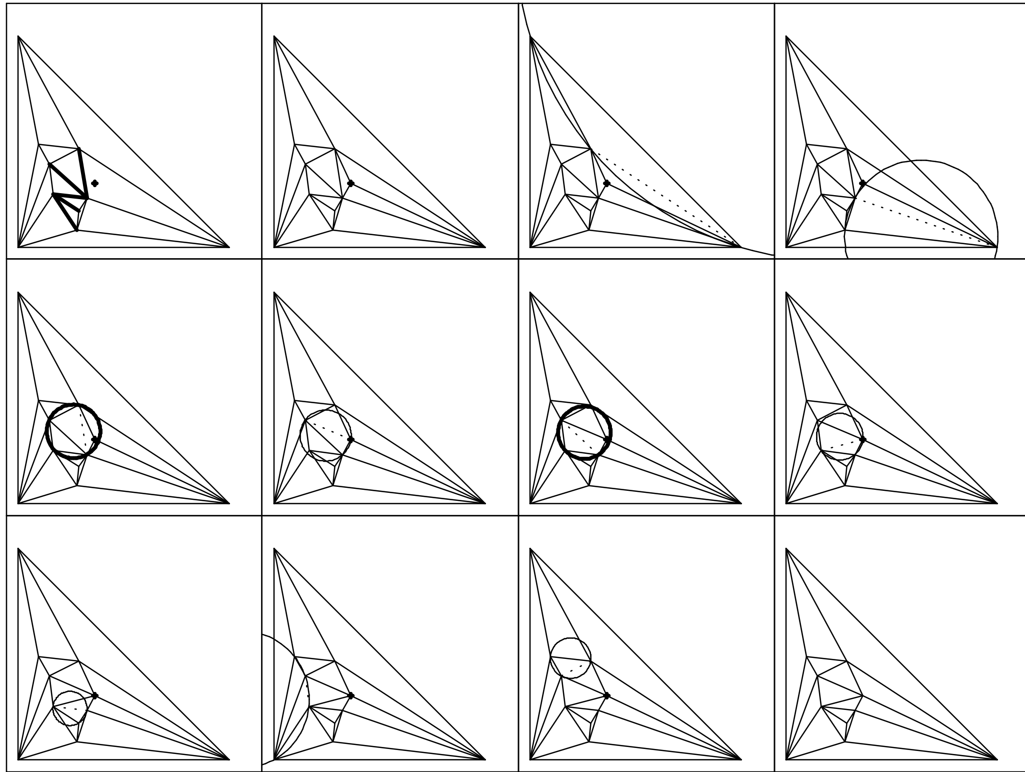


Figure 2.10: The incremental algorithm.

diagram. Then, we start to test the validity of the edges surrounding p. The dotted edges are the ones being tested. The first two circles are empty and drawn with thin lines. The next circle contains p and is drawn in bold. The following frame shows the flipped edge as dotted and the corresponding new circle, which is now empty. The algorithm now moves on to the edges neighboring the edge that it just flipped. The next two frames show another edge flip. The next three edges are all valid. In the last frame, the algorithm has reached the starting edge, so the insertion is complete.

The worst case runtime of this algorithm is $O(n^2)$ since it is possible to construct a point set and insertion order where inserting the kth point into the diagram causes $O(k)$ updates. However, if the points are inserted in a random order, Clarkson and Shor's analysis [CS89] shows that one can design an algorithm with an expected runtime of $O(n \log n)$ operations. In addition, the performance of the algorithm is independent of the distribution of the input, and only depends on random choices made by the algorithm.

The algorithm works by maintaining the current diagram along with an auxiliary data structure called a *conflict graph*. The conflict graph is a bipartite graph with edges connecting sites to triangles in the current diagram. If an edge (s, t) exists, it means that the site s lies within the circumcircle of the triangle t. When a new site s is added to the diagram, the algorithm deletes all the triangles that s conflicts with, adds new triangles in their place, and updates the conflict graph. The standard incremental algorithm could be modified to do this by adding code to update the conflict graph after each edge flip. Clarkson and Shor use random sampling results to show that the total expected runtime of the algorithm is $O(n \log n)$ steps.

Guibas, Knuth, and Sharir [GKS92] propose a similar algorithm that does not use a conflict

graph, and provide a simpler analysis of its runtime. In particular, they show when we use a random insertion order, the expected number of edge flips that the standard incremental algorithm performs is linear. The bottleneck in the algorithm then becomes the `Locate` routine. Guibas, Knuth, and Sharir propose a tree-based data structure where internal nodes are triangles that have been deleted or subdivided at some point in the construction, and the current triangulation is stored at the leaves. Whenever a triangle is deleted, it marks itself as such and then creates two pointers to the triangles that replaced it in the diagram. We can then modify `Locate` to search this tree, rather than doing the standard edge walk. It is then not hard to show that the expected cost of `Locate` will be $O(n \log n)$ time. Sharir and Yaniv [SY91] prove a bound of about $12nH_n + O(n)$.

Modifying the existing incremental algorithm to maintain this point location structure is not difficult. The easiest way to do this would be to keep a separate tree structure with links back to the current quad-edge representation of the Delaunay triangulation. However, the resulting algorithm is somewhat unsatisfactory, since it uses $O(n \log n)$ expected time on point location when actually maintaining the Delaunay triangulation only costs $O(n)$ expected time.

We can remedy this situation if we assume that the input consists of points from some probability distribution. For example, if the distribution is the uniform distribution over the unit square, many known algorithms run in linear expected time [BWY80, KK87, Mau84, OIM84]. The motivation for studying this case comes from the fact that many applications involve data that is fairly uniform. In the next section, we will see how to modify the randomized incremental algorithm to run in linear expected time on such data.

2.4 A Faster Incremental Construction Algorithm

The goal is to speed up point location in the randomized incremental algorithm when the input is uniformly distributed. We concentrate on point location because under the assumption of a random insertion order, this is the dominant cost of the incremental algorithm. We use a simple bucketing algorithm similar to the one that Bentley, Wiede and Yao used for finding the nearest neighbor of a query point [BWY80]. The idea is that point location in the triangulation is equivalent to nearest neighbor searching. Since the bucketing algorithm finds the nearest neighbor of a point in constant expected time for certain distributions of points, we can reduce the total expected cost of the point location steps from $O(n \log n)$ time to $O(n)$ time while maintaining the relative simplicity of the incremental algorithm.

The bucketing scheme places the sites into a uniform grid as it adds them to the diagram. If two or more points fall in the same bucket, the last one inserted is kept and the others are discarded. To find a near neighbor, the point location algorithm first finds the bucket that the query point lies in and searches in an outward spiral for a non-empty bucket. It then uses any edge incident on the point in this bucket as a starting edge in the normal point location routine. If the spiral search fails to find a point after a certain number of layers, the point location routine continues from an arbitrary edge in the current triangulation.

The bucketing scheme uses a dynamic hash table to deal with the fact that sites are processed in an on-line fashion. The scheme also does not bother to store all the points that fall in a particular bucket, it just stores the last point seen. This is because the bucket structure does not have to provide the insertion algorithm with the true nearest neighbor of a query, it only has to find a point that is likely to be close the the query. Therefore, it makes sense not to use the extra space on information that we do not need. Let $c > 0$ be some small constant that we can choose later. The point location maintains the bucket table so that on average, c sites fall into each bucket. It does this on-line by

```
Build_DT()
{
  grid_size := 4;
  maxl := 1;
  N := 0;
  for each site S {
    insert S into the diagram;
    bucket S in the current grid;
    if (++N > 4*c*grid_size) {
      make a new grid of size 4*grid_size;
      re-bucket all points;
      grid_size *= 4;
      maxl++;
    }
  }
}

Spiral(p)
{
  bucket := compute_bucket(p);
  layer := 0;
  while (layer < maxl) {
    for each bucket in layer {
      if P is a point in this bucket return P.edge;
    }
    layer++;
  }
  return NULL;
}
New_Locate(p)
{
  edge := Spiral(p);
  if (edge == NULL) {
    edge = some arbitrary edge;
  }
  Locate(edge, p);
}
```

Figure 2.11: The incremental algorithm with spiral search for point location.

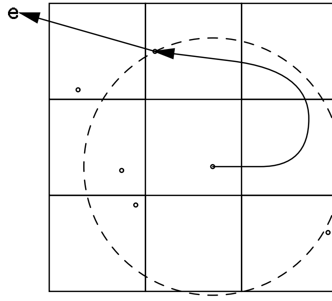


Figure 2.12: Spiral search for a near neighbor.

expanding the size of the bucket grid periodically (see Figure 2.11).

Now, we modify `Locate` to first search the bucket structure for a near neighbor, and then use some edge incident on that neighbor as a starting edge in a standard edge walk. The search procedure starts in the bucket that the new site lies in and searches out in a spiral pattern until has searched more than `maxl` layers (see Figure 2.12). If it still hasn't found a non-empty bucket at this point, it gives up and `Locate` proceeds from an arbitrary edge. Assuming that each point `P` keeps track of an incident edge in `P.edge`, the modified point location routine is shown in Figure 2.11.

The bucketing procedure makes sure that the expected number of points per bucket is always at least c and at most $4c$. The variable `maxl` makes sure that the routine never looks at more than $\log_4 n$ layers of buckets. Thus, it is not hard to show that the expected cost of one point location is constant when the points are uniformly distributed. The probability that a point falls into a particular bucket is $q = c/n$. The probability that a particular bucket is empty is then

$$(1-q)^n = (1-c/n)^n \leq e^{-c}$$

The cost of the point location algorithm for a given query point is proportional to the number of buckets examined in the spiral search and the number of edges examined in the locate step following the spiral search. There are two cases to consider. First, the spiral search may succeed before the cutoff point, after examining i buckets in the first $\log_4 n$ layers. Referring to the picture, the analysis of Bentley, Weide and Yao says that the expected number of points in the dashed circle is bounded by $8c(11i+7)$. The expected number of edges in the area is bounded by six times the expected number of points. Thus, the expected number of edges that locate examines is less than $48c(11i+7)$. The total cost of the procedure in this case is then bounded by

$$O(1) \sum_{i \leq 4 \log^2 n} e^{-c(i-1)} O(i) = O(1).$$

If the spiral search fails, then the cost of the locate procedure is no more than $O(n)$ operations. But, since the algorithm must look at $O(\log^2 n)$ buckets before the spiral search fails, the probability of this happening is bounded by $e^{-c \log^2 n} = O(1/n)$, so the expected cost is also $O(1)$ operations.

The cost of maintaining the buckets is $O(n)$ if the floor function is assumed to be constant time. Since we insert the points in a random order, the analysis of Guibas, Knuth and Sharir [GKS92] shows that expected number of edge flips that the algorithm does is $O(n)$. Thus, the total expected run time of this algorithm is $O(n)$ time when the points are uniformly distributed in the unit square.

The two important principles at work here are that "bottom up" search from the bucket structure to the current triangulation takes advantage of the local nature of nearest neighbor search, and that

```
// Q is a dictionary of (facet, half-space) pairs.
Construct_DT:
  Find an initial cell C.
  For each facet F of C, let H(F) be the half-space defined by F that
    does not include the interior of C.  Insert (F, H(F)) into Q.
// Now iteratively advance the front until it is empty.
  While Q is not empty
    let (F,H) be some element of Q.
// Site-search
    search H for the site X that will complete F's unknown cell.
    let C be the new cell.
    delete (F,H) from Q.
    For each vertex V in F
      let newF be the facet defined by (F-{V}) and X
      let newH be the half-space defined by newF not containing V.
// Facet search
      if (newF,newH) is not finished
        insert (newF,newH) in Q
      else
        mark (newF,newH) as finished
```

Figure 2.13: Algorithm IS.

the cost of adapting the bucket grid as more points are added can be amortized over the whole construction process. The resulting algorithm is only slightly more complicated than the original incremental algorithm, but as we will see, in most practical situations, it is nearly as fast as all of the methods that we have seen so far.

2.5 Incremental Search

Another class of incremental algorithms construct the Delaunay triangulation by incrementally discovering valid Delaunay triangles, one at a time. Although we are only dealing with two dimensional problems, it will be convenient to use a generic terminology for any dimension. Following Dwyer [Dwy91], in d dimensions, our algorithms will assume that no set of more than $d+1$ sites are co-spherical. In this case, the Delaunay triangulation partitions the convex hull of S into simplices that we will call *cells*, and the circumsphere of each cell contains no site. The d-dimensional simplices will be called *facets*. In the plane, cells are triangles, facets are edges, the circumcircle of each triangle is empty, and no four sites are co-circular.

Figure 2.13 shows pseudo-code for the incremental search algorithm, which we will call Algorithm IS. The algorithm constructs one simplex of the diagram and then proceeds to extend this to the whole diagram by adding new cells adjacent to known facets one at a time. This algorithm works by keeping a queue of facets and half-spaces. Each facet joins two cells, so we call a facet *finished* if both of these cells have been discovered by the algorithm. The algorithm stores the facets that it has found in a dictionary along with a flag indicating whether it is finished. Each time a new cell is discovered, the unfinished facets that belong to the cell are placed on a queue. The queue represents a front which advances over the point set as new cells are discovered.

In the plane, facets are edges and cells are triangles. Thus, the algorithm starts with one triangle

and incrementally finds new triangles. The advancing front is just a queue of unfinished edges. In three dimensions, the algorithm is basically the same, except that it manipulates tetrahedron and faces.

The performance of this algorithm is determined by the time needed for facet search and the time needed for site search. If the facet dictionary is stored as a hash table, then the expected cost of maintaining the dictionary is $O(n)$ time. The data structure that is critical to the performance of the algorithm is the one that supports site searches. The next section will describe a reasonable solution to this problem in the case where the site are uniformly distributed in the unit square. In order to generalize the algorithm for other classes of inputs, and to higher dimensional problems, all we need to do is replace the `Site-Search` routine with something more appropriate.

2.5.1 Site Search

Dwyer's algorithm uses a relatively sophisticated algorithm to implement site searching. First, the sites are placed in a bucket grid covering the unit square. The search begins with the bucket containing the midpoint of (a, b). As each bucket is searched, its neighbors are placed into a priority queue that controls the order in which buckets are examined. Buckets are ordered in the queue according a measure of their distance from the initial bucket. Buckets are only placed in the queue if they intersect the half-plane to the right of (a, b) and if they intersect the unit circle. When the correct site is found, it will be closer to the edge (a, b) than any of the other buckets in the queue, so the queue will empty out and the algorithm terminates. Dwyer's analysis shows that the total number of buckets that his algorithm will consider is $O(n)$.

Our site search algorithm is a variant on "spiral search" [BWY80]. A new site is found by searching outward from an unfinished edge in a spiral pattern. We will assume that the the sites in S are chosen from the uniform distribution in the unit square, and, like Dwyer, we will use a bucket grid over the unit square to help speed up the search process. With n points, the data structure is a uniform grid of $\sqrt{n}/c \times \sqrt{n}/c$ buckets, for some constant $c \geq 1$. Generally, we choose c to tune the performance of a particular implementation. To simplify our analysis, we assume that $c = 1$.

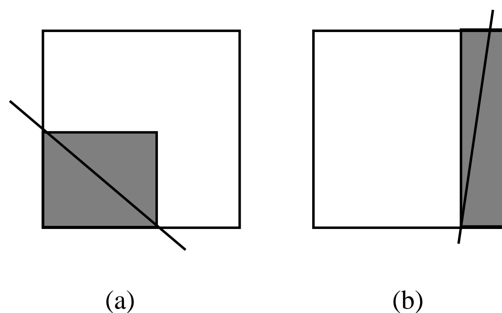
Figure 2.14 shows a high level description of the site search routine. The routine first computes the mid-point of the edge (a, b) and begins looking for the new site in the bucket B that the midpoint falls in. The algorithm checks each site in B and picks the one forming the smallest circle, C, with (a, b). The algorithm then expands the range of the search to include the boxes surrounding B. This expansion continues until the either range of boxes searched by the algorithm contains the bounding box of the current "best" circle, or until it contains the intersection of the half-plane to the right of (a, b) and the unit square. In will have either found the correct site to connect to (a, b), or will have shown that the edge (a, b) is on the boundary of the convex hull of S. Figure 2.14 summarizes the details of the algorithm.

Our site search routine differs from Dwyer's in several ways. The main difference is the lack of a priority queue to control the action of the search routine. Dwyer's algorithm is careful not to inspect any buckets that are either to the left of (a, b) or outside of the unit circle. In this case, he can prove that his algorithm only does $O(n)$ work.

`Site-search` is somewhat sloppier about looking at extra buckets. The algorithm examines at least every bucket in the intersection between the unit square and the bounding box of the final circle discovered. It is possible that it will search a larger area than this if some intermediate circle is much larger than the final one. For site searches near the center of the unit square, the search area expands uniformly in all directions. For site searches near the edge of the unit square, the

```
// Search half-space to the right of (a,b) for new site.
// circle(C,r2)  == circle centered at C with radius sqrt(r2);
// C.r2                   == radius square of circle C;
// dist2(C, p)            == square distance from center of C to p;
// right-of(a,b,P)        == P is to the right of the segment (a,b);
// new-circle(a,b,P)      == circle through a,b and P
// Box(B, L, R, U, D)   == box centered at B and extending
//   L layers to the left, R layers to the right, etc
//
Site-search (a,b)
  queue = empty;
  m = midpoint of (a,b);
  B = bucket(m);
  r2 = squared distance from m to (a,b)
  C = circle(m, r2);
  H = half-plane to the right of a,b;
  plane-box = intersection between H and the unit square;
  thesite = nil;
  search B;
  searchbox = Box(B,1,1,1,1);
  while (not done)
    foreach unseen bucket in searchbox
      if bucket intersects H
        foreach point p in bucket
          if (right-of(a,b,p) && dist2(C,p) < C.r2)
            C = new-circle(a,b,p);
            thesite = p;
    if thesite == nil
       done = searchbox is contained in plane-box;
    else
       done = bounding box of C is contained in searchbox;
    if not done
       expand searchbox
```

Figure 2.14: Code for Site Search.



(a) (b)

Figure 2.15: Example bounding boxes that the site search algorithm examines.

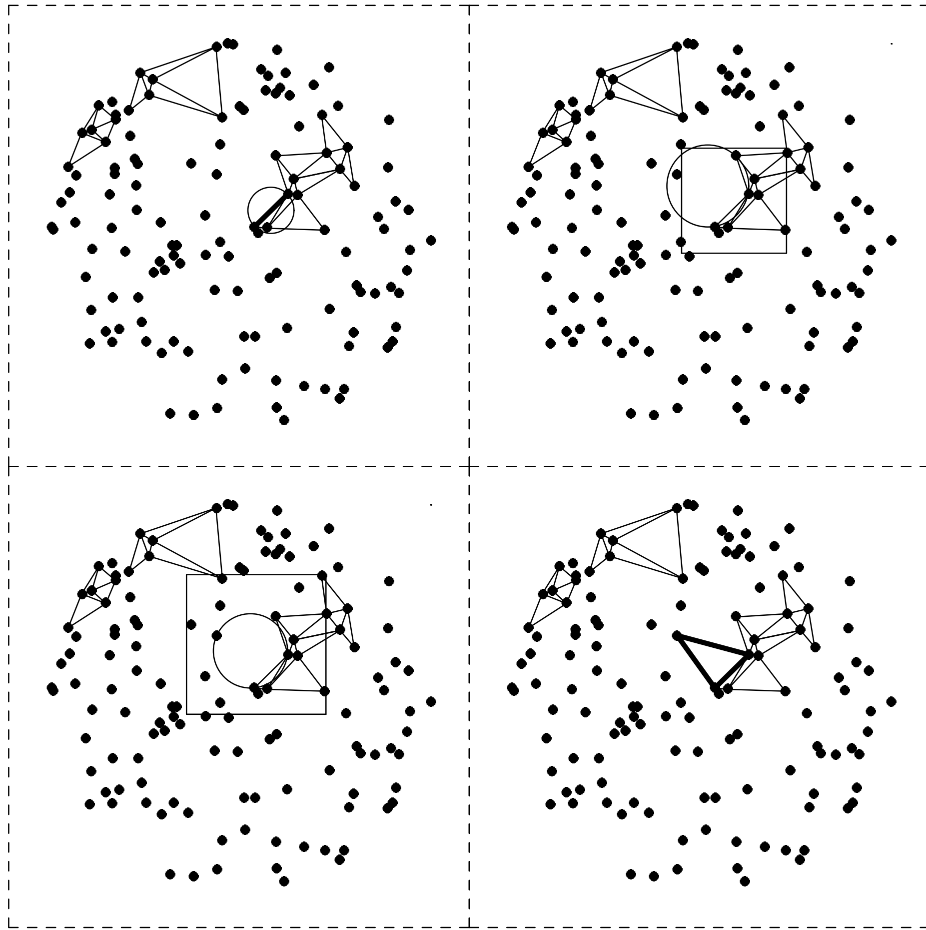


Figure 2.16: Site search in Algorithm IS

algorithm does not needlessly expand the seach area in a non-profitable direction (see Figure 2.15). The advantage of our scheme is that it is well tuned to the case when sites are distributed in the unit *square* and it avoids the extra overhead of managing a priority queue, especially avoiding duplicate insertions.

Figure 2.16 illustrates the operation of `Site-search`. In the top left frame, the site search algorithm is initialized to find the unknown site of the bold edge. This site lies above and to the left of the edge. The top right frame shows the first candidate site and the resulting circle circle found by `Site-search`. The box in the frame is the area searched so far. In the bottom left frame, `Site-search` expands the search area, and finds a second candidate site and circle. The box shows that the algorithm has now searched an area large enough to prove that the second circle is empty. Therefore, two new edges, shown in bold in the bottom right frame, are added to the queue. The operation of this algorithm is actually reminiscent of Fortune's algorithm, although the two use totally different methods to control the evolution of the expanding front.

2.5.2 Discussion

Several algorithms in the literature, including ones due to Dwyer [Dwy91], Maus [Mau84], and Tanemura et. al. [TOO83] are based on the basic idea behind Algorithm IS. However, they all

differ in the details, and only Dwyer's algorithm has been formally analyzed. Dwyer's analysis assumed that the input was uniformly distributed in the unit d-ball, and extending this analysis to the unit cube appears to be non-trivial. However, in the plane, the difference is not great, and the experiments in the next section will show that the empirical runtime of the algorithm appears to be $O(n)$.

2.6 Empirical Results

In order to evaluate the effectiveness of the algorithms described above, we will study C implementations of each algorithm. Rex Dwyer provided code for his divide and conquer algorithm, and Steve Fortune provided code for his sweepline algorithm. I implemented the incremental search and construction algorithms myself. None of the implementations are tuned in any machine dependent way, and all were compiled using the GNU C compiler and timed using the standard UNIX$^{\text{tm}}$ timers. The other performance data presented in this section was gathered by instrumenting the programs to count certain abstract costs. A good understanding of each algorithm, and profiling information from test runs determined what was important to monitor. Each algorithm was tested for set sizes of between 1024 and 131072 sites. Ten trials with sites uniformly distributed in the unit square were run for each size, and the graphs either show the median of all the sample runs or a "box plot" summary of all ten samples at each size. In the box plots, a dot indicates the median value of the trials and vertical lines connect the 25th to the minimum and the 75th percentile to the maximum.

Finally, the programs include code to generate simple animations of their operation. This code outputs simple graphics commands that can be interpreted either by an interactive tool for X windows or a Perl program that generates input for various typesetting systems. These "movies" were very helpful in gaining intuition about every detail of the operation of the various algorithms, and also made debugging much easier. The movies in this chapter were all generated automatically by the animation scripts.

2.6.1 Performance of the Incremental Algorithm

The performance of the incremental algorithm is determined by the cost of point location and the number of circle tests the algorithm performs. While the standard incremental algorithm spends almost all of its time doing point location, the bucket-based point location routine effectively removes this bottleneck. Figure 2.17 compares the performance of the two algorithms on uniformly distributed sites.

The plots show that the standard incremental algorithm uses an average of $O(\sqrt{n})$ comparisons per point location step. Simple regression analysis indicates that the number of comparisons per point grows as $0.86n^{.49}$. Therefore, I have plotted the curve $.86\sqrt{n}$ in with the data. The curve is a close fit to the experimental data.

Adding the point location heuristic improves the performance of the incremental algorithm substantially. It is apparent that the number of comparisons per site is bounded by a constant near 12. Almost all of these comparisons are tests done during second phase of the algorithm, after the spiral search. These tests take much longer than the simple comparisons needed in spiral search. Therefore, although the spiral search accounts for about 10% of the total number of comparisons, its contribution to the runtime of the point location algorithm is much less significant.

The cost of point location depends on whether $\log_2 n$ is even or odd. This is easy to explain when we remember that the algorithm re-buckets the sites at each power of four. Because of this, at each power of four, the average search time dips, since one extra re-bucket step reduces the average

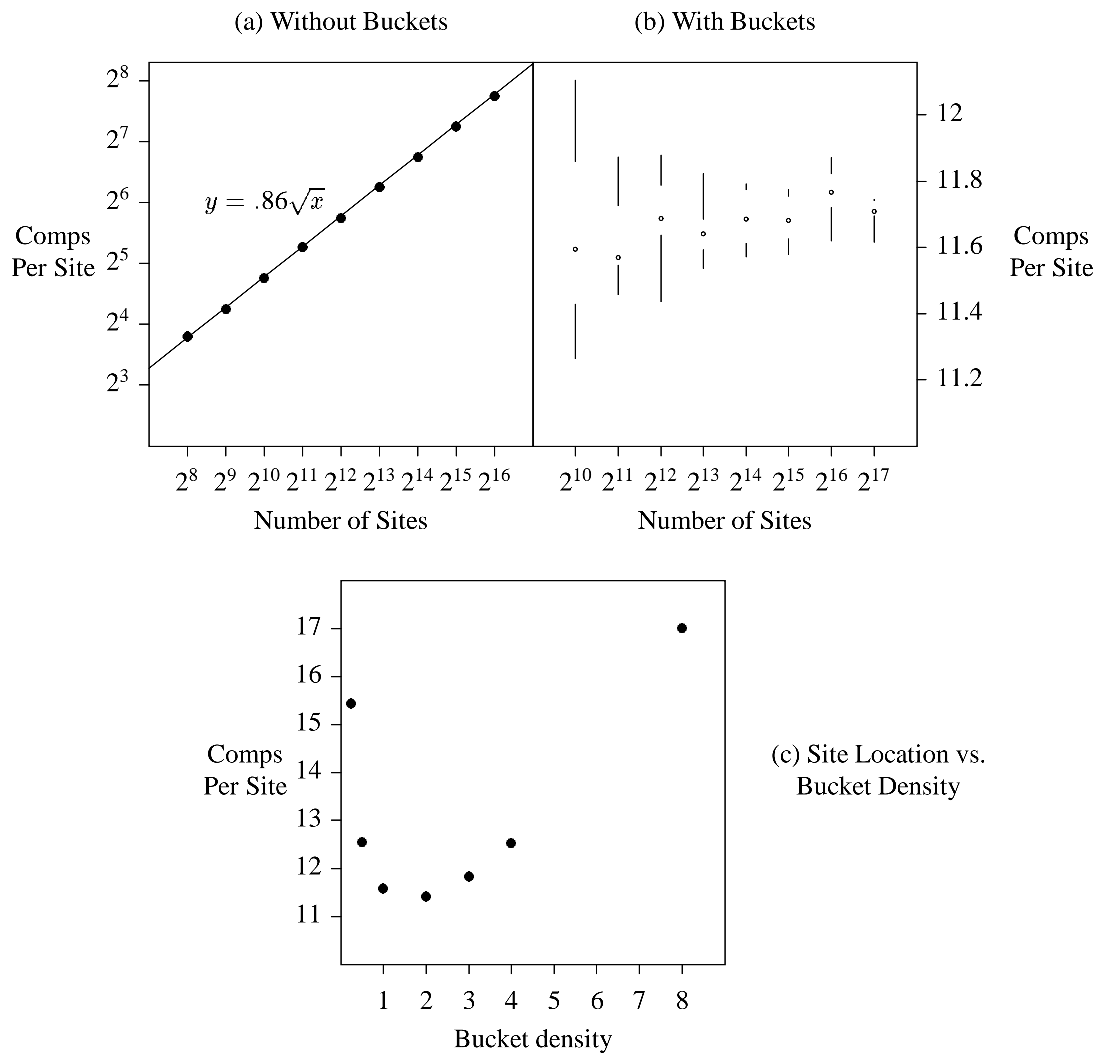


Figure 2.17: Comparison of point location costs.

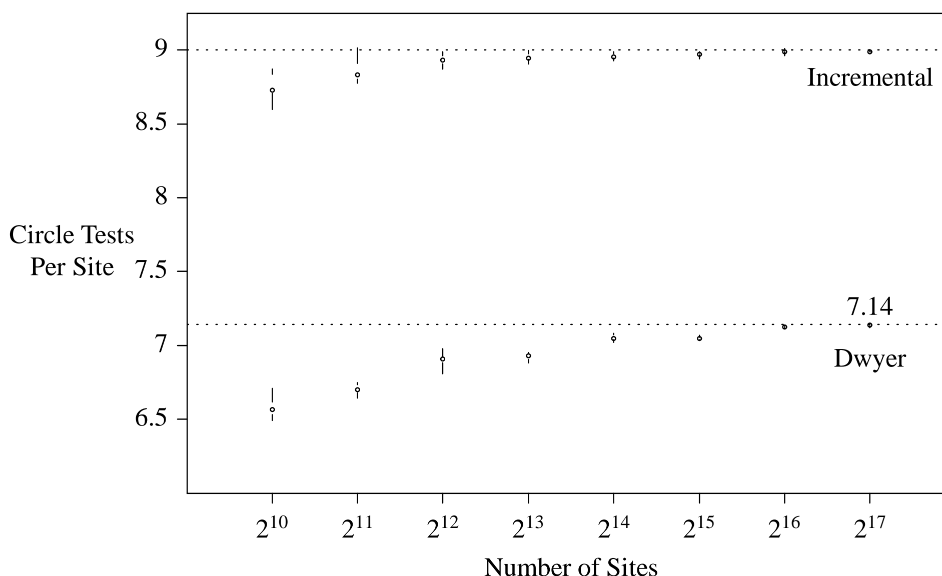


Figure 2.18: Circle tests per site for two algorithms.

bucket density in the later phases of the construction process. As the input size moves towards the next power of four, the bucket density increases steadily. Thus, the cost of point location see-saws up and down.

The number of comparisons needed for point location also depends on the average density of the sites in the bucket grid. If the density of points in buckets is too high, then the point location routine will waste time examining useless edges. On the other hand, if it is too low, the algorithm will waste time examining empty buckets. Figure 2.17c shows the dependence of the density on the cost of point location. The graph shows the average cost of point location over ten trials with $n = 8192$ and c ranging from 0.25 to 8. Based on this data, I used $c = 2$ for my timing runs. Although the graph shows a large variation in the number comparisons needed per site, the actual effect on the runtime of the algorithm was less than 10%.

The runtime of the incremental algorithm now depends on how many circle tests it performs. Since the algorithm inserts the points in a random order, the analysis of Guibas, Knuth and Sharir [GKS92] shows that the total number of circle tests is asymptotically $O(n)$. Sharir and Yaniv [SY91] tightened this bound to about $9n$. Figure 2.18 shows that this analysis is remarkably accurate.

Also shown on the plot is the cost of Dwyer's divide and conquer algorithm. Since this algorithm is also based primarily on circle testing, it makes sense to compare them in this way. The plot shows that Dwyer's algorithm performs about 25% fewer circle tests than the incremental algorithm. Profiling both programs shows that circle testing makes up roughly half the runtime of each, so Dwyer's algorithm should run roughly 10 to 15 percent faster than mine.

2.6.2 The Incremental Algorithm with a Quad-Tree

Ohya, Iri and Murota [OIM84] describe a modification to the incremental algorithm that buckets the points like my algorithm does but then inserts the points using a breadth-first traversal of a quad-tree. In their paper, the authors claim that their algorithm runs in expected linear time on sites that are uniformly distributed, and they provide experimental evidence for this fact. In order to show that my algorithm was competitive with, or better than other similar methods, I implemented

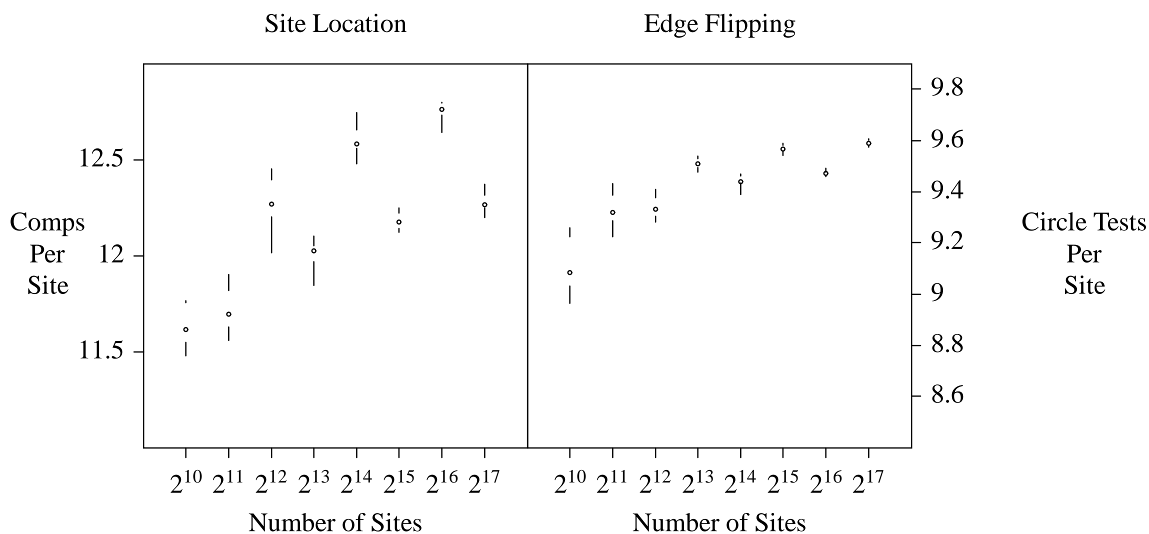


Figure 2.19: Performance of the quad-tree algorithm.

this algorithm and compared its performance with mine.

Like before, a profile of this program showed that the algorithm spends most of its time either testing circles or locating points. Therefore, further experiments monitored these two operations in more detail. These results show that the quad-tree algorithm performs almost identically to my algorithm. Like my algorithm, its performance depends on the parameter c, and in these tests, c was set to two just like before, as this provided the best overall performance. As Figure 2.19 shows, the quad-tree algorithm performs about 10% more work than the bucket-based incremental algorithm. For the uniform case, we can conclude that our algorithm performs slightly better than this more complicated alternative.

2.6.3 The Incremental Search Algorithm

The determining factor in the runtime of Algorithm IS is the cost of `Site-search`. This can be factored into the number of distance calculations performed, and the number of buckets examined. By *examined* we mean that a bucket is at least tested against an edge to see if the algorithm should search its contents for a new site.

Figure 2.20 summarizes the behavior of these parameters in our tests. Figures 2.20a and 2.20b show the performance of the algorithm for sites chosen from both the uniform distribution in the unit square, while 2.20c and 2.20d show the show the performance of the algorithm for sites chosen from the uniform distribution in the unit circle. The reason for looking at both distributions in detail is that the behavior of Algorithm IS is heavily influenced by the nature of the convex hull of the input. In the square distribution, the expected number of convex hull edges is $O(\log n)$ [San76]. The graph shows that the number of distance calculations per site stays constant over our test range, while the number of buckets examined actually decreases. This reflects the fact the number of edges on or near the convex hull of such point sets is relatively small, and that the algorithm only examines a large number of useless buckets on site searches near the convex hull.

However, it is apparent that this isn't the case when the sites are distributed in the unit circle, where the expected size of the convex hull is $O(n^{1/3})$ [San76]. Here, there are a larger number of edges on or near the convex hull, and this is reflected by the fact that the number of buckets that the

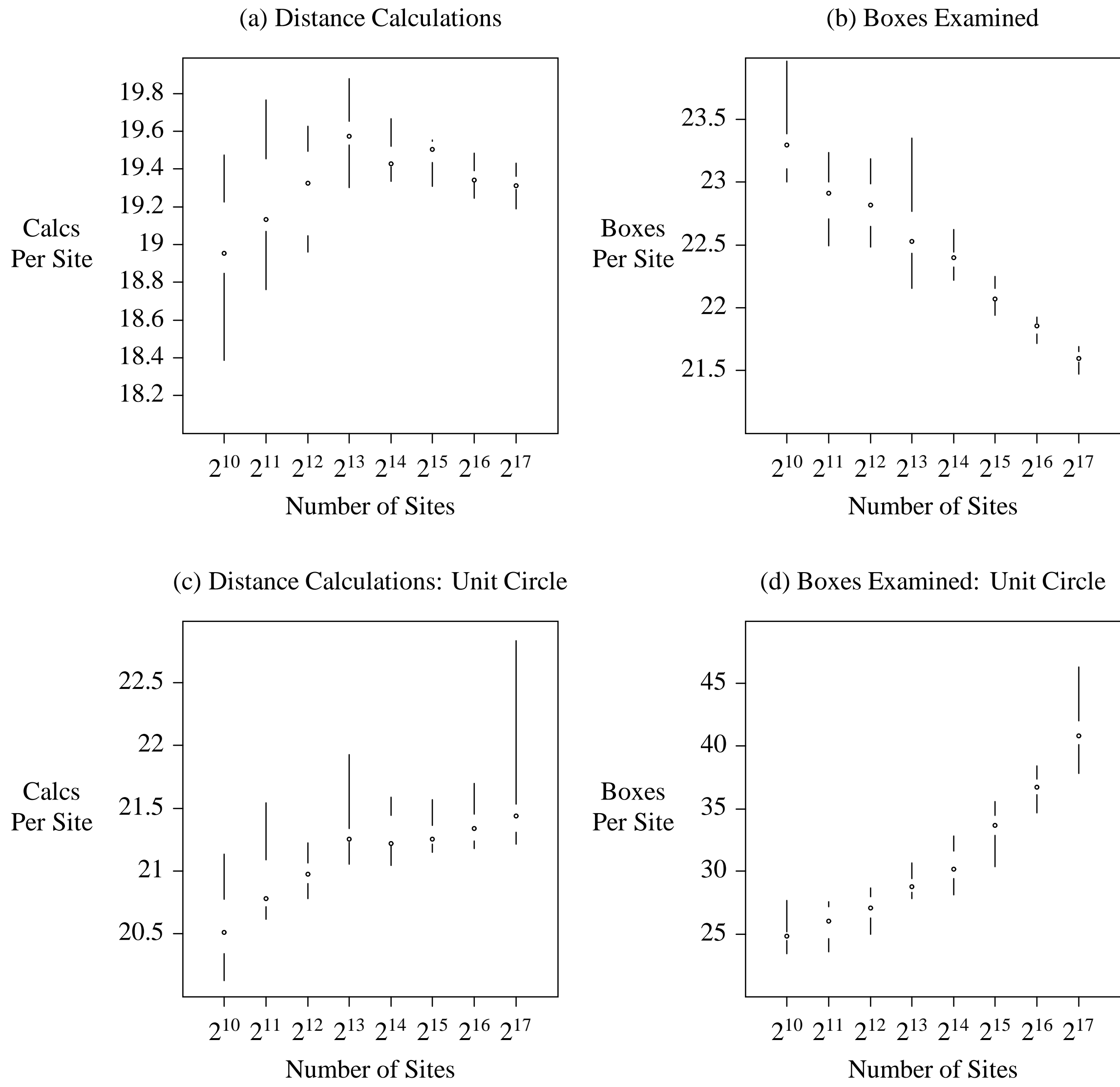


Figure 2.20: Performance of Algorithm IS

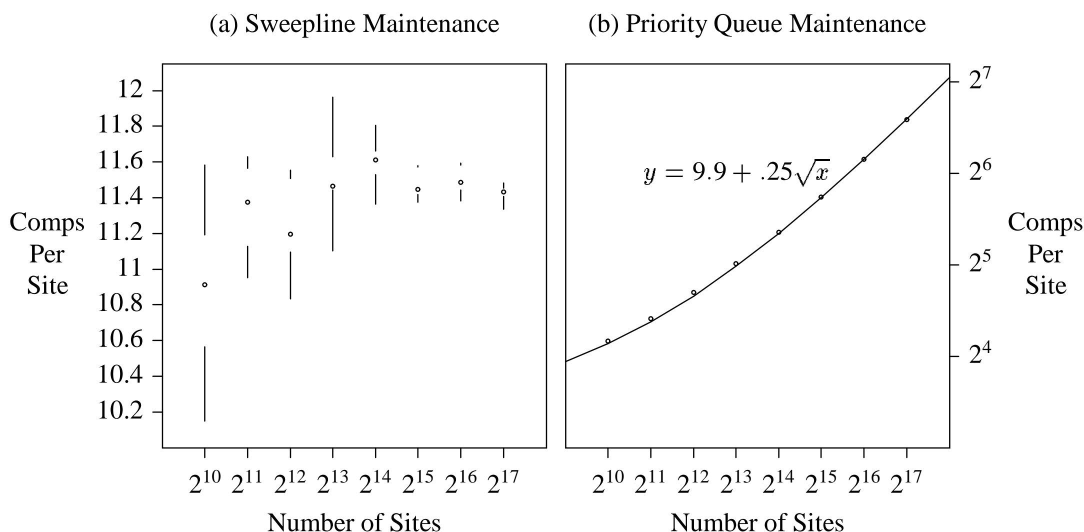


Figure 2.21: Cost of Fortune's algorithm. The two main factors determining the performance of the algorithm are the work needed to maintain the heap and sweepline data structures.

algorithm examines increases dramatically when compared to the earlier case. This sensitivity to the distribution of the input is a major weakness of the algorithm, and indicates that a more adaptive data structure is needed to support the `Site-search` routine.

Pre-computing the convex hull of the point set and searching from hull edges inward may help to minimize the effect of this problem. But, the bucket-based data structure will still be sensitive to clustering in the point set, and a non-uniform distribution of triangle sizes or shapes. The best way to fix these problems is to replace the buckets with a nearest-neighbor search structure that is less sensitive to the distribution of sites, and can perform non-local searches more efficiently. Bentley's adaptive k-d tree [Ben90] is a good example of such a data structure.

2.7 Fortune's Algorithm

The runtime of Fortune's algorithm is proportional to the cost of searching and updating the data structures representing the event queue and the state of the sweepline. Fortune's implementation uses hash tables for this purpose. We would expect that these data structures would perform well on uniform inputs. In fact, for small input sets, the algorithm seems to run in linear time.

Figure 2.21 shows the performance of the sweepline and priority queue data structures in Fortune's implementation. With sites that are uniformly distributed in the x direction, the bucket structure representing the frontier performs exactly as we would expect. Figure 2.21a indicates that the search procedure performs around 12 comparisons per point, on average.

The main bottleneck in Fortune's algorithm ends up being the maintenance of the priority queue. The priority queue is represented using a uniform array of buckets in the y direction. Events are hashed according to their y-coordinate and placed in the appropriate bucket. In addition, it is important to realize that only circle events are explicitly placed in the event queue. The $O(n)$ site events are stored implicitly by initially sorting the sites.

The problem here is that while the sites are uniformly distributed, the resulting priorities are not. Circle events tend to cluster close to the current position of the sweepline. This clustering increases the cost of inserting or deleting events into Fortune's bucket structure. Each operation

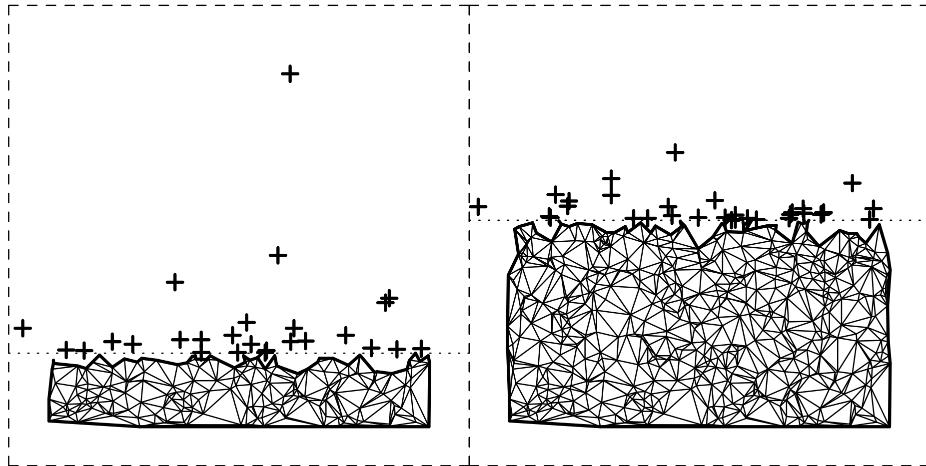


Figure 2.22: Circle events cluster close to the sweepline in Fortune's algorithm. The first frame is early on one run in the algorithm, the second frame is later in the same run. Note how the "cloud" of circle events (+ signs) moves with the sweepline.

requires a linear time search through the list of events in a particular bucket. With large buckets, this becomes expensive. Regression analysis shows that the number of comparisons per point grows as $9.95 + .25\sqrt{n}$ (see Figure 2.21b).

Watching animations of large runs of Fortune's algorithm provides a heuristic explanation for this behavior. Since the sites are uniformly distributed, new site events tend to occur close to the frontier. If the new site causes a circle event to be added to the priority queue, chances are that the circle will not be large, and thus the y-coordinate of the top of the circle, which is the priority of the new event, will be close to the current position of the sweepline. If the circle is large, so the priority of the resulting event is far above the sweepline, it is likely that the event is invalid since large circles are likely to contain sites. Eventually some site or circle event will invalidate the large circle and replace it with a smaller one that lies closer to the sweepline. The result is the clustering that is clearly observable in the animations (Figure 2.22).

Given the behavior of the bucket data structure, it is natural to speculate as to whether a different data structure would provide better performance for larger problems. To investigate this possibility, I re-implemented Fortune's algorithm using an array-based heap to represent the priority queue. This guarantees that each operation on the priority queue costs $O(\log n)$ time in the worst case, and using the array representation minimizes additional overhead.

To test the effectiveness of the new implementation, I performed a more extensive experiment. Each algorithm was tested on uniform inputs with sizes ranging from 1,000 to 10,000 sites. Figure 2.23a shows the performance of the heap data structure in the experiment. The line $2\lg n + 11$ shows that the number of comparisons used to maintain the heap is growing logarithmically in n, rather than as $\sqrt{n}$. The plot also shows a more detailed profile of these comparisons. This profile indicates that most of the work is performed by the `extract-min` routine. By comparison, `insert` and `delete` are relatively cheap. This behavior is exactly opposite to the bucket structure.

In actual use, the heap does not significantly improve the runtime of the algorithm. Figure 2.23b compares the runtime of the two algorithms over the same range of inputs. In this graph, each data point is the ratio of the runtime of the bucketing algorithm to the runtime of the heap-base algorithm. The graph shows five trials for input sizes of between 2^{10} and 2^{17} sites at evenly spaced

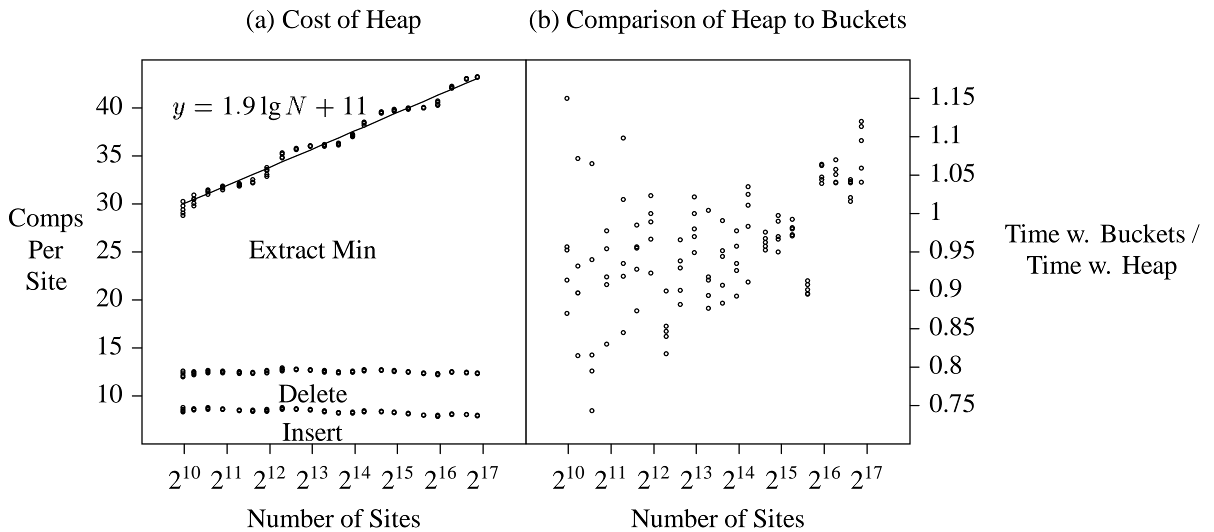


Figure 2.23: Cost of Fortune's algorithm using a heap. Part (a) shows a breakdown of the cost of using a heap to represent the event queue. Part (b) plots the ratio of the runtime of the old algorithm to the new.

intervals.

The timings were taken using the same machine and configuration as in the next section. The plot shows that the algorithms are pretty much identical in performance until 2^{16} points, when the heap version starts to dominate. At 2^{17} points, the heap version is roughly 10% better. The main reason that the improvement is not greater is that maintaining the heap seems to incur more data movement overhead than maintaining the bucket structure. The bucket structure appears to use the workstation's cache more effectively, and stays competitive, even though it is doing much more "work".

An interesting feature of the graph in Figure 2.23a is the fact that the number of comparisons periodically jumps to a new level, stays relatively constant, then jumps again. This is due to the fact that the heap is represented using an implicit binary tree. Thus, the number of comparisons jumps periodically when the size of the heap is near powers of two. On the graph, these jumps occur at powers of four, rather than two because the average size of the heap over one run of Fortune's algorithm is $O(\sqrt{n})$ rather than $O(n)$. To prove this, we start with the following definition.

Definition 2.1. Let S be a set of n sites, and suppose that the sweepline in Fortune's algorithm is at height y. For each site s whose y-coordinate, $Y(s)$ is less than y, we say that s is **unfinished** if the algorithm has not yet computed all the edges in the final Delaunay triangulation incident on s.

If a site is incident on a circle in the event queue, it must either be unfinished or on the convex hull of S. For sites uniformly distributed in the unit square, the expected size of the convex hull is $O(\log n)$ [Dwy88], so to bound the expected size of the priority queue, we only need to estimate the number of unfinished sites. The following lemma is a special case of a result due to Katajainen and Koppenin. The proof is repeated here in a slightly simpler form for the sake of completeness [KK87].

Lemma 2.1. Assume that S consists of n points chosen from the uniform distribution in the unit square. If s is a site whose distance from the sweepline is t, then the probability that s is unfinished

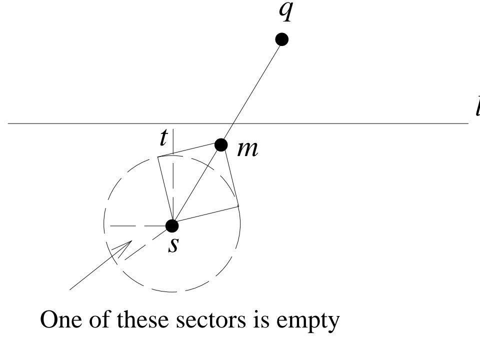


Figure 2.24: Delaunay edges imply empty regions of the plane.

is at most

$$16(1 - \pi t^2/32)^{n-1}.$$

Proof. Let UF denote the condition that the site s is unfinished. Then there exists a point q above the sweepline such that (s, q) is in the final Delaunay triangulation. Consider a circle C centered at s with radius $t/\sqrt{2}$. Divide C into 16 sectors, each with an internal angle of $\pi/8$. Now consider a square with side $t/\sqrt{2}$ with its diagonal along (s, q). Let m be the other endpoint of this diagonal. Dwyer [Dwy87] proved that there exists a site free circle that contains one of the two triangles incident on (s, m) (see Figure 2.24). This circle also contains one of the 16 sectors. The area of each of the sectors is $\pi t^2/32$. Thus, there exists a region with area $\pi t^2/32$ that is site free. So we have

$$Pr[UF] <= \sum_{i=1}^{16} (1 - \pi t^2/32)^{n-1},$$

which is bounded by $16(\pi t^2/32)^{n-1}$. □

Using this result, we can estimate the expected number of unfinished sites, given that the sweepline has reached height y.

Theorem 2.2. Given the same assumptions as above, the expected number of unfinished sites is $O(\sqrt{n})$.

Proof. Katajainen and Koppenin use Lemma 2.1 to show that the expected number of unfinished sites in a rectangle of height h and width w is bounded by $103(h + w)\sqrt{n}$ [KK87]. Thus, if the sweepline has reached height y, the number of unfinished sites will be bounded by $103(y + 1)\sqrt{n} <= 206\sqrt{n} = O(\sqrt{n})$. Of course, this constant is much higher than the constant that we actually see in practice. □

Since each unfinished site can contribute to at most two circle events, the expected number of circle events is also $O(\sqrt{n})$.

2.8 The Bottom Line

The point of all of this is, of course to develop an algorithm that has the fastest overall runtime. In the following benchmarks, each algorithm was run on ten sets of sites generated at random from the uniform distribution in the unit square. Each trial used the same random number generator, and the same initial seed so all of the times are for identical point sets. Run times were measured on a

Sparcstation 2 using the `getrusage()` mechanism in UNIX. The graphs show user time, not real time, and all of the inputs sets fit in main memory, and and were generated in main memory so I/O and paging would not affect the results. Finally, the graph for Fortune's algorithm shows the version using a heap rather than buckets, since this algorithm was better on the larger problems, and not much worse on smaller ones.

Figure 2.25 shows the results of the timing experiments. The graph shows that Dwyer's algorithm gives the best performance overall. Fortune's algorithm and the incremental algorithm are about the same, but for large problems the incremental algorithm does better. Fortune's algorithm does the worst on the largest problems due to the overhead in its priority queue data structure. Algorithm IS is not competitive with any of the other three because the simple bucket-based data structure that it uses for site searching is not efficient enough.

Dwyer's algorithm is faster because it does almost $1/3$ fewer circle tests than the incremental algorithm. Profiling information shows that each algorithm spends about half of its total time in circle testing, so this accounts for the fact that Dwyer's program is about ten to fifteen percent faster than the incremental. However, although the incremental algorithm is marginally slower than Dwyer's, it is much simpler to code. In addition, the "on-line" nature of the incremental algorithm is useful in some applications.

The quad-tree order incremental algorithm is slower than my algorithm and Dwyer's algorithm, but is somewhat faster than the others. Recall that this algorithm performs roughly 10% more edge tests in point location and 5-10% more circle tests than the randomized incremental algorithm. These factors account for the its consistently slower runtimes.

Finally, the incremental search algorithm is clearly slower than all the other algorithms. It is clear that the constant factors in the runtime of `Site-search` are somewhat higher than those of the other algorithms. The main problem is that the bucket grid data structure is most efficient when searching small ranges. This makes it ideal for the role it played in the randomized incremental algorithm because there, only local information was needed. In Algorithm IS, the data structure is called upon to search larger areas of the unit square, and its performance degrades.

2.9 Nonuniform Point Sets

Each of the algorithms that we have studied uses a uniform distribution of points to its advantage in a slightly different way. The incremental algorithm uses the fact that nearest neighbor search is fast on uniform point sets to speed up point location. Dwyer's algorithm uses the fact that Delaunay only sites near the merge boundary tend to be effected by a merge step. Fortune's implementation uses bucketing to search the sweepline. Algorithm IS depends on a uniform bucket grid to support site searching.

From the analysis and experimental experience we gained previously, we would expect that since it can do no more than $O(n \log n)$ operations, Dwyer's algorithm would be the least sensitive to pathological input. Next in line would be the incremental algorithm, which would perform $O(n^{3/2})$ operations on average if the distribution of the sites totally defeated the bucketing strategy.

We have already seen that Fortune's implementation is not efficient for large problems even when the input is uniform. If the input set were extremely irregular, so that most of the priorities fell into just a few buckets, we would expect the performance of the implementation to be even worse than the incremental algorithm.

Finally, we would expect that Algorithm IS would be the most sensitive to bad inputs, since its performance is the most closely coupled to the distribution of the sites in the bucket grid.

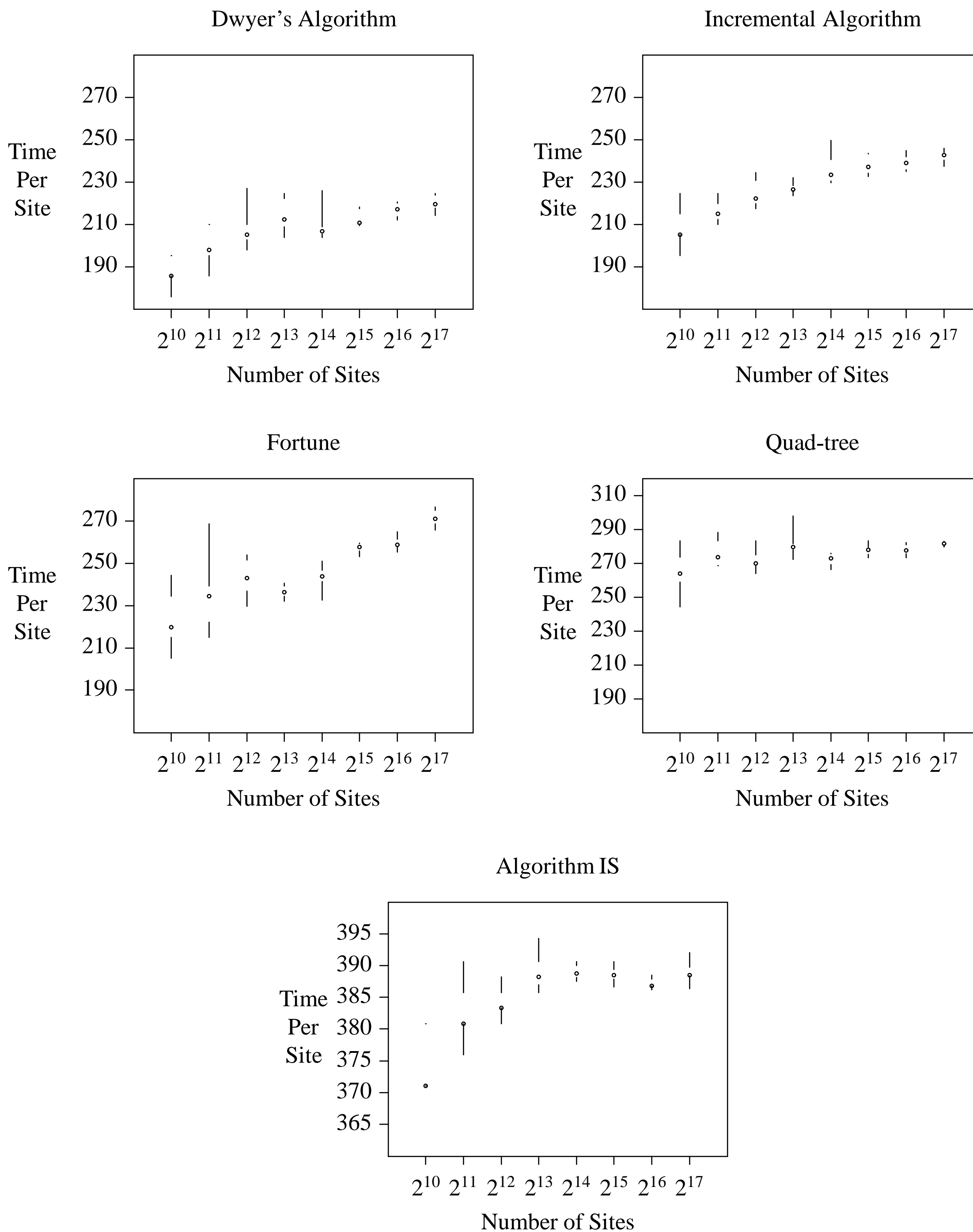


Figure 2.25: Comparison of the expected runtimes of different algorithms on sites chosen at random from a uniform distribution in the unit square. Times are in microseconds.

Name	Description
unif	uniform in the unit square.
ball	uniform in a unit circle.
corners	$U(0, .01)$ at each corner of the unit square.
diam	$t = U(0,1)$, $x = t + U(0, .01) - .005$, $y = t + U(0, .01) - .005$
cross	$N/2$ points at $(U(0,1), .5 + U(-0.005, .005))$; $N/2$ at $(.5 + U(-0.005, .005), U(0,1))$.
norm	both dimensions chosen from $N(0, .01)$.
clus	$N(0, .01)$ at 10 points in the unit square.
arc	in a circular arc of width .01

Table 2.1: Nonuniform distributions.

In order to see how each algorithm adapts to its input, we will study further tests using inputs from very nonuniform distributions. In Table 2.1 the notation $N(s)$ refers to the normal distribution with mean 0 and standard deviation s, and $U(a, b)$ is the uniform distribution over the interval $[a, b]$.

The graphs show each algorithm running on five different inputs of 10K sites from each distribution. The uniform distribution serves as a benchmark. Figure 2.26 shows the effect of these point distributions on the incremental algorithm. As expected, point distributions with heavy clustering, such as corners and normal, stress the point location data structures in each algorithm, increasing point location costs by up to a factor of ten. These represent worst case inputs for these data structures. However, the comparison tests in the point location routine are substantially faster than circle tests, so even in the worst cases here, the whole runtime didn't increase by more than a factor of two. Using sampling to build a more adaptive data structure for nearest-neighbor search [Ben90, Wei78] should reduce these problems substantially. The distribution of the input has little effect on the number of circle tests that each algorithm performs. The performance of the quad-tree algorithm is slightly more erratic than the bucket-based incremental algorithm, but the effect is relatively minor.

Figure 2.27 summarizes the performance of Fortune's algorithm in this experiment. The first graph shows that the bucket-based implementation of the event queue is very sensitive to site distributions that cause the distribution of priorities to become extremely nonuniform. In the cross distribution, this happens near the line $y = 0.5$. At this point, all of the circle events associated with with $n/2$ sites near the line cluster in the few buckets near this position. The corners distribution causes a similar problem, but to a lesser degree. Here, all of the events associated with the $O(\sqrt{n})$ circles in the event queue tend to stay clustered in one bucket at $y = 0$ and another at $y = 1$. In both of these cases, the non-uniform distribution of sites in the x-direction also slows down site searches on the frontier, but this effect is less pronouced than the bad behavior of the event queue.

The second graph shows that the performance of the heap is much less erratic than the buckets. The small jumps that do appear are due to the fact that the event queue does become larger or smaller than its expected size on some distributions. However, since the cost of the heap is logarithmic in the size of the queue, this does not cause a large degradation in performance.

Figure 2.28 shows the performance of Dwyer's algorithm in the experiment. Dwyer's algorithm is slowed down by distributions that cause the algorithm to create many invalid edges in the subproblems, and then delete them later in the merge steps. This effect is particularly pronounced

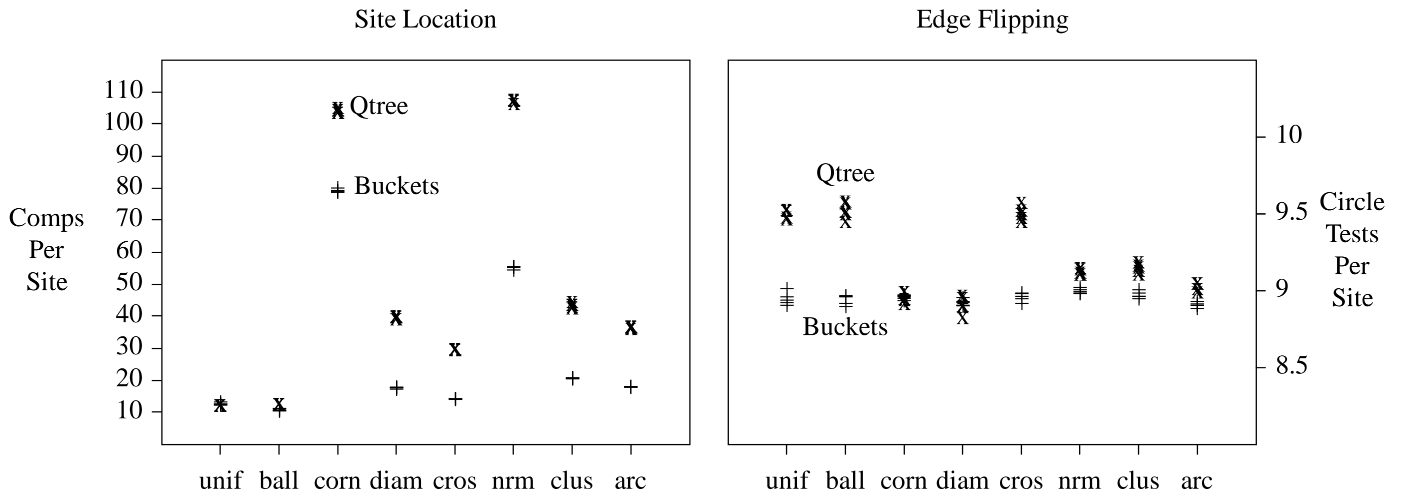


Figure 2.26: The incremental algorithm on non-uniform inputs, n is 10K.

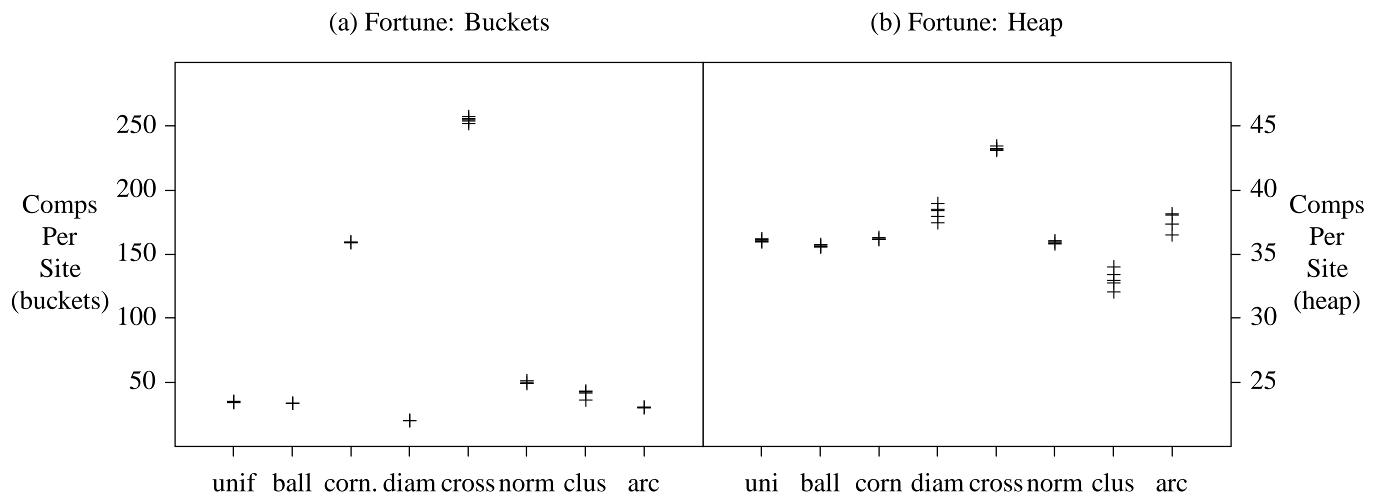


Figure 2.27: Fortune's algorithm on non-uniform inputs, n is 10K.

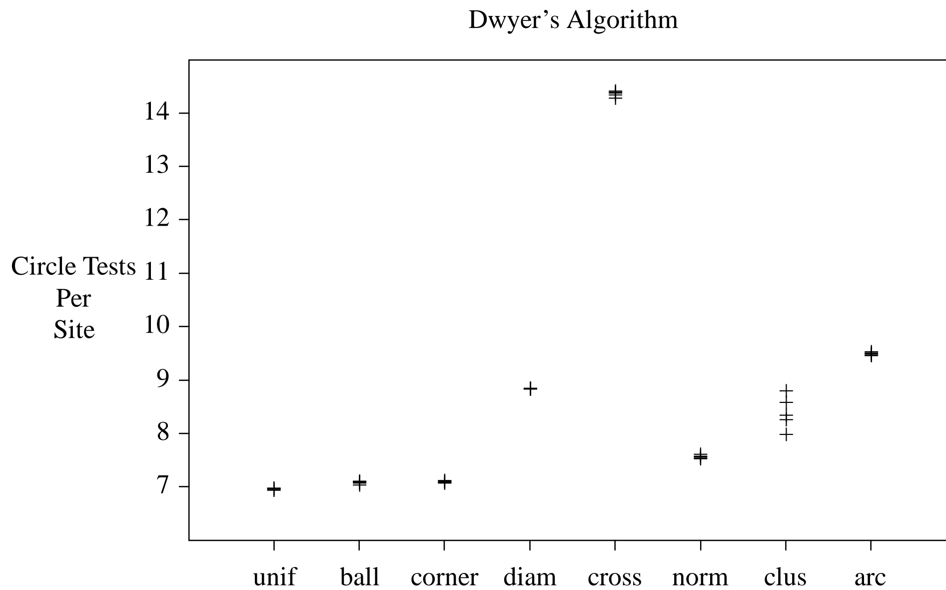


Figure 2.28: Dwyer's algorithm on non-uniform inputs, n is 10K.

run with the cross distribution because the group of sites near the line $x = 0.5$ is very tall and skinny, creating a worst case for the merge routine.

Figure 2.29 shows how the bad inputs affect Algorithm IS. These figures leave out the two worst inputs for this algorithm: corners and normal, because the algorithm would have taken several hours to finish the benchmark. The $O(n^2)$ behavior of the algorithm on these inputs is shown in Figure 2.30.

Algorithm IS is easily the most sensitive to the distribution of its input. This is not surprising, since it depends on essentially the same routine that the incremental algorithm uses for point location, and we have already seen that the point location subroutine performed badly on bad inputs. This did not handicap to the incremental algorithm to a large degree because the point location routine is not the major bottleneck in that algorithm. However, the performance of `Site-search` largely determines the runtime of Algorithm IS.

Finally, to understand how the abstract measures actually effect performance, Figure 2.31 shows the average runtime of the five trials with each algorithm except Algorithm IS. Since none of the runtimes in the graph are much greater than Algorithm IS's performance even in the uniform case, I didn't feel that including it in this graph would add any useful information. The graph reflects the fact that the primitives that each algorithm uses incur a wide range of actual runtime costs. Looping through buckets is faster than restructuring trees, and `CCW` tests are cheaper than `in-circle` tests. Thus, even though the abstract costs of some of these algorithm vary widely over the different inputs, the actual runtimes do not.

2.10 Notes and Discussion

The experiments in this chapter led to several important observations about the performance of serial algorithms for constructing planar Delaunay triangulations. These observations are summarized below:

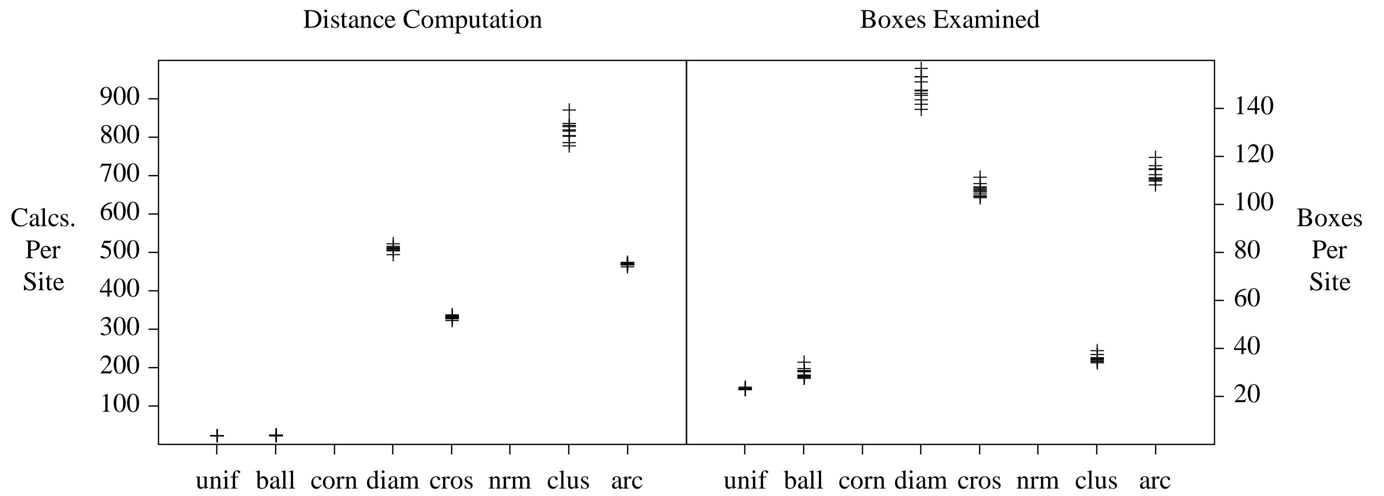


Figure 2.29: Algorithm IS is very sensitive to bad inputs.

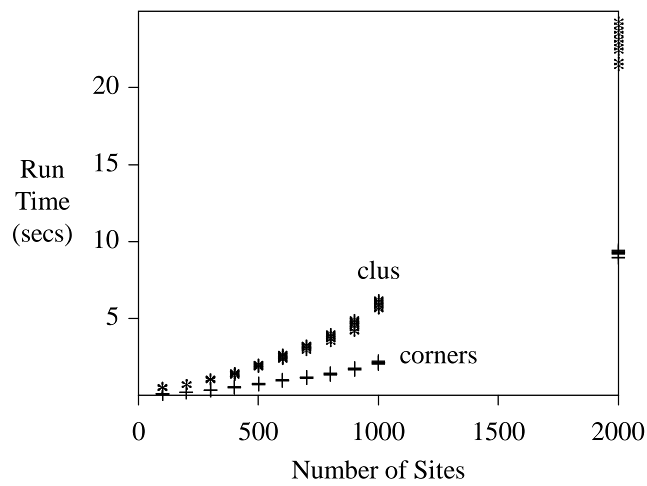


Figure 2.30: Algorithm IS the “cluster” and “corners” distributions.

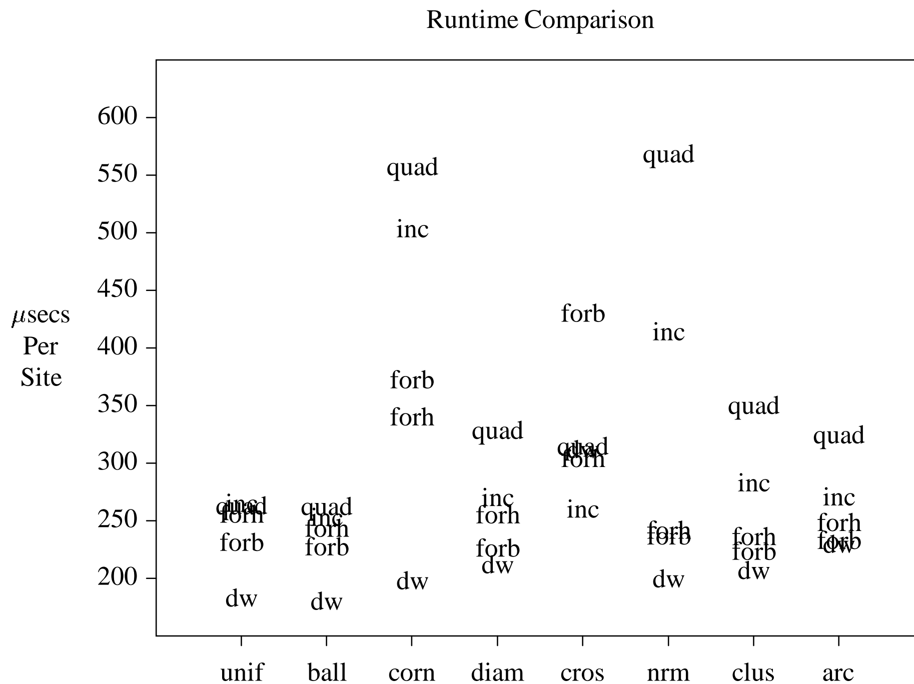


Figure 2.31: Runtimes on non-uniform inputs, n is 10K. inc and quad are the incremental algorithms, dw is Dwyer's algorithm, forb and forh is Fortune's algorithm with buckets and a heap respectively.

- A simple enhancement of the incremental algorithm results in a an on-line algorithm that is competitive with known algorithms for constructing planar Delaunay triangulations. The algorithm has an optimal expected runtime of $O(n)$ time, is simple to implement, and its runtime constants are small. In particular, the algorithm is simpler and faster than previous improvements to the incremental algorithm.

- Although the performance of the new incremental algorithm is dependent on the distribution of the sites, only extremely nonuniform point sets degraded performance to a significant degree.

- Algorithms based on incremental search appear to be somewhat slower than the other methods that we have examined. Improved data structures for site searching may help matters here. Such data structures would definitely help to decrease the sensitivity of this algorithm to non-uniform inputs.

- On sets of n uniformly distributed sites, the frontier in Fortune's algorithm has an expected size of $O(\sqrt{n})$ edges. The expected size of the event queue in the algorithm is also $O(\sqrt{n})$.

- On uniformly distributed sites, circle events in Fortune's algorithm cluster near, and move with, the sweepline. This causes a hashing-based priority queue implementation to perform badly on large inputs. It is possible that a better hash function would alleviate this problem, though it isn't clear exactly how to implement the idea.

- Using a heap to represent the event queue in Fortune's algorithm improves its performance on large problems by a small amount. Most of the cost in maintaining a heap in Fortune's

algorithm is incurred by `extract-min`. Reducing this overhead would make the heap-based algorithm much more efficient than the bucket-based one, but it is not clear how to achieve this.

- Dwyer's algorithm is the strongest overall for this range of problems. It is consistently faster than the incremental (and all other) algorithms, but is also arguably harder to program. In addition, incremental algorithm is more "online", and can be extended to handle dynamic problems efficiently.

2.10.1 Other Algorithms

While the experiments in this chapter cover the most of the available algorithms for this problem, they were not quite comprehensive. I did not consider the divide and conquer algorithm discussed by Guibas and Stolfi because Dwyer's algorithm always at least as fast. I also did not implement the incremental algorithm of Guibas, Knuth and Sharir [GKS92], because it is likely that their point location scheme is slower than mine on the inputs that we considered. Finally, the convex hull algorithm developed by Barber [Bar93] was not available soon enough for me to do a full analysis of it for this chapter. However, the algorithm appears to be practical, efficient and robust and should be the subject of future studies.

2.11 Principles

The design and implementation of the incremental algorithm illustrates several principles in algorithm design that are of general interest.

Bottom Up Search. The point location algorithm speeds up near neighbor searches in the current Delaunay triangulation by augmenting the data structure with an array of buckets.

Dynamic Hashing. The bucket array used for point location dynamically adjusts to the size of the problem. The amortized cost of these adjustments is linear in the size of the final diagram.

Randomization and Probabilistic Analysis. The incremental algorithm takes advantage of recent results in the analysis of randomized incremental algorithms to be more resilient against poor point distributions. The performance of the incremental algorithm on these distributions is further experimental verification of the utility the randomized incremental framework.

Adaptive Data Structures. Randomizing the input order to the incremental algorithm makes the update procedure adapt gracefully to non-uniform inputs. No pathological input can make the insertion procedure perform many more edge edge flips than average. The performance of the incremental algorithm could be improved by also making the point location routine more adaptive to highly clustered point sets. Using recent variants on Bentley's k-d tree data structure [Ben90] might be an effective way to do this.

The experiments and other studies in this chapter also make use of many tools from the experimental analysis of algorithms. Much of this work is heavily influenced by the writings of Bentley [Ben82, Ben89], and the Ph.D. dissertation of McGeoch [McG86]. Later on in this thesis, we will make new use of these ideas in the context of *parallel* algorithms design.

Abstraction. Algorithms should be analyzed in terms of reasonably high level abstract operations as opposed to low level machine operations. Maintaining abstraction makes it easier to obtain results that are independent of a particular machine or implementation and thus provides better insight into the basic behavior of an algorithm.

Explicit cost models. We cannot maintain abstraction at an unreasonable cost. Therefore, we must choose our primitives carefully and we must be reasonably sure that they are efficient on the target machines. The algorithms in this chapter all depended on arithmetic primitives such as `in-circle` which are easily implemented on most serial machines. In the chapters on parallel algorithms, we will see how low level cost models can model the performance of simple algorithms very accurately, and how higher level cost models make the analysis of more complicated algorithms manageable.

Pragmatic Algorithms Design. The worst case asymptotic complexity of an algorithm should not the be only measure of algorithmic efficiency. This is especially true for parallel algorithms, where constant factors play an even larger role. The studies in this chapter have concentrated on analyzing or measuring the expected complexity of the algorithm under various input models (i.e. uniformly distributed sites, random insertion orders). Good primitives and cost models can let the algorithms designer know which parts of an algorithm are the critical bottlenecks and to analyze and possibly optimize those parts appropriately.

Algorithm Animation. Animations of algorithms are a valuable tool for gaining intuition about their behavior. The "movies" in this chapter illustrate the basic ideas behind each algorithm. In addition, the pictures of the frontier in Fortune's algorithm motivated the analysis of the size of the event queue.

Simulation Experiments. This chapter has shown how careful experiments, data analysis, and visualization can lead to important observations about the performance of algorithms. McGeoch outlined many of these techniques in her thesis[McG86]. In this chapter, the simulations took the form of instrumented programs that implemented the algorithms under study. This provided a way to precisely characterize the performance of the algorithms for specific types of inputs and problem sizes. Later on, we will use similar techniques to study implementations of parallel algorithms.

In her thesis, McGeoch points out that the simulation program need not implement the algorithm under study, as long as it can measure the abstract cost of the algorithm. Such "simulation shortcuts" will also be useful in our study of parallel algorithms. In particular, using serial simulations of parallel algorithms allows the algorithm designer to evaluate multiple design choices without the cost of developing full parallel implementations on many target machines.

Chapter 3
Models and Machines

By copying, ancient models
should be perpetuated.
—Hseih Ho

Shining and free; blue-massing
clouds; the keen and
unpassioned beauty of a great
machine;
—Rupert Brooke

This chapter presents the machines and models that we will use to design, implement and evaluate parallel algorithms. Section 3.1 discusses parallel programming models that are currently popular. Section 3.2 discusses the weaknesses of the PRAM model while section 3.3 outlines our programming model. The descriptions of these models will be informal and operational. More formal and abstract discussions of models and semantics for parallel computation are outside the scope of this thesis.

This chapter presents implementations for two rather different machines: The Cray Y-MP, a vector multiprocessor, and the KSR-1, a distributed shared memory multiprocessor. These machines represent the two major design points in high performance computer architecture today. The Cray is a more conventional machine with a relatively small number of large, highly pipelined CPUs connected to a high-bandwidth centralized memory. The KSR-1 has a relatively large number of smaller CPUs and a distributed memory system. Section 3.5 will give an overview of each architecture and its programming systems.

We use multiple architectures to illustrate two conflicting points about programming parallel algorithms. First, we would like to show that good parallel algorithms are, to a certain extent, independent of the machine one which they are implemented. The analysis and implementation of a simple algorithm for nearest neighbor search in the next chapter will illustrate how explicit cost models can accurately predict the performance of algorithms across multiple architectures and programming systems.

However, carefully matching an algorithm to an architecture can make programming and studying the algorithm more straightforward than if machines are chosen blindly. For example, in Chapter 6, the algorithms that we will study for constructing the Delaunay triangulation are complex enough that high level mechanisms for concurrency control and shared data management are of great utility. Using these mechanisms, the implementation of the algorithms is straightforward

enough that we can concentrate more on *algorithmic* questions, rather than arcane programming tricks. Therefore, for these algorithms it makes sense to do our initial studies on the KSR-1, which provides such mechanisms, rather than the Cray, where for a variety of reasons multiprocessor programming is more difficult.

3.1 Popular Programming Models

One of the main difficulties facing the designer of parallel algorithms is the wide variety of programming models that exist for parallel architectures. This is in contrast to conventional algorithms design, where the RAM model is generally used by everyone, and where most people think that the model is a reasonable approximation of reality. The RAM model succeeds because it models those aspects of an algorithm which usually dominate its performance.

Recall that in the RAM model, a processor is connected to an infinite memory. In each time step, the processor fetches data from the memory, executes an instruction, and writes a result back to the memory. This is clearly a simplification of reality, since it ignores register usage, cache misses, virtual memory, multiprogrammed operating systems and a host of other systems issues. Yet, for the most part, efficient RAM algorithms[1] have proven to be efficient on actual machines.

In theoretical computer science, the model of choice for studying parallel algorithms has become the PRAM, which is a natural extension of the RAM model. In the PRAM model, we replicate the single RAM processor P times, and postulate that all P processors share the same infinite memory. In unit time, each processor can execute one step of a RAM program, and in the strongest version of the model, the memory system resolves concurrent reads and writes in some pre-defined way.

Practitioners in parallel computing have long ridiculed the PRAM model as unrealistic and PRAM machines as unbuildable. However, this point of view is somewhat misguided. First of all, building a shared memory abstraction on top of a distributed memory system has been a fertile area for research in architecture, operating systems and language design. Many research and commercial systems currently support a shared memory programming model [AJF91, CF89, EK89, GW88, LLGG90, LLJ+92, KSR91, WL92]

In addition, data parallel [SH86] programming models, which are very similar to PRAM, have recently become popular. These models use a single thread of control combined with parallel primitives that specify operations over large aggregate data structures such as vectors.[2] These primitives fall into three classes: elementwise arithmetic, permutation of vectors (routing) and computing parallel prefix operations (scans). In his thesis [Ble90], Blelloch argues that an abstract model based on these vector primitives results in simpler algorithms which are easier to analyze and more efficient than PRAM algorithms for many kinds of problems. Blelloch and his students have shown that algorithms using these primitives can be efficiently implemented on SIMD and vector architectures [CBZ90]. In addition, Chatterjee has shown that with suitable compilers, programs based on these primitives can be efficiently translated to run on shared memory MIMD machines [Cha91].

Similar work has also been done for languages that don't explicitly use vector models. Projects like Fortran-D [HKT91], Dino [RSW90], Kali [KMR90], Crystal [CCL88], Proteus [MNP+91],

---

[1] That is, RAM algorithms that are both asymptotically efficient and simple enough to have low constant factors. Examples include quicksort, the Delaunay triangulation algorithms in the previous chapter, and so on.

[2] Sometimes people call this a single program multiple data, or SPMD programming model.

and MIMD C* [QHJ88], are all concerned with compiling data parallel programs. The Fortran D document gives a good summary of many research compiler systems. These compilers take shared memory programs, possibly with annotations, and generate code for a variety of machines (SIMD, MIMD, shared memory, distributed memory, and so on). Thus, in both theory and practice, the most popular *programming* model for parallel algorithms is basically PRAM.

3.2 What is wrong with PRAM

If PRAM is so similar to the popular parallel programming models, why are PRAM results and PRAM algorithms generally only of theoretical value? The answer lies in what PRAM algorithms seek to achieve and how PRAM algorithms are analyzed. Designers of PRAM algorithms are often interested in the mathematical question of determining the asymptotic complexity of a given problem, as opposed to finding practical solutions for these problems on existing machines. Since these designers aren't interested in practical solutions to problems, it isn't surprising that their analysis does not address questions that are relevant to the performance of real machines. This lack of interest manifests itself in two ways:

- Too much importance is placed on NC. In particular, algorithms which have "fast" runtimes but use, for example, $O(n^2)$ processors are simply of no use. Many theoretical algorithms attempt to use too much concurrency on the assumption that a real machine would simply run the algorithm more slowly by simulating many threads on each real processor. The problem with this approach is the simulation of multiple threads does not come for free, and these added costs can dominate the runtime of the algorithm.

- Complicated data structures or scheduling techniques are used to reduce the parallel "runtime" of basic algorithms. This results in complicated algorithms for simple things such as scans, histogramming, sorting, load balancing and so on. One canonical example of this is Cole's pipelined, parallel mergesort [ACG89]. While it is a tour-de-force of elegant algorithm design and analysis, and contains many beautiful ideas, the algorithm is so complicated and uses so much global data movement that it simply can't compete against simpler algorithms such as radix sort [Ble89, Nat90].

This problem with asymptotics is not restricted to PRAM analysis. Many theoretical algorithms for mesh machines, hypercubes, other network-based models, and even the RAM model are equally unrealistic. This doesn't mean that the ideas in these algorithms are useless. On the contrary, when examined more pragmatically, papers on theoretical algorithms are often hiding simple, elegant and highly practical ideas beneath all of their asymptotics.

The user of a parallel machine is interested in the practical question of how to solve a problem faster by exploiting the concurrency available in the machine, for problem sizes that are relatively small by asymptotic standards (i.e. $N < 2^{32}$). The situation is much the same in sequential algorithms design, where real world algorithms must deal with machine constraints that don't exist in the RAM model. Careful analysis, implementation, and experimentation are needed to obtain the best performance. Until now, most researchers in theoretical computer science have been content to study more abstract questions, so the PRAM model and many PRAM algorithms have been seen as more and more out of touch with reality.

3.3 Our Model

We will use a data parallel model to design new algorithms for the geometric problems outlined in Chapter 1. The model mixes Blelloch's vector model with the PRAM model, the abstract machine being a PRAM augmented with special vector instructions. The vector instructions specify aggregate operations like scans while conventional PRAM code is used for local computation. The pseudocode is a mix of conventional code, parallel loops and parallel primitives. Parallel loops are marked with `foreach`. When appropriate, the code makes use of some of Blelloch's vector primitives. In particular, `scan-plus` stands for a prefix-sum computation, and `pack` moves the flagged items in a source array into the front of a destination array and returns the number of items in the destination array[Ble90].

Formally, the model will assume that the processors in the machine synchronize after each parallel loop or vector instruction. Informally, we know that while routing and scans may require synchronization, local computation does not. Thus, we will assume that processors are loosely coupled enough to execute independently of one another but tightly coupled enough to synchronize fairly quickly. This assumption matches well with the characteristics of currently available machines.

The complexity of an algorithm in this model is measured using two statistics. First, the step complexity of the algorithm, denoted $S(N)$ is the number of parallel steps that the algorithm needs to solve a problem of size N. The work complexity, $W(N)$ measures the total work performed by an algorithm. We calculate the work complexity by keeping track of the length of vector arguments to each vector operation and the work done by active threads executing in each parallel loop. The sum of these values over the history of a program's execution is $W(n)$.

The goal is to design parallel algorithms with a low step complexity and with a work complexity that is no more than the runtime of a good sequential algorithm. In some cases, we will concentrate on worst-case time, but for the most part we will deal with expected-case runtimes. Since both of the problems that we are interested in can be solved in linear expected-time, we would like to find parallel algorithms that run in $O(n/P)$ time on P processors and do linear expected work. Chapters 4 and 6 will describe such algorithms.

In order to achieve the goal of good performance, the model assigns explicit costs to each class of primitive operations. For example, at the machine level, we might use the following parameters:

$$\begin{aligned} A &= \text{Cost of local arithmetic.} \\ R &= \text{Cost of routing.} \\ S &= \text{Cost of scans} \\ P &= \text{Number of CPUs available} \end{aligned}$$

These constants model the per-element time needed to perform vector operations on a particular machine. We can measure these costs experimentally using any reasonable implementation of the primitives on a given machine. Chapter 4 gives a concrete example of how to define and use these low level models.

For more complicated algorithms, it may be appropriate to use a higher level cost model. For example, the algorithms in the previous chapter were all analyzed with respect to calls to circle-testing and point-location primitives. While these operations can be broken down into sequences of low level calls, using a higher level abstraction makes the algorithm easier to analyze without sacrificing accuracy, provided that we pick our primitives carefully. Chapter 6 uses this style of analysis to design parallel algorithms for constructing the Delaunay triangulation.

Abstract cost models keep the algorithm designer aware of machine dependent issues without forcing her to address them until the last stages of implementation. Then, algorithms can be tuned and evaluated by substituting costs from a real machine into the analysis, and transforming the code as needed to obtain better performance.

Recent work in parallel sorting has used this approach to develop high performance sorting algorithms on several architectures: Blelloch, et al. on the Connection Machine CM-2 [BLM+91], Blelloch and Zagha on the Cray Y-MP [ZB91], and Hightower, Prins and Reif on the MasPar MP-1 [PHR92] all show that algorithms can be described at a high level while maintaining an accurate estimate of their performance on real machines. In each of these papers, algorithms are analyzed in terms of an abstract model that assigns explicit costs to primitive operations. In each paper, straightforward analysis of the algorithms resulted in performance models that accurately predicted the speed of the algorithms over practical problem sizes. The algorithms in this thesis extend the applicability of these methods to parallel geometric algorithms, which have not been heavily studied in a practical setting.

3.4 Other Models

The number of proposed programming models for parallel algorithms is huge. This section will briefly survey some recent attempts to define more realistic models for parallel computation, and why we do not consider them in detail in this dissertation. This survey is not meant to be a comprehensive look at all the available literature. Instead, it is only meant to illustrate the general directions that researchers have taken in tackling this problem.

First, we do not consider network-based models suitable for the practical study of algorithms because they are inherently machine dependent. There is also evidence that tuning algorithms for specific network architectures does not necessarily result in better performance. For example, neither of the two fastest sorting algorithms in Blelloch et al.'s study uses the hypercube network in the Connection Machine CM-2 in a direct way [BLM+91]. In addition, none of the new commercial machines allow the programmer to directly control the routing of messages through the communications network. The structure of the communications network is largely hidden from the programmer.

Extensions of the PRAM model make up most of the rest of the literature in this area. Each of these models extends the PRAM model to deal with a potential bottleneck in the performance of parallel algorithms. Leiserson and Maggs consider limited network bandwidth [LM88]. Aggarwal, Chandra and Snir study communication latency [AGSS89] and locality [ACS90] in PRAM computations. Alpern, Carter and Feig consider hierarchical memory systems in uniprocessors [ACF90], while Heywood does the same for multiprocessors [HR91]. Aggarwal and Vitter [ACG+88], and later Vitter and Shriver and Nodine [VS92, NV91] studied the design of basic algorithms in two-level memory/disk hierarchies. Womble, et al. [WGWR93] discusses the implementation of basic scientific libraries in memory/disk hierarchies and Cormen's thesis [Cor92] presents strategies for implementing virtual memory systems for data parallel computers. Cole and Zajicek [CZ89] consider the cost of synchronization in PRAM algorithms, as does Gibbons [Gib89].

Finally, Valiant [Val90] and Culler, et al. [CKP+92] break away from the PRAM model and propose programming models based on point-to-point communication primitives and their costs. Valiant's BSP model allows processors to execute asynchronously, and models communication latency and limited bandwidth. Culler, et al.'s LogP is less abstract, and attempts to reflect current trends in the technology used to construct parallel machines. The model is asynchronous, and

uses four parameters to model network latency (L), communication overhead (o), communication bandwidth (g), and the size of the machine (P).

All of these models remove one or another of the PRAM model's simplifying assumptions. In theory, this should make the models more reflective of practical situations. However, analyzing algorithms in these models is now much more complicated. This is reflected by the fact that only the most basic primitives, such as summation, permutation, list ranking, FFTs, and matrix arithmetic have been received much attention under these models. [AGSS89, ACF90, CGO90, CKP$^+$92, Gib89, VS92]. In addition, there has been some study of graph algorithms [ACS90, CGO90] and sorting [NV91, VS92].

For our purposes, this extra complexity is the main reason not to use any of these models for our analysis. The algorithms that we will study are complicated, irregular, dynamic, and not yet well understood. In addition, many of these models do not accurately reflect mechanisms that are available to programmers of current parallel machines. The asynchronous PRAM does not make atomic operations such as `fetch&add`, `compare&swap` or queue locks available, even though every major commercial multiprocessor has such a mechanism. LogP models point-to-point communication for remote data access, but does not consider the effects of caching and cache-coherency protocols. The result is that in addition to making analysis difficult, these models are more difficult to program than the machines they are meant to be abstractions for.

The high level, data parallel model that we have chosen to use gives us this freedom to ignore machine details when they are not important while still keeping us conscious of the machine-level costs of the primitives used by an algorithm. If we then find that an algorithm's performance is severely hindered by a particular bottleneck, we can then turn to more detailed models to study that problem in isolation.

3.5 Machine Descriptions

The following sections present short descriptions of the machines used in studying the parallel algorithms in the thesis. After much study, two architectures were chosen, each representing a major design point among current architectures. The Cray represents the more traditional, modestly parallel, heavily pipelined architecture with a large, centralized memory. The KSR-1 represents a new breed of machines with much more available parallelism and a physically distributed memory system.

The motivation for these choices was to illustrate the utility of being able to design high level algorithms in a machine-independent way, and the practical issues involved in translating those high level algorithms to efficient machine-dependent code.

We have purposely avoided machines that force programmers to use a message-passing programming style. This is because all current message-passing systems make it difficult to define and utilize large distributed data structures. No software or hardware mechanisms are available to help the programmer manage data which might be shared by multiple processors. Machines with a centralized memory don't need such mechanisms while cache-coherent machines, such as the KSR-1, provide them in hardware. In addition, since large cache-coherent systems are built on top of message-passing hardware, they can provide a higher level programming model at a reasonable cost. Finally, if the programmer really wishes to construct message-passing programs, it is possible to use shared memory constructs to do so.

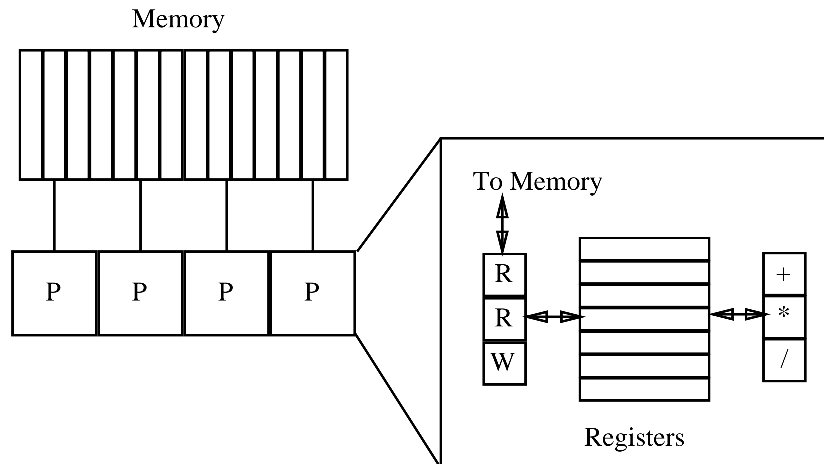


Figure 3.1: Schematic diagram of the Cray Y-MP.

3.5.1 The Cray Y-MP

The Cray Y-MP is a vector multiprocessor. Each "head" of the Y-MP is a large, pipelined CPU. Each CPU can execute two kinds of instructions. Scalar instructions correspond to the instruction set of a conventional architecture while vector instructions take maximum advantage of the CPU's pipelined functional units. There are functional units for floating point operations, logical operations and memory operations. The functional units take input from, and write results to, one of eight vector registers, each of which can hold 64 words of data. After a fixed startup cost, the arithmetic and logical functional units return one result per clock tick (see Figure 3.1).

The performance of the memory unit depends on the access pattern of a particular instruction. The Cray memory system is divided into 256 banks. The banks are grouped in four sections each containing eight subsystems that hold eight banks each. Thus, each section has a total of 64 banks of memory. The memory system is interleaved so that consecutive words of memory live in different banks. As long as an access pattern avoids even strides (and in particular, strides that are multiples of 32), the memory system can deliver one word per clock tick after an initial startup time. In addition, the memory unit supports general routing using scatter/gather instructions. Again, these instructions will be efficient except when the access pattern causes bank conflicts.

Compilers for the Cray accept programs written in C or Fortran and, using compile-time analysis or directives, they find loops whose iterations can be executed independently of one another. Such loops are called vectorizable, and transforming a program into vectorizable form is called vectorizing the program. Vectorizable loops can also be partitioned across multiple CPUs on the Y-MP. Cray provides automatic tools for achieving this. As long as we are careful to write our Cray code in a certain style, the Cray compilers can do a reasonable job of obtaining good machine performance.

We can vectorize each class of parallel instructions in our model. We consider each vector register element to be one "virtual" processor, or thread of control. In effect, virtual processor provide a way to in principle translate any shared memory program into vector form. Of course, such translations may not be efficient. A full Cray Y-MP has eight CPUs with vector registers that are 64 elements long. This gives us 512 virtual processors to work with. The machine I used only has four CPUs, for a total of 256 virtual processors. Local arithmetic instructions will vectorize

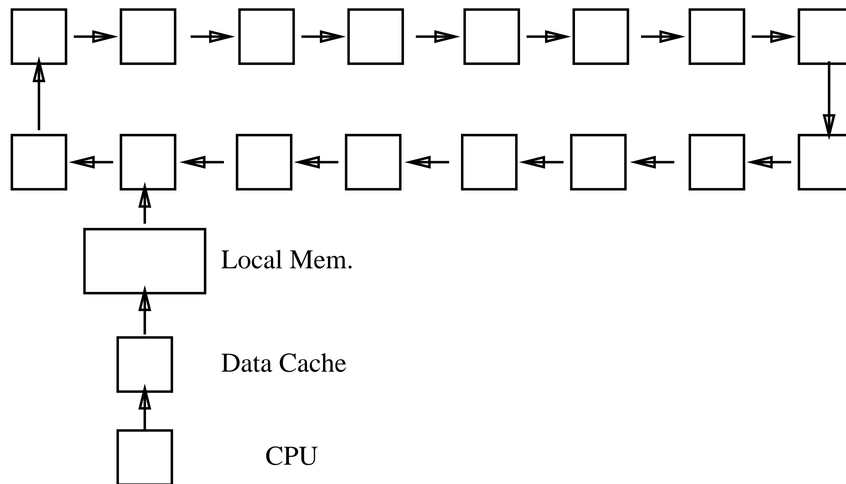


Figure 3.2: Schematic diagram of the KSR-1

trivially. We can handle routing using scatter/gather instructions. Finally, scans vectorize well if we use the clever algorithm of Blelloch, Chatterjee and Zagha [Ble90, ZB91].

3.5.2 The KSR-1

The KSR-1 is a distributed shared memory multiprocessor. While the physical memory of the machine is distributed among the processors, the KSR-1 provides the programmer with a globally shared virtual address space. It uses sophisticated cache coherency hardware to support this programming model. When a thread generates a virtual address for data that does not currently live in the processor on which the thread is executing, the KSR-1 memory system will fetch the data transparently from a remote processor. If an algorithm has enough locality, the cost of these fetches can be amortized over many local operations. If not, the performance of the algorithm will be limited by the speed of the memory system. Thus, it is critical to design algorithms that access memory using localized, low-contention patterns to achieve good performance on this machine (see Figure 3.2).

Each node of the KSR-1 is a fairly conventional CPU along with 32MB of memory. The processor is made of up several processing units, the CEU, which executes memory and control instructions, an IPU for integer instructions, an FPU for floating point instructions, and connections to the network. The KSR-1 network is a hierarchy of ring networks. Each ring has 32 nodes, and the top level ring can have up to 34 sub-rings for a total of 1088 processors. The KSR-1 virtual address space is broken up into 128 byte units units called "sub-pages." We will use the term "block" to mean sub-page. When a process generates a virtual address, the KSR-1 memory system first searches the local CPU's 256K data cache, then the CPU's local memory for the correct block. If the block is not available locally, the cache controller puts the request on the first level ring. If some processor on this ring has the block, it is fetched and brought back to the requesting processor. Otherwise, the next level of the hierarchy is searched. The memory hardware also uses an invalidate protocol to keep cache blocks consistent, and provides atomic operations for locking and unlocking blocks.

Both C and Fortran compilers are available for the KSR-1. Each comes with a runtime library

Vector Operation	T_e (clocks/elem)	$n_{1/2}$ (elements)
Add	1.19	212
Multiply	1.23	189
Divide	3.87	67
Add scan	2.39	910
Random Permute	1.89	115
Reverse Perm	1.63	225
Identity Perm	1.24	146

Table 3.1: Costs for vector operations on one processor of the Cray Y-MP

that supports multiprocessing through multiple threads. These libraries provide for placement and scheduling of threads along with mutual exclusion, condition variables, barriers, and so on. The Fortran system also has facilities for automatically parallelizing loops, but it isn't flexible enough to be useful for our algorithms. All of the implementations use the threads library directly, rather than depending on a parallelizing compiler.

3.6 Benchmarking

This section presents simple benchmark results for the low level vector primitives. These results will be used in the detailed analysis of the next chapter.

3.6.1 The Cray

Each of the low level primitives in the model corresponds to a simple loop in C. To benchmark the primitives on the Cray, we will study the performance of such code after being processed by the Cray vectorizing compiler. With appropriate directives, the compiler can vectorize all the primitives except for scans. For these operations, we will make use of a set of library routines developed at CMU by Blelloch and his group [CBZ90]. Each of the loops was tested on input vectors of between 100 and 100K elements. Timings were taken using the `cpused()` library function, which keeps track of user time in clock ticks. The results reported are for one CPU of a Cray Y-MP. Table 3.1 shows the performance of these primitives. The parameter T_e is the asymptotic cost per vector element, and $n_{1/2}$ is vector length at which half the asymptotic performance is achieved.

The values for T_e and $n_{1/2}$ are derived experimentally from a least squares fit on the benchmark data. T_e is the slope of the line, and $n_{1/2}$ is the absolute value of the x-coordinate of the x-intercept of the line. These are the standard benchmark metrics for vector computers.

The table shows representative values for each of the three classes of instructions in the model. Division is much more expensive than other forms of arithmetic, so we will avoid it in our algorithms. Also, the performance of the permute operation depends on the structure of the permutation. The table shows the cost of the identity, reverse and random permutations. In our algorithms, we will assume that most permutations are random, so the cost of routing will be about 1.8 clocks per element. For the algorithms that we will discuss, this is a valid assumption, but it may not be in general.

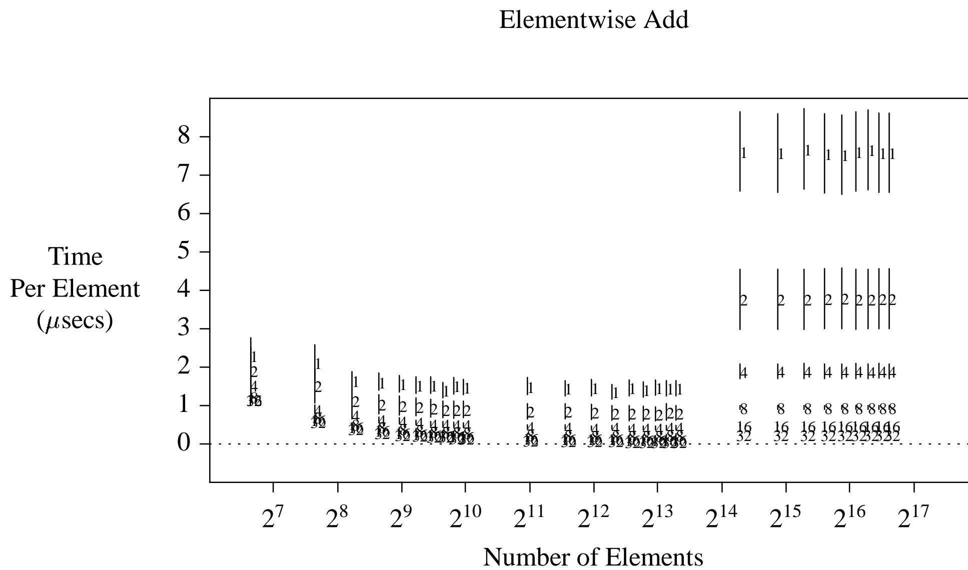


Figure 3.3: Performance of elementwise vector addition on the KSR-1.

3.6.2 The KSR-1

To test the performance of the primitives on the KSR-1, I implemented versions of all of the primitive operations in C using the KSR threads library. Most of the code was ported from the runtime library used in Chatterjee's VCODE compiler for the Encore Multimax [Cha91]. The code was compiled without optimization because optimization introduced bugs into the parallel programs that were difficult to track down. Turning optimization on would probably improve absolute performance somewhat, but all of the relative speedups would remain the same.

Each primitive was tested for vector lengths between 100 and 100K, and with between 1 and 32 threads. Each test was run five times, collecting five sample values from each thread. The results reported are then the mean from this sample set. We take this approach because the KSR performance monitor registers are replicated on each processor, so the machine lacks a central clock. Therefore, it makes sense to collect one sample time per thread. Most of the graphs also show the 95% confidence interval, but in two, these are left out because they made the graphs unreadable.

Figure 3.3 shows the performance of elementwise addition on the KSR-1. The graph shows the mean runtime over five samples for each thread. The number of threads in each experiment is used as the plot symbol, and is shown along with the 95% confidence interval for the sample set. The graph is cleanly split into two regions, depending on whether the vectors fit in the local data cache of each processor. When the local cache overflows, the runtime for each test increases dramatically. Table 3.2 shows the time per element at 10,000 and 100,000 elements for each of the curves in Figure 3.3.

The table shows that the cache overflow effect becomes less pronounced as we increase the number of processors participating in the computation. This makes sense, since more processors have more total cache.

Figure 3.4 shows the performance of `add-scan` on the KSR-1. The plot is split into two separate scales so that the relative performance on large vectors is easier to read. In addition, there are no confidence intervals in the left hand plot because they provided no useful information. The cost of a scan is about twice the cost of elementwise addition, which is consistent with our

Number of Threads	10000	100000
1	1.43	7.57
2	0.79	3.77
4	0.42	1.88
8	0.22	0.94
16	0.11	0.47
32	0.06	0.23

Table 3.2: Cost for elementwise addition on the KSR-1.

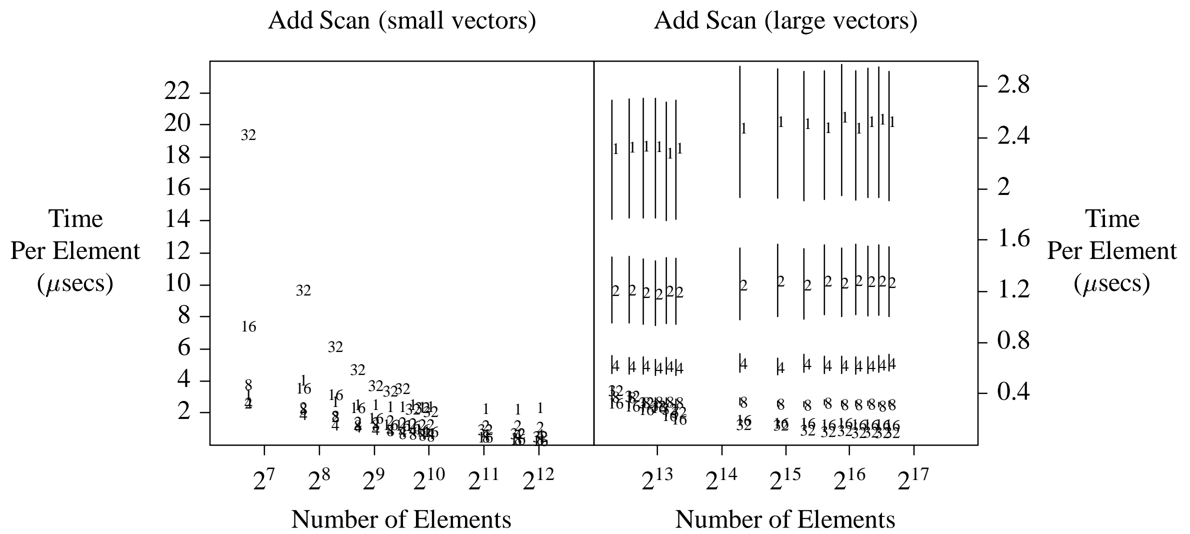


Figure 3.4: Performance of parallel prefix sum on the KSR-1.

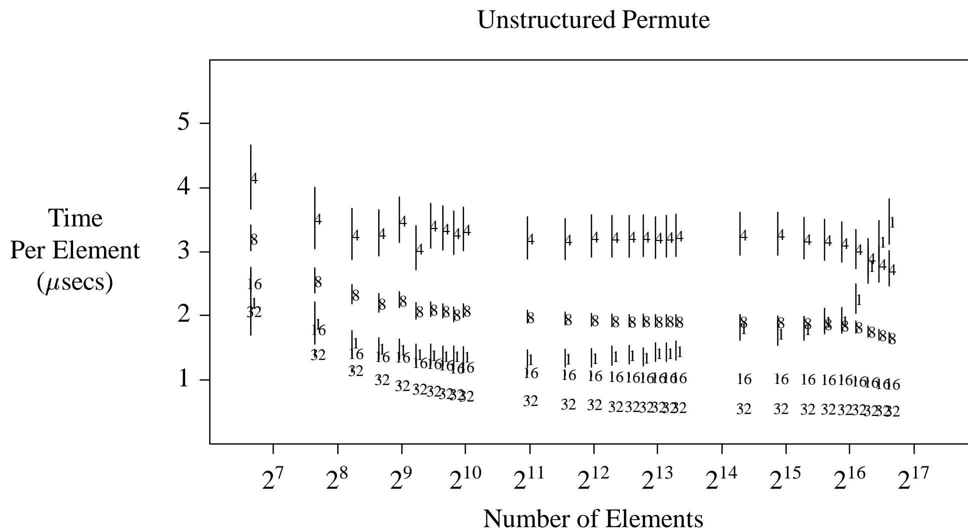


Figure 3.5: Performance of unstructured permute on the KSR-1.

earlier results on the Cray. The scan routine does not show the cache overflow effect because it only involves two vectors, the source and the destination. The KSR-1 data cache is 2-way set associative, and avoids the interference effects that we saw in the simple arithmetic benchmark.

The scan routine is structured as two local vector sums with a global sum in between. In the program, the global sum is performed sequentially by a master thread, which incurs a large amount of overhead with large numbers of threads. This overhead inhibits speedup somewhat with 32 threads, for example. Replacing the sequential routine with a parallel tree-sum would allow the scan algorithm to scale more gracefully [Cha91].

Finally, we will examine the performance of `permute`. As on the Cray, the runtime of this routine depends heavily on the structure of the permutation. We will test the reversal and random permutations. The performance of the KSR on the identity permutation would be nearly the same as in the addition benchmark, since that benchmark is memory bound. The reversal permutation is representative of a relatively structured communication pattern, while the random permutation places the highest stress on the KSR-1 memory system.

Figure 3.5 shows the cost of performing unstructured permutations on the KSR-1. To make the graph more readable, the times for two threads were left off. The lack of locality in the communication pattern inhibits the effectiveness of multiple threads on this benchmark. With small numbers of threads, the overhead of passing cache blocks between processors dominates the runtime of the program, and there is little or no advantage gained. Larger numbers of threads eventually do pay off, but efficiency stays low.

The performance of the KSR-1 on the reversal permutation is much better, because this permutation has much more locality than the random one. Figure 3.6 shows that this operation is only no more expensive than elementwise addition or scans.

Using linear regression, we can estimate the asymptotic values of A, R and S. Using the data from the benchmark experiments, we obtain the following equations with $P = 32$ in units of microseconds:

$$A = 111 + .24N$$
$$R = 355 + .53N$$

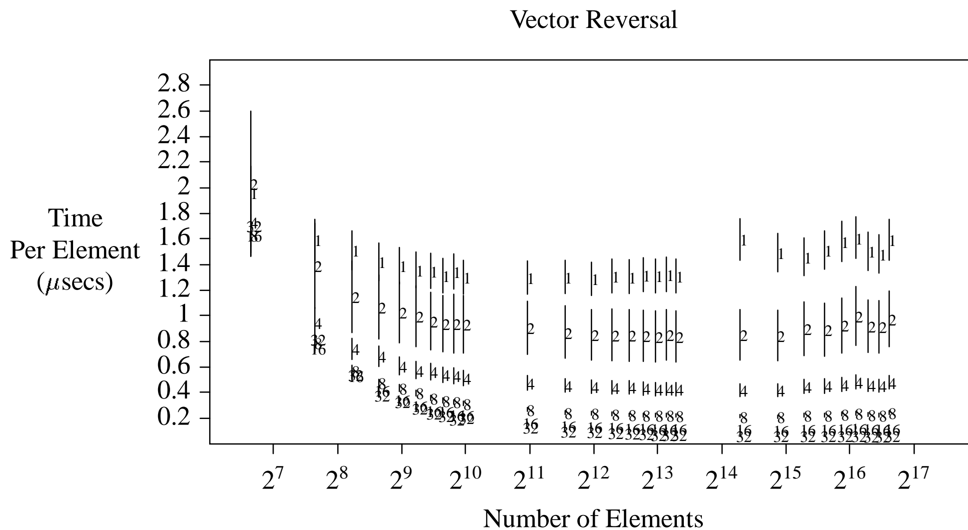


Figure 3.6: Performance of reverse on the KSR-1.

$$S \quad = \quad 1920 + .08N$$

Therefore, we may use $A \approx 8(N/P)\mu$s, $S \approx 2.5(N/P)\mu$s and $R \approx 16(N/P)\mu$s. These values should be used with caution, as they are only valid for this specific configuration of machine and relatively large problems. This will be useful for our analysis in the next chapter, but these are not useful as general parameters for accurately modelling all programs. In particular, on small problems the startup costs and system overhead that each of these routines incurs will distort timing results and produce inaccurate answers.

The performance of the KSR-1 on these benchmarks shows that the primitives in our model can be efficiently implemented on architectures that are very different from traditional vector or SIMD machines. However, the performance of the primitives is less predictable than on the Cray. Furthermore, since it is not natural to program the KSR-1 in terms of large memory-to-memory vector operations, we will see that relating the performance of these primitives to real programs on the machine requires more care than on the Cray, where the vector style of programming fits the machine very naturally.

3.7 Summary

This chapter presented a programming and complexity model for parallel algorithms. The model is based on a set of primitive operations that act on large, aggregate data structures such as vectors. In addition, we discussed alternatives to this model and presented our rationale for choosing data parallel models over the others that are available. Finally, the chapter closed by presenting benchmarks results that show how the primitives in the model can be implemented on our target machines. The next chapter will demonstrate that we can use the benchmark results like these to analyze implementations of relatively simple algorithms very accurately.

Chapter 4
Concurrent Local Search

toy problem: *[AI] n. A deliberately oversimplified case of a challenging problem used to investigate, prototype or test algorithms for a real problem.*
—The Jargon File

In Chapter 2, we saw how a simple bucket-based data structure could be used to speed up sequential algorithms for constructing the Delaunay triangulation. Buckets capture the locality inherent in the Delaunay triangulation construction problem, and provide a simple way to exploit that locality to obtain good performance. This chapter will show how to exploit locality in parallel algorithms by using many concurrent threads all searching a local area of a shared data structure. We will illustrate this technique, which we will call *concurrent local search*, by parallelizing the spiral search algorithm from Chapter 2 to solve the all-nearest-neighbors problem. The sections below will describe the algorithm and show how it is implemented on each of the two target machines. The analysis and implementation of this algorithm illustrate the usefulness of parameterized cost models in developing simple algorithms on multiple architectures.

4.1 The Problem

Recall that we are given a set S of n points. For each $s \in S$, we wish to find the point p in $S - \{s\}$ which is closest to s under the Euclidean metric. This corresponds to the original definition of the problem, but with $Q = S$. As before, the algorithms assume that the input is a set of points uniformly distributed in the plane. However, we will discuss how to adapt the algorithms to handle non-uniform inputs.

4.2 The Algorithm

The algorithm is a parallelization of Bentley, Wiede and Yao's spiral search algorithm [BWY80]. This algorithm starts by bucketing the points in S into a uniform grid of $\sqrt{n/c}$ by $\sqrt{n/c}$ cells. We call the parameter c the *cell density*. After the bucketing phase, the expected number of points that fall into a cell will be c. One can use different cell densities to tune the performance of the algorithm in a particular environment, as we did in Chapter 2. To simplify the analysis in this chapter, we assume that $c = 1$. To find the nearest neighbor of a query point q, the algorithm finds the bucket that q lies in and searches in an outward spiral for a non-empty bucket B. Then it searches all buckets that intersect a circle whose radius is the distance between q and some point in $p \in B$. Each query point is processed in this way until all the queries have been answered.

Bentley, Wiede and Yao show that on inputs from a large class of distributions, this algorithm only searches a constant number of buckets per point, on average. Thus, the algorithm solves the all-nearest-neighbors problem in $O(n)$ expected time.

To solve this problem in parallel, we will parallelize the spiral search. That is, build the bucket structure in parallel, and execute the search steps in parallel. The locality inherent in the spiral search allows the algorithm to exploit a high level of concurrency. Since each search task examines only a small part of the shared data structure, many tasks can work in parallel without fear of conflict.

The next section will describe a parallel/vector version of this algorithm. In addition, it will provide a running analysis of an implementation on the Cray Y-MP. Afterwards, we will see how the same same algorithm, with minor modifications, also exhibits good performance on the KSR-1. This will illustrate that although the original algorithm was designed with a shared-memory vector machine in mind, it doesn't depend on any of these features for good performance.

4.3 All Nearest Neighbors on the Cray

For the Cray algorithm, the runtime will be expressed in terms of clock ticks per vector element per CPU. For large vectors, we have that $A \approx 1.2$, $R \approx 1.8$ (for random permutations) and $S \approx 2.4$. Since the Cray is a vector machine, we add one more model parameter, $L = 64$ to reflect the number of elements in each vector register. This is a measure of the amount of parallelism that each CPU can emulate. At this time, the algorithm does not take advantage of multiple CPUs, so $P = 1$. However, since the algorithm is almost fully vectorizable, it is likely that it could take advantage of multiple CPUs using the automatic parallelization facilities in the Cray compilers.

The first step is to vectorize the bucketing phase of the algorithm. This isn't too hard. We first compute a histogram of the keys so we know how many list nodes to allocate, then we ship the points to the correct places in the bucket structure (see Figure 4.1).

In the parallel procedure, the array `last` keeps track of how many sites have fallen into each bucket. The scan operation computes the first element of `nodes` that falls into each bucket. The second loop then fills `nodes` with pointers to the sites that belong to each bucket. At the end of the second loop, `buckets` will contain the index of the first site in each bucket and `last` will contain the index of the last site in each bucket. So, the sites contained in some bucket b are stored in $nodes[buckets[b]]$ to $nodes[last[b-1]]$ (see Figure 4.2).

The main problem with implementing the code in Figure 4.1 is dealing with write collisions on elements of the `last` array. We can detect and such collisions in the following manner. Initially, each virtual processor is holding a value to write into the array `dest`.

First, each virtual processor first writes its processor number into an array called `test`. (Figure 4.3, part A). Every VP then examines `test` and compares the value there to its ID. Every virtual processor that reads its own ID back from `test` can safely write into `dest` (Figure 4.3, part B). The routine then processes the remaining VPs sequentially (Figure 4.3, part C).

The analysis that follows will show that for uniformly distributed sites, the average number of collisions will be small, so the serial part of the bucketing loop will not become a bottleneck. First, we can see that the cost of the code in Figure 4.1 is:

$$(S + 7A + 5R + (A + 2S + R))n + 2T_C.$$

There are four arithmetic operations to compute the array indices, two to incremental the histogram lasts and one more for the write of `i` into each table entry. The routing operations come from the two

```
// sites[0..n-1] = array of sites
// nodes[0..n-1] = array of list nodes
// bindex[0..n-1] = bucket index
// buckets[0..n/c-1] = array of buckets
// last[0..n/c-1] = histogram
foreach i = 0 to n-1
begin
  bindex[i] := index of sites[i];
  last[bindex[i]]++
end
buckets := last := scan-plus(last)
// The following loop assumes that write
// collisions on the last array are taken
// care of so all processors get a correct index
// into nodes.
foreach i = 0 to n-1
begin
  nodes[last[bindex[i]]++] := i
end
```

Figure 4.1: A parallel loop to bucket points. In the first loop, each point is hashed according to its coordinates and the size of each bucket is computed. The scan-plus operation computes the starting position of each bucket in the `nodes` array. Then, the points are routed to their destinations in the second loop.

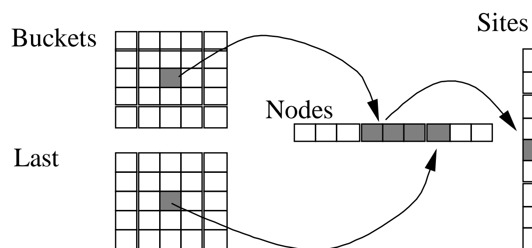


Figure 4.2: The bucket data structure.

fetches from `bindex`, two fetches from `last`, and one fetch from `nodes`. The extra $(2A + S + R)$ counts the cycles needed to deal with possible collisions. This term accounts for the cost of the `pack` and `append` operations. The `append` operation can be implemented as a scan plus a block copy into a statically allocated array holding `write-queue`. This accounts for $A + S$ cycles. The other $S + R$ accounts for the `pack`. Here we conservatively assume that we need to do one `pack` and one `append` per iteration. Finally, T_C is the average cost of the sequential code at the end of the routine. For each collision we must do some arithmetic and one memory write which will take roughly 30 cycles on the Cray, so the expression above becomes:

$$28n + 60C \text{ cycles},$$

where C is the total number of collisions. As long as C is small compared to n, this routine will run in 28 cycles per site, on average.

Using the "birthday paradox," we can see that the expected number of collisions in each group of L sites will be $L(L-1)/2n$ [CLR90]. On average, the total number of collisions will be

$$C = \frac{L(L-1)}{2n} \frac{n}{L} = (L-1)/2.$$

We conclude that as long as L is small compared to n, the bucketing routine will perform well when the sites are uniformly distributed.

Once we have bucketed the points, we can use Bentley, Weide and Yao's spiral search technique to do nearest neighbor searching (see Figure 4.4). For each query point, q, the spiral search is divided into two phases. In the first phase, we find the bucket that the point lies in and search in an outward spiral until we find a non-empty bucket. We then calculate the distance D between the query point and some point in this bucket. In the second phase, we compute the nearest neighbor of the query point by examining every point that lies within a bucket intersecting a circle of radius D centered at q. For simplicity, the algorithm checks any bucket that intersects a square of side $2D$ centered at q. Thus, the algorithm searches D layers of buckets centered at the bucket that q falls in. If q lies in bucket (i, j), then the $k^t h$ *layer* of buckets away from q is defined as all buckets (l, m) such that either $l = i \pm k$ and $j - k \leq m \leq j + k$, or $m = j \pm k$ and $i - k \leq l \leq i + k$.

The main obstacle to vectorizing this algorithm is that some searches may take much longer than others, so we will have to be careful about load balancing to keep all the virtual processors busy. A straightforward way to do this is to create a virtual task for each query point. This task isn't really a task in the operating systems sense since it only keeps the state necessary to perform spiral searching. To do the search, the algorithm has each task do one search step, and then check to see if it is finished. It then uses a call to `pack` to delete any finished tasks, and iterates this process until all the tasks have finished. This scheme uses a small amount of overhead to ensure that no search task does any extra work. The parallel algorithm will do no more work than the original sequential algorithm, except for the overhead due to `pack`. In our algorithms, the expected number of iterations through this loop is constant, so the total overhead due to `pack` is also $O(N)$ work. Therefore the total expected work done by the parallel algorithm algorithm will be $O(N)$.

The load balancing scheme is complicated by the fact that the spiral search loop has a triply nested structure. The outermost loop iterates over the layers of buckets that the spiral search is examining. The second level loop iterates over buckets that lie in each layer, and the inner-most loop iterates over points that lie within each bucket. It is the inner-most loop that does the actual distance computations.

```
// dest[bindex[i]] := dest[bindex[i]]+1
// L virtual processors numbered from 0 to L-1
foreach i=0 to L-1 pnum[i] := i

offset := 0;
write-queue := [];
while (offset < n) do begin
  // Part A
  // each virtual processor writes an ID and reads it
  // back to check for collisions
  foreach i = 0 to L-1 begin
    tindex[i] := bindex[i+offset]
    dest[tindex[i]] := pnum[i]
    test[i] := (dest[tindex[i]] = pnum[i])
  end

  // Part B
  // those who detect no collisions go ahead
  foreach i = 0 to L-1 begin
    if (test[i])
      dest[tindex[i]] := dest[tindex[i]]+1
  end

  // Part C
  // Put other indices onto a queue for
  // sequential processing.
  if some element of test is zero begin
    num-zero := pack(tindex,tindex,not test);
    append tindex to write-queue;
   end
  // Now do the next batch of L writes.
  offset := offset + L
end

for i := 1 to length(write-queue) begin
  dest[write-queue[i]] := dest[write-queue[i]]+1
end
```

Figure 4.3: A three-phase loop to handle implement a `permute` operation that allows collisions in the destination vector. In the first loop, each processor writes a unique ID into the destination and then looks to see if the write was successful. If it finds that no other processor has written into the same location, writes its result value into that location. In this case, it increments the destination. In the third loop, processors that did not find their ID in the destination must be handled sequentially, since they have collided with some other processor.

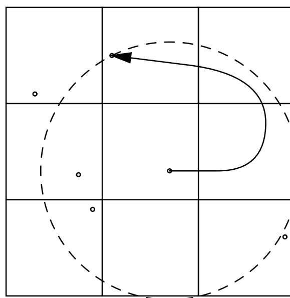


Figure 4.4: Spiral search to find a near neighbor.

```
layer := 0
numtasks := N
while (numtasks > 0)
  foreach i = 0 to numtasks-1 begin
    tasknum[i] := i
    <Prepare for this layer>
  end
  doOneLayer(layer, numtasks, tasknum)
  foreach i = 0 to numtasks-1 begin
    done[i] := <Task is finished>
  end
  numtasks := pack(tasknum, tasknum, not done)
end
```

Figure 4.5: The outer loop of the spiral search algorithm.

The outer loop of each search walks over layers of buckets. After each layer, the algorithm must delete tasks which have finished. It does this with a call to `pack` at the end of each iteration (see Figure 4.3).

Each time the algorithm moves to a new layer, each task checks to see if it is done and if not, it pre-calculates some information for the bucket loop. The cost of these calculations is about $13A$.

In general, a task survives to layer k only if the first point found by the initial stage of the spiral search algorithm was more than $k-1$ layers away from the query point. This means that $k-1$ layers of buckets must be empty. Since the probability that a given bucket is empty is $(1 - c/n)^n < e^{-c}$, the probability that $k - 1$ layers of buckets are empty is $O(e^{-k^2})$ [BWY80]. Thus, the algorithm deletes at least one half of the current tasks each time it moves to a new layer, so the total expected time needed for this bookkeeping is $2n(13A + 2S + 2R)$ or about 48 cycles per point.

The next loop level walks over the buckets within a layer (see Figure 4.6). Each time the algorithm moves to a new bucket, each virtual processor updates its bucket index, does a boundary check, and then participates in a call to `pack`. Any task which has walked through all of its buckets will be deleted at this point. This takes $3(A + R) + 2S$ time. When $c = 1$, the algorithm does one distance calculation per bucket, so the expected loop overhead for walking over the buckets is $nD(3A + 3R + 2S)$ time, where D is the average number of distance calculations that each point performs. We will calculate a value for D below.

The inner loop of the algorithm does a distance calculation between the query point and each

```
doOneLayer(layer, numtasks, tasknum)
  for each bucket in this layer
    foreach i = 0 to numtasks-1
      temp[i] := tasknum[i]
      <do boundary check on next bucket>
      done := <boundary check failed>
      bindex[i] := bucket(tasknum[i])
      done[i] := done or <bucket bindex[i] empty>
    end
    numtasks := pack(bindex, bindex, not done)
    pack(temp, temp, not done)
    doOneBucket(numtasks, temp, bindex)
  end
end
```

Figure 4.6: The second outer loop of the spiral search algorithm.

```
doOneBucket(numtasks, tasknum, bindex)
  bi := 0
  while (numtasks > 0)
    foreach i = 0 to numtasks-1
    Check nodes[buckets[bindex[i]]]+bi]
    done[i] := (bi > last[bindex[i]]
    pack(bindex, bindex, not done)
    numtasks := pack(tasknum, tasknum, not done)
    bi++
  end
end
```

Figure 4.7: The inner loop of the spiral search algorithm.

member of each bucket (Figure 4.7). Again, after each distance computation, the algorithm looks for tasks which have finished with their bucket, and packs them away.

Bentley, Weide and Yao show that we can express D as a sum:

$$D = \sum_i e^{-ci} O(i),$$

where i is the number of cells searched in the first stage. We would like a more exact estimate of this number. Since the coefficients are exponentially decreasing, it suffices to consider just the first few terms of this sum to work out an accurate approximation. If the first cell is not empty, then the algorithm searches the first cell and the eight cells around it. If the first cell is empty, but one of the next eight is not, then we must search the 25 cells in the first two layers around the point. The expected number of distance computations will then be

$$D = 9c(1 - e^{-c}) + \sum_{1 \le k \le 7} 25ce^{-ck} + E$$

where

$$E = \sum_{i \geq 8} e^{-ci} O(i).$$

In order to show that E is negligible, consider the integral

$$\int_a^b x\, e^{-x} dx = e^{-x} - x\, e^{-} x \Big|_a^b (*).$$

If we set $a = 8$, and consider (*) as a function of b, we have that

$$f(b) = 9e^{-8} - (1+b)\, e^{-b},$$

which converges to $9e^{-8} \approx .003$ as $b \to \infty$. We can therefore conclude that E will be small compared to the first few terms in the sum describing D. For $c = 1$, this sum tells us that the algorithm performs approximately 20 distance calculations per site. For each distance calculation we must increment and check an index, load the appropriate site, perform an inner product calculation and update the running minimum. This takes

$$(2A + 2R) + 4R + 3A + (2A + 3R) = 8A + 9R.$$

When we add in the overhead for load balancing, the total expected time works out to be:

$$20(2S + 11R + 8A)n.$$

On the Cray, this will be about 684 cycles per site, or 4.1μs per site. Using this value of D, we can conclude that the outer loop of the algorithm uses about 252 cycles per point.

Putting everything together, we have that the whole algorithm will use about 985 cycles, or about 6μs per point, on average, to solve the all-nearest-neighbor problem.

4.4 Measurements

I have implemented the above algorithm in C, mixed with calls to Blelloch, Chatterjee, and Zagha's [CBZ90] assembly language routines for `pack` and `scan-plus`. The code also runs on my Sun workstation, which proved to be an invaluable aid for modifying and debugging off-line from the Cray.

In addition, I have implemented a C version of Bentley, Wiede and Yao's original algorithm. By comparing the vector algorithm to the sequential, I obtain a more realistic idea of how efficient the vector algorithm is.

Since the algorithm spends most of its time doing distance computations, the first logical parameter to check is the number of distance calculations that each algorithm performs. Figure 4.8 shows the number of distance computations per point that each of the implementations performed as a function of the size of the problem. The graph shows a summary of several runs at each inputs size. The dots in the graph represent the median values of the trials, while the vertical lines connect the first and third quartile values to the minimum and maximum values respectively.

Figure 4.8 shows that the average number of distance calculations performed per point is bounded by a constant near 15. This is close to the value predicted by the analysis. The sequential algorithm performed fewer calculations because it never computes the distance between a point and itself, while the vectorized algorithm avoids the extra conditional check at the expense of a few

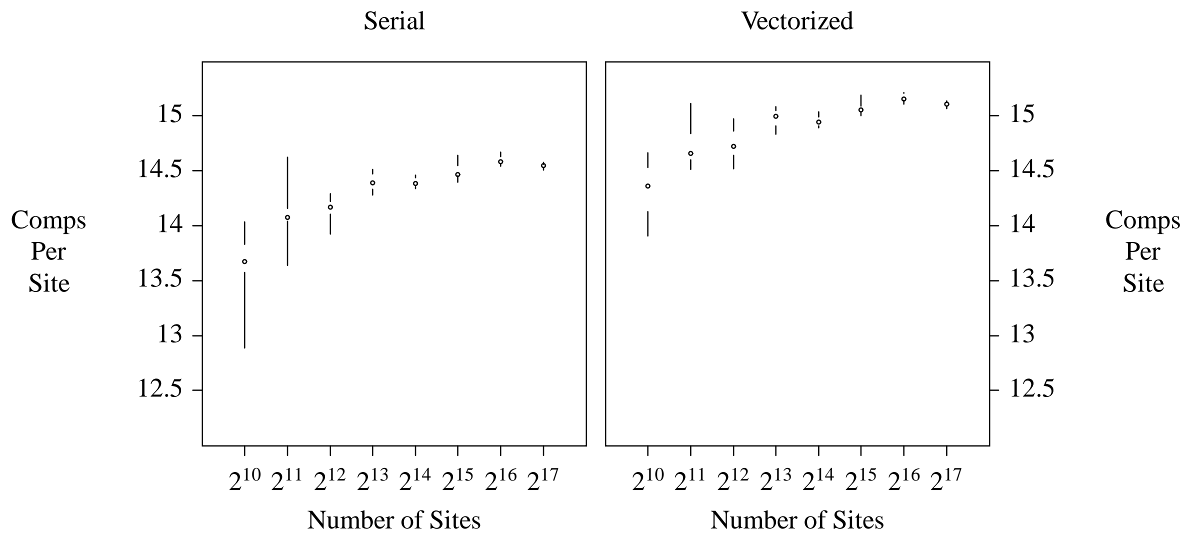


Figure 4.8: Distance computations per site for each algorithm.

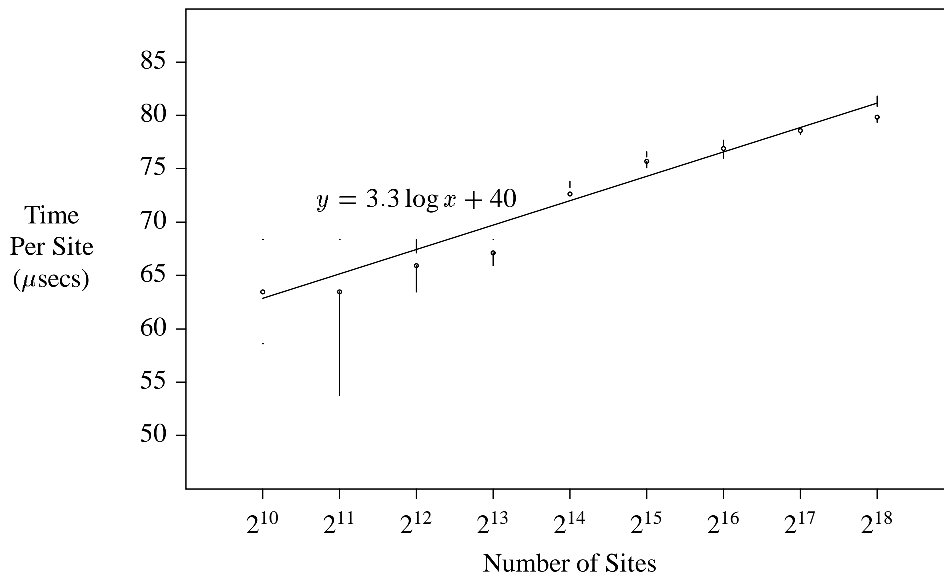


Figure 4.9: Time per point for the spiral search algorithm on a Sparcstation 2 (microseconds).

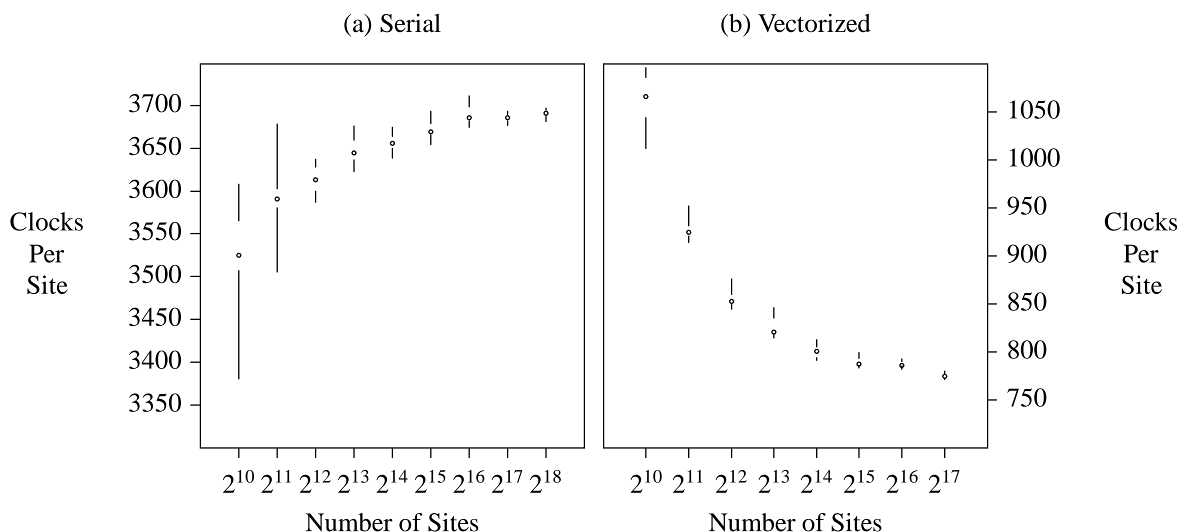


Figure 4.10: Time per point for the spiral search algorithm on the Cray (6ns clocks).

extra computations. On the Cray, this is a good tradeoff since conditionals are expensive inside vectorized loops.

Figure 4.9 shows the runtime of the scalar algorithm on a fast workstation. The timings were taken on a Sparcstation 2 with 64MB of memory, using "`gcc -O2`". The graph shows that the run time of this algorithm tops out at about 80μs per point for the problem sizes that I tested. Obtaining a good asymptotic performance model for this code proved to be difficult due to cache effects. Note that the runs on smaller problem sizes display a linear run time until the problem is too big to make effective use of the workstation's cache. In fact, for large problem sizes, a regression model of the runtime data indicates that the expected cost of the code is roughly proportional to $3.3 \log n + 40$, which is about 80μs when n is 2^{18}. Figure 4.9 shows that this fit is accurate for large n, but less so at small values of n.

Figure 4.10 shows the performance of both the scalar and vector algorithms on the Cray Y-MP. The code was compiled using the Cray standard C compiler using the highest available scalar and vector optimization levels, but no multitasking. The Cray run times are given in units of 6ns clock cycles. Thus, Figure 4.10a shows that the scalar code on the Cray uses about 3690 cycles, or 22.1μs per point.

Figure 4.10b shows the performance of the vectorized algorithm on one processor of the Cray Y-MP. Least squares regression indicates that the asymptotic run time of the algorithm is about 773 cycles per point, or 4.6μs per point. This is about 4.77 times faster than the scalar time on the Cray, 17.24 times faster than the run time on the workstation, and 600 times faster than the time originally reported by Bentley, Weide and Yao for this algorithm when implemented on a PDP-10 [BWY80].

4.5 Extensions and Applications

Although the current algorithm performs well on uniformly distributed point sets, its performance on non-uniform sets is suspect. Bentley, Weide and Yao show that their sequential implementation still performs well even when the points are non-uniform. To check their results, and examine the effect of non-uniformity on the vectorized algorithm, I ran another set of trials with clustered inputs. The clusters were generated using the formulas $x = p_x^i + o_x$ and $y = p_y^i + o_y$ where o_x and o_y are

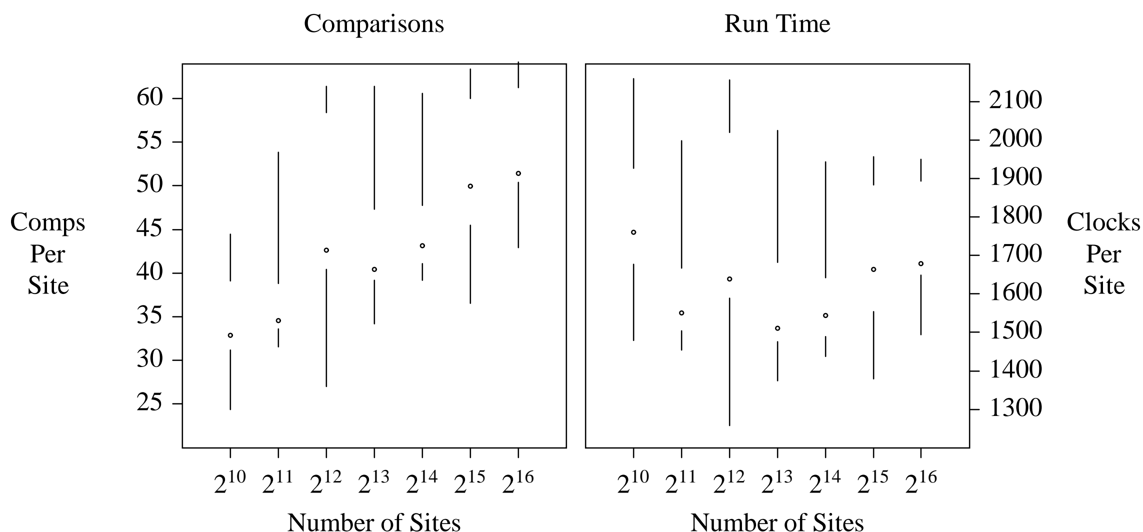


Figure 4.11: Performance of the vectorized algorithm on clustered inputs.

normally distributed and p_x^i and p_y^i are chosen at random for $i = 1, 2, \cdots 10$. I set the scale on the normal to .1 to model mild clustering.

Figure 4.11 shows the results of these experiments. These Figures indicate that mild clustering slows the algorithm down by a factor of two, which is consistent with the earlier results reported by Bentley [BWY80]. More severe clustering would degrade performance even more, in a way that would be similar to the behavior of the point location algorithm in Chapter 2.

For point sets that are extremely non-uniform, we can add a pre-processing phase to the algorithm that attempts to adapt the bucket grid to the distribution of the input. This phase would use a small sample of the input to define a non-uniform bucket grid, and then would further subdivide these buckets in a uniform way. If the sample is a good predictor for the real distribution of the points, then the distribution of points within the new bucket structure will be smooth. Weide shows how to use this technique in the context of conventional algorithms [BWY80] and many parallel algorithms have used random sampling in a similar way [BLM+91, GG91, RS92].

4.6 Discussion

The analysis of this algorithm is an accurate upper bound on the real performance of the program. The actual run time of the algorithm is about 20% better than the analysis predicted. We can trace this discrepancy to two incorrect assumptions in the analysis. First, we assumed that a call to `pack` would cost one scan and one routing operation, or about 4 cycles per element. In fact, the assembly language routine for `pack` overlaps the scan with the routing operation and only costs about 2.5 cycles per element. Second, my analysis predicts that the number of iterations through the inner loop per point is about 20. The actual constant is only about 17. If we re-calculate the cost of the algorithm with these new values, we get about 741 cycles per point, which is a little too fast. The Cray C compiler didn't really vectorize all of the bucketing code. It could not fully vectorize the last loop, even though it is clear the loop can be vectorized using the collision resolution technique shown in Figure 4.3. This makes the bucketing routine run about a factor of two slower than the analysis predicted, but it is a small effect since this routine is not a large percentage of the total run time.

These discrepancies illustrate the inherent difficulty of exactly modeling the behavior of the complete Cray system from the compiler through the hardware. The result is that parts of the resulting program are somewhat faster than my analysis predicted, while other parts are somewhat slower.

The performance of the Cray implementation shows that the analysis was accurate. This is satisfying because the only constants in the analysis are dependent on the Cray architecture, so by reassigning the costs of the primitives, we can easily redesign the algorithm for other machines. Section 4.7 will discuss this in more detail.

The critical assumption that is necessary for the algorithm to perform well is that the size of the input is much larger than the amount of parallelism available in a given machine. This is critical to keeping all processors busy, and for guaranteeing good load balance. Because of this, the algorithm needs large per-processor memories to hold problems big enough to be effective. This is not really much of a problem, since the current generation of parallel architectures all have per-processor memories that are large enough to handle big problems.

Finally, the performance of the algorithm compares well with the performance of the linear time sequential algorithm, but it could be better. The analysis, and experimental data provided by the Cray performance measurement tools show that the inner loop is memory bound, doing nine indirect memory operations in order to perform eight arithmetic operations. Thus, the algorithm uses too much memory bandwidth for each unit of "useful" work. The key to speeding up the algorithm even more is to, if possible, simplify the data structure so that it uses less indirection, or to simplify the structure of the parallel loops to remove as much overhead as possible.

4.7 Implementation for the KSR-1

The KSR implementation of this algorithm follows the same basic outline as the Cray, except that it uses concurrent threads to perform the parallel work rather than vector pipelines. For example, the bucketing stage of the algorithm constructs linked lists of points for each bucket directly, using a simple locking protocol, rather than the histogram and scan routine we used on the Cray (see Figure 4.12). Each thread is given a local pool of nodes to allocate list elements out of. The private counter `next_bucket` keeps track of the next free node. The algorithm never frees nodes, so a more sophisticated memory management routine is not needed here. Using private counters also avoids the large amount of contention that a shared counter would incur.

The code in Figure 4.12 is in a somewhat different style than traditional data parallel code. In particular, it uses an asynchronous programming style that is not generally associated with data parallelism. All of our implementations on the KSR-1 will have this characteristic, but as we will see, it is perfectly possible to analyze the algorithms in the data parallel framework that we defined in Chapter 3. A need for reasonable performance drives our choice of this programming style. Since global synchronization is an expensive operation for MIMD multiprocessor to perform, it would not be reasonable to execute a more traditional, synchronous data parallel program directly on the KSR-1. First, the program must be transformed to into larger, more asynchronous pieces. In the future, this transformation may be done automatically by a compiler (see, for example, Chatterjee's thesis [Cha91]), but at this time, no such compiler exists in a production environment. Thus, our KSR programs may be thought of as what a sophisticated compiler might produce when asked to translate data parallel programs into efficient code for a MIMD multiprocessor.

We will model the performance of this code using the benchmark numbers for local arithmetic and permutation routing from Chapter 3. To obtain a lock, a thread must fetch the cache block

```
% next_bucket is a shared global counter
private int next_bucket;

foreach point P {
  idx = bucket index of P;

  lock(&buckets[idx]);

  head = buckets[idx];
  new = next_bucket;
  next_bucket++;
  nodes[new].next = head;
  nodes[new].p = i;
  buckets[idx] = new;

  unlock(&buckets[idx]);
}
```

Figure 4.12: Threads code for bucketing points.

containing the lock variable and obtain exclusive ownership of it. Since this is likely to involve a remote fetch, the cost of this operation is well-modeled using the `permute` benchmark. We will call this cost R. The rest of the loop performs simple arithmetic, and updates local data structures. This cost is well-modeled by the `elementwise-add` benchmark. We will call this cost A. With P processors, $A \approx 8$ microseconds per point per processor, while R is approximately 16 microseconds per point per processor. The cost of the above loop will then be $(R + 3A)(n/P)$, or about 40 μs per point per processor.

After bucketing the points, each thread executes the same algorithm as the serial spiral search algorithm. Figure 4.13 describes the structure of the inner loop: The cost of each iteration in this loop is bounded by $3R$ for the remote fetches of the bucket and point data, $8A$ for bucket index and distance computations, and another $2A$ for the conditional update of the nearest neighbor array. This is a total of 128 μs per iteration, or 2176 μs per site. This is a very inaccurate upper bound on the real performance of the algorithm. The problem is that the parameters R and A both implicitly charge for memory operations that are not actually occurring in this loop.

To remedy this, we first note that each distance calculation has the following structure:

```
load data for p
distance-calc(p,q) {
   load data for q;
   locally calculate dist(p,q);
   conditionally update nearest-neighbor;
}
```

Now, we can easily write a small synthetic program (like the benchmarks from Chapter 3) to simulate this behavior, and use its performance to analyze the inner loop of the spiral search algorithm. Figure 4.14 shows the performance of the KSR on this benchmark. As before, the mean is plotted with the number of threads for that trial, and the vertical line indicates the confidence

```
// Let p be the query point.
// Let l be the number of layers that we need to search.
// Let x,y be the bucket that the query falls into.
// There are n*n buckets.

minx = x - layer; if (minx < 0) minx = 0;
miny = y - layer; if (miny < 0) miny = 0;
maxx = x + layer+1; if (maxx > n) maxx = n;
maxy = y + layer+1; if (maxy > n) maxy = n;
min = 1.0e99;
for(i=minx; i < maxx; i++) {
  for(j=miny; j < maxy; j++) {
    for each point q in bucket[x,y] {
       d = dist(p,q);
       if (d < min) { min = d; NN[p] = q};
    }
  }
}
```

Figure 4.13: Inner loop of the spiral search.

Procs.	Benchmark	$17 \cdot$ Benchmark	Actual	% Error
4	3.197424	54.35	70.369640	22%
8	1.683282	28.56	36.354937	21%
16	0.931272	15.81	18.354905	6%
32	0.618577	10.50	11.544979	9%

Table 4.1: Comparison of synthetic benchmark values to actual runtime.

interval for the sample.

The performance of the synthetic program indicates that the cost of each distance calculation is dominated by the time it takes to fetch information on `q`. This isn't surprising, since the algorithm distributes the point data in a basically random fashion. With the data from runs with 32 processors, a linear regression model predicts that the cost of the benchmark loop is $308 + .63n$ microseconds. If this is the true constant, cost of the inner loop of the algorithm is approximately 20 μs per distance computation. Therefore, the cost of the inner loop is about 340 μs per point per processor, and the whole search costs 380 μs per point per processor.

Comparing this estimate with the actual performance of the program shows that our performance model is accurate, at least for large problems. Figure 4.15 shows the performance of the concurrent spiral search algorithm on the KSR-1 for 4, 8, 16 and 32 processors. The figures show that the synthetic program is an cost model for the actual application, at least asymptotically. From our earlier experiments, we know that the spiral search algorithm performs roughly $17n$ distance calculations. For $n = 10,000$, the average values shown in Figures 4.14 and 4.15 for each machine size are shown Table 4.1. When we remember that that `init_buckets` takes up about 10% of the total runtime of the algorithm, it is apparent from the table that the runtime of the parallel algorithm is within 10% of 17 times the runtime of the benchmark.

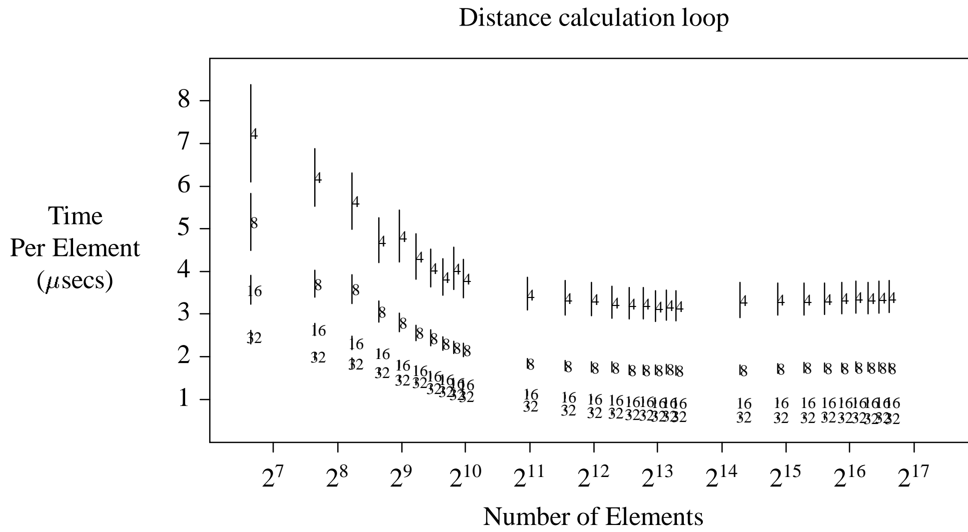


Figure 4.14: Performance of the synthetic benchmark.

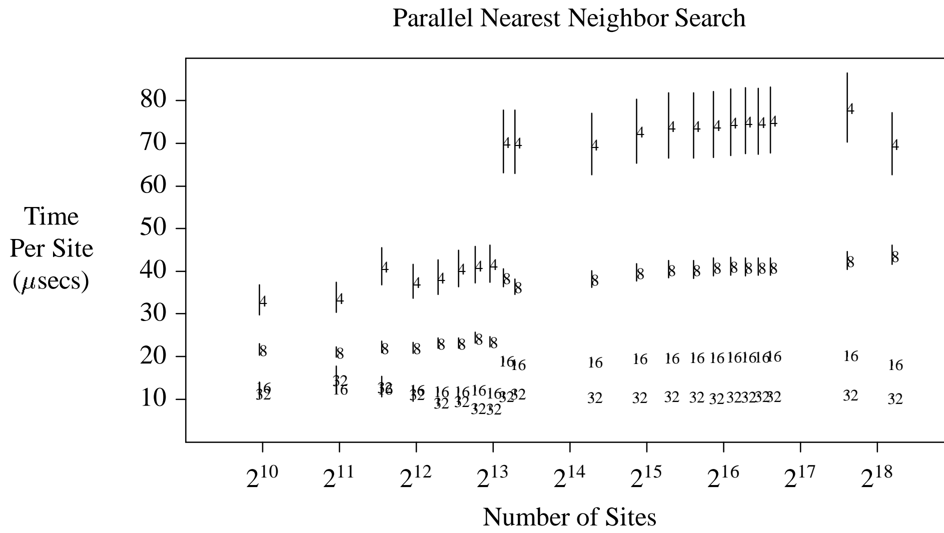


Figure 4.15: Performance of the concurrent local search algorithm on the KSR-1.

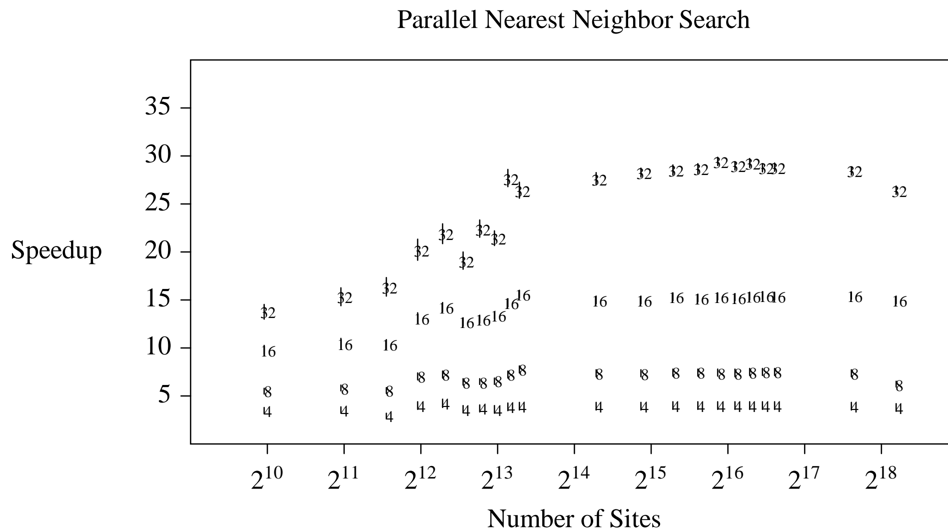


Figure 4.16: The parallel algorithm compared to one KSR node. The plot symbols are the number of threads for each test.

The following profile shows us why this is so. The spiral search program spends almost all of its time in the inner loop modeled by our benchmark:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
46.4	107.04	107.04	400000	0.27	0.27	find_pt [2]
29.0	173.94	66.90				next [4]
8.5	193.50	19.56				drand48 [5]
5.6	206.50	13.00	40	325.00	325.00	init_buckets [10]

Since this is a profile of the entire execution, the time needed to generate the points is included. The two routines `drand48` and `next` are executed during this phase. In the search phase, `init_buckets` performs the bucketing loop and `find_pt` performs the spiral search. A more detailed profile of `find_pt` shows that the inner loop of the spiral search accounts for most of the runtime.

Finally, we need to compare the performance of the parallel algorithm with the performance of the original algorithm. We will compare the parallel algorithm against both a single processor of the KSR and the SPARC-2 workstation from before. The first comparison provides information about how the algorithm will scale on a given parallel architecture, while the second evaluates the algorithm on a cost/performance basis.

Figure 4.16 shows the speedup of the parallel algorithm over the serial one running on one KSR-1 processor. As before, the mean is plotted with the number of threads for that trial, and the vertical line indicates the confidence interval for the sample. The graph shows that the algorithm scales well as we add processors. The main sources of overhead are lock contention in the bucket phase, and communication overhead in the search phase. The former does not contribute much to the total runtime of the algorithm, and the latter becomes small as needed cache blocks are replicated. Therefore, the algorithm obtains close to optimal speedup.

Figure 4.17 compares the parallel spiral search algorithm against the serial spiral search running on my SPARC-2. The relative speedups are better for small problems sizes because on

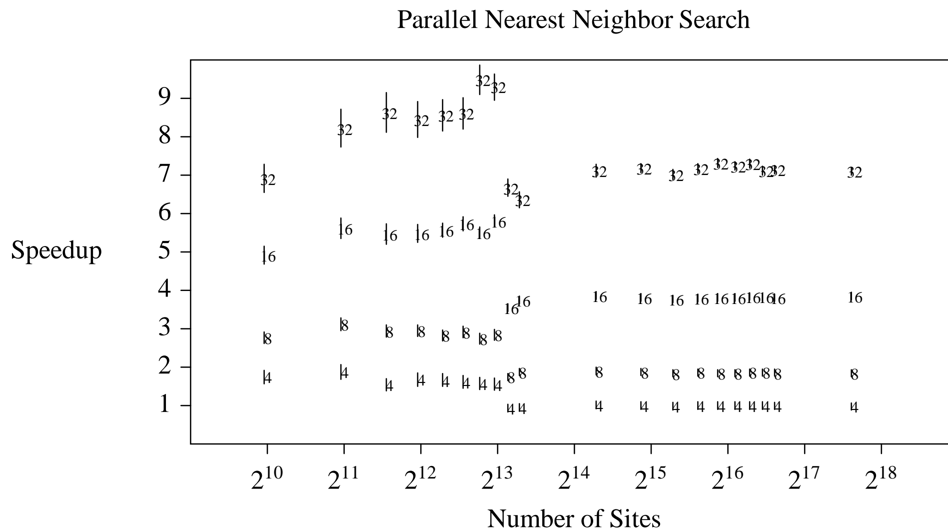


Figure 4.17: The parallel algorithm compared to a Sparcstation.

those problems the parallel algorithm makes better use of the local data cache. However, once the KSR processors begin to depend on their large main memories for data, the relative performance of the parallel algorithm degrades. This is caused by overhead associated with maintaining a virtual address space that is shared by many processors, including cache directory lookups, fetching remote cache blocks, and so on. This memory overhead, and the fact that each KSR processor is relatively slow compared to the SPARC CPU are the main obstacles to obtaining better speedups with this algorithm.

4.8 Summary

This chapter has presented the design and analysis of a simple, parallel algorithm for the all-nearest-neighbors problem. The algorithm executes efficiently on both centralized and distributed memory systems, and may be used as a basis for solving many other proximity problems for points that are independently chosen from smooth probability distributions. In addition, we have constructed accurate performance models of both implementations by combining the use of well-known techniques in algorithms analysis and simple cost models derived from benchmarking the primitive operations of the vector programming model on the target machines. Finally, we saw how to use additional synthetic benchmarks to model the cost of the inner loop of the algorithm when the bounds given by the standard cost parameters were inaccurate.

The main purpose of this chapter was to illustrate that it is possible to design effective algorithms that can both be implemented and accurately analyzed. The concurrent local search idea works very well in this context, because it allows many concurrent threads to process nearest neighbor queries without much contention on the main data structure. However, it is not a general solution to the problem. In particular, when the points were not uniform, we saw that the performance of the algorithm degraded rather drastically. Thus, an interesting avenue for future work would be to investigate the design of methods that are not as sensitive to worst-case inputs as the simple spiral search. Two promising techniques would be using sampling to construct two-level bucket structures or Bentley's "semi-dynamic" k-d trees[Wei78, Ben90].

Chapter 5
Parallel Delaunay Triangulation Algorithms

Dragons of the highest
probability were everywhere,
even in the streets of the capitol,
and the place literally swarmed
with virtuals.
—Stanislaw Lem [Lem85]

This chapter begins our study of parallel algorithms for constructing the Delaunay triangulation. Sections 5.1 through 5.3 will survey the current theoretical literature on this problem, and will also provide a basic overview of randomized divide and conquer, which has recently emerged as a useful technique in parallel algorithms design. It is the basis of several fast sorting algorithms [BLM+91, PHR92], and the technique extends to geometric algorithms. Section 5.4 presents the framework in which we will design such algorithms, and sections 5.5 through 5.8 will propose and analyze a parallel randomized divide and conquer algorithm for constructing the Voronoi diagram.

While the theoretical analysis suggests that the algorithm will be able to make effective use of a multiprocessor, the experimental analysis of this algorithm indicates that its constant factors are too high to make it effective in practice. Never the less, it is useful to study this algorithm because it provides necessary background in randomized algorithms and probabilistic analysis, and because it illustrates how careful experiments can be used to evaluate algorithms before embarking on a time-consuming implementation project.

5.1 Parallel Divide and Conquer

In her thesis, Chow [Cho80] presents a PRAM algorithm that constructs the Voronoi diagram of n points in $O(\log^3 n)$ time on $O(n)$ processors. The algorithm makes use of a well-known relationship between the k-dimensional Voronoi diagram and the $k+1$-dimensional convex hull [Dwy88, GS85]. Let S be a set of n sites in the plane, and imagine projecting them into three dimensions using the function $f(x, y) = (x, y, x^2 + y^2)$. The sites are projected onto a paraboloid of revolution in three-space. Now, suppose that a site (x, y) lies inside the circle centered at (p, q) with radius r. Then

$$(x - p)^2 + (y - q)^2 \leq r^2$$

which is true if and only if

$$(x^2 + y^2) \leq p^2 + q^2 - r^2 - 2px - 2qy.$$

Thus, (x, y) lies within the circle if and only if its projection lies below the hyperplane defined by the right hand side of the equation. It follows that the circle is empty if an only if the corresponding

halfspace does not contain the projection on any site. Thus, empty circles the plane in correspond to empty half-spaces in space, so computing the convex hull of the projected point set is equivalent to computing the Delaunay triangulation, or Voronoi diagram of S. Many of the algorithms that we discuss in this chapter will use this relationship by computing three dimensional convex hulls rather than directly computing the Voronoi diagram.

Aggarwal, et. al. refine Chow's work by computing the Voronoi diagram directly using a parallel version of Guibas and Stolfi's algorithm [ACG$^+$88]. To parallelize this algorithm, one must determine a way to implement the merge step in parallel. This is non-trivial, since in Guibas and Stolfi's algorithm, each iteration of the merge loop depends on the cross-edge found by the iteration before it. In parallel, we must find a way to find all the cross-edges at once.

The algorithm accomplishes this indirectly, by constructing a *contour* of Voronoi edges passing between L and R. Each of these edges is dual to a cross edge. The key to finding these edges in parallel is to first find edges in Vor(L) and Vor(R) that would cross such a contour. It follows that a edge in Vor(L) crosses the contour if and only if the nearest neighbor of one of its endpoints is in R. Therefore, to find the cross-edges between L and R, it is sufficient to find the nearest neighbor, in R, of each Voronoi vertex in L and vice versa. Then, using this information, the algorithm can determine how to connect the cross-edges into the merged diagram. Processing nearest-neighbor queries in the Voronoi diagram is equivalent to planar point location, and at the time, the best-known worst case bound on that problem was $O(\log^2 n)$ on $O(n)$ processors.

Therefore, Aggarwal's algorithm is more clever than this. It builds two specialized point location data structures for the region of the plane between L and R. The first allows the algorithm to locate points within Voronoi regions that intersect the boundary of the convex hull of L. The other allows the algorithm to quickly determine when a vertex in the Voronoi diagram of L is closer to R than L, or vice versa. These data structures allow the algorithm to find the cross-edges in $O(\log n)$ time, so the whole algorithm has a cost of $O(\log^2 n)$ on $O(n)$ processors.

Goodrich, Cole and O'Dunlaing [CGO90] refine this method with even more sophisticated data structures, and some non-trivial scheduling techniques to obtain an algorithm that runs in $O(\log^2 n)$ time on $O(n/\log n)$ processors, thus matching the optimal work bound.

Neither of these two algorithms is particularly practical, since they both depend on complex, expensive data structures. In addition, they are both designed for $O(n)$ processor PRAM machines, and simulating this environment on current multiprocessors is not practical.

There has also been some investigation of divide and conquer techniques from outside the PRAM community. Lu [Lu86], and Jeong and Lee [JL90] describe mesh algorithms for this problem. Lu uses a dual algorithm to obtain runtime of $O(\sqrt{n}\log n)$ time on a $\sqrt{n} \times \sqrt{n}$ mesh, while Jeong and Lee compute the diagram directly in $O(\sqrt{n})$ time, which is optimal for the mesh. In addition, Stojmenovic [Sto86] describes an algorithm designed for hypercube architectures. These algorithms use basically the same techniques as the Aggarwal algorithm, but with some variation in the actual details. Again, these results are primarily of theoretical interest, because none of them were designed with real-world architectural constraints in mind.

5.2 Optimal Expected Time Algorithms

It is possible to obtain more efficient algorithms when the input is assumed to be uniformly distributed. Levcopoulos, Katajainen and Lingas [LKL88] use a combination of bucketing and Aggarwal's algorithm to obtain an algorithm that runs in $O(\log n)$ expected time on $O(n/\log n)$ processors. Their algorithm works in two phases. First, the points are placed in buckets and the

algorithm computes the "inner" diagram, which is the intersection of the Voronoi diagram with the unit square. Then, the algorithm computes the "outer" diagram, which contains the Voronoi edges that lie outside the unit square.

Recall that given a point set S, for each site $s \in S$, $V(s)$ denotes the Voronoi polygon of s. The first phase of the algorithm uses a hierarchical bucketing scheme to find, for each site s, a "rectangle of influence", called $RI(s)$ containing all the points in the set that may contribute to $V(s)$. They show that the total expected size of all these rectangles is $O(n)$, and that the inner diagram can be computed in $O(\log n)$ expected time with $O(n/\log n)$ processors. They then show that the expected number of sites that contribute to the outer diagram is small enough that a brute force algorithm can compute the rest of the diagram in $O(\log n)$ expected time with $O(n/\log n)$ processors. The structure of this algorithm is similar to the serial algorithm presented by Bentley, Weide and Yao[BWY80], but the hierarchical bucket structure is more general than the single level data structure used in that paper.

MacKenzie and Stout [MS90] present another algorithm along these lines, but with a much faster theoretical runtime. Using a sophisticated and complicated bucketing scheme, their algorithm is able to construct the Voronoi diagram in $O(\log\log n)$ expected time on $O(n/\log\log n)$ processors. The algorithm uses a primitive which MacKenzie and Stout call "padded sort" which is able to bucket the sites in $O(\log\log n)$ time. Then, the algorithm computes the inner diagram using spiral search, and the outer diagram using brute force. An analysis similar to Levcopoulos, Katajainen and Lingas is enough to bound the cost of the brute force construction by $O(\log\log n)$.

Most recently, a paper by Venmuri, Varadarajan and Mayya [VVM92] shows how to modify the standard divide and conquer algorithm to run in $O(\log n)$ expected time on n processors. This algorithm divides the original problem into vertical strips, constructs the Voronoi diagram of each strip, and then constructs the dividing chains of the strips in parallel. The algorithm now takes advantage of the fact that only dividing chains that intersect need further processing. Therefore, rather than recursively merging all pairs of subproblems, this algorithm constructs the rest of the diagram by first examining the current set of dividing chains for intersection points and then continuing the constructing process from those vertices. When no more intersections occur, the algorithm is finished. Using techniques similar to those in Dwyer's thesis [Dwy88], the authors show that the expected complexity of this scheme is $O(\log n)$ time since under the uniform distribution assumption, the probability that a given dividing chain intersects a large number of other chains is low.

Each of these algorithms use ideas that are basically the same as the fast expected time sequential algorithms. The first two are doing spiral search, while the last one is combining bucketing with divide and conquer the way Dwyer's algorithm did in Chapter 2. However, for the sake of achieving "optimal" asymptotic runtimes within the PRAM model, each of these algorithm has been made too complex to really be practical. Not only do the algorithms use esoteric and unnecessary data structures and scheduling techniques, the runtime analysis of each hides huge constant factors within the confines of their asymptotic bounds.

For example, MacKenzie and Stout use a bucketing technique which they call "padded sort" to fill the buckets in their spiral search algorithm. This algorithm runs in nine stages. The first six alternate between attempting to place items into bins and reallocating items which have failed. The last three sort the bins themselves. The reallocation procedure is similar to the `pack` operation from the previous chapter, but is complicated by the fact that it must run in $\Theta(\log\log n)$ time.

What all of this amounts to is an algorithm that is similar to the vectorized bucketing scheme

in Chapter 4. The algorithm attempts to bucket one item per processor, but if some items collide, then it must iteratively pack away the finished items and retry the failed ones. We have seen that this simple idea, without the trappings of PRAM complexity analysis, works extremely well on vector machines like the Cray, while the locking-based algorithm for achieving the same effect works very well on MIMD machines like the KSR-1. In addition, the analysis of both of those algorithms showed that the bucketing step was not the main bottleneck in the program, the searching steps were. Therefore, not only do the PRAM results present unnecessary optimizations, they optimize a relatively minor part of the algorithm.

Of course, this criticism should not be seen as an attempt to question the integrity or taste of the authors. Their stated goal was to study basic questions about the complexity of parallel algorithms, not to address the problem of constructing efficient parallel programs for these problems. What this example does illustrate is that it is often the case that the elegant and sophisticated proofs and techniques used by theoreticians to prove their complexity results are often based on very simple and practical ideas.

In fact, some practical work has been done on algorithms similar to the ones described above. Merriam [Mer92] has implemented a parallel algorithm for three dimensional Delaunay triangulation construction based loosely on the ideas in these papers. His algorithm uses a k-d tree and many iterated nearest-neighbor queries to construct the edges of the Delaunay triangulation one at a time. This method is similar to the algorithms of Maus [Mau84] and Dwyer[Dwy88]. Merriam's implementation is for the Intel Paragon, and uses message passing to maintain the distributed data structures and coordinate the searches. His experiments indicate that the algorithm's runtime is $O(n \log n/p)$.

In addition, Teng, et. al. [TSBP93] have implemented a similar algorithm, also for the three dimensional problem, in a vector style on the Connection Machine CM-2 and CM-5. Their algorithm uses buckets to perform searches, and their experiments indicate that the runtime of the algorithm is $O(n/p)$ on n sites and p processors.

5.3 Randomized Algorithms

In Chapter 2 we saw that by randomizing the order in which we processed insertions in the incremental Delaunay triangulation algorithm, we could bound the expected number of updates that the algorithm performed without regard to the distribution of the point set. Intuitively, the reason for this behavior was that a random insertion into the Delaunay triangulation of a random sample of the point set is unlikely to cause many edge flips.

This result is actually a special case of a general framework for designing randomized geometric algorithms, both sequential and parallel. To motivate the formal machinery that we will present later, consider the problem of sorting a set S of n real keys. Insertion sort is a simple way to do this. Each key from S is added to a set R in some order, and R is kept in sorted order as each key is inserted. The problem with this algorithm is that finding the correct position to place a new key can be expensive. To alleviate this, for each key $k \in S - R$, we store the interval (a, b) in the sorted set R that k belongs to. Alternatively, for every pair of adjacent keys a and b in R, we store a list of keys from S that belong in the interval from a to b. Now, to insert a new key, k, look up the two keys, a and b that k lies between, and we split the list of keys belonging to the interval (a, b) into one for (a, k) and another for (k, b). The resulting algorithm just a version of quicksort, and as a result, shares quicksort's worst case. To alleviate this, we may insert the keys of S into R in random order, in which case the lists for each interval in R will be roughly the same size, and the

resulting algorithm has an expected runtime of $O(n \log n)$.

We can also use this idea to create a divide and conquer algorithm. Here, we pick a subset $R \subset S$ at random, and sort it. Then, for each key $k \in S - R$, we use binary search to find the interval in R that k belongs to, and add k to a subproblem for that interval. Finally, we sort the subproblems one by one. Many sorting algorithms for parallel and distributed architectures use this scheme, because the sample provides a convenient way to split the original problem into many subproblems that are solvable in parallel. Blelloch, et al survey these algorithms and describe a high performance algorithm designed for the Connection Machine CM-2 [BLM+91]. A later paper by Prins, Hightower and Reif presents a similar study for the MasPar MP-1 [PHR92].

The key idea behind these algorithms, which we will explore in the context of geometric algorithms, is to use oversampling to obtain good load balancing with high probability. That is, with P processors available, rather than choosing a sample of size P, we choose a sample of size kP, sort it, and then form R with the elements that have ranks $k, 2k, 3k, \cdots, (P-1)k$. Blelloch, et al show that using this scheme, the probability that the largest subproblem is more than α times as big as the mean is less than

$$N e^{-(1-1/\alpha)^2 \alpha k/2}.$$

for $\alpha > 1$. In practice, the results that their implementation obtains are somewhat better than this.

5.4 Preliminaries: The General Framework

Clarkson and Shor [CS89, CMS92] present a generalized formal framework for describing and analyzing randomized geometric algorithms. We will present this framework here to motivate the fact that, in principal, it could be applied to the design of a wide range of parallel algorithms. To keep the discussion relevant, our running example will show how to apply the ideas to the construction of the Delaunay triangulation.

Let S be a set with $|S| = n$ elements, called *objects*. Let $\mathcal{F}(S)$ be a multiset whose elements are non-empty subsets of S. The elements of $\mathcal{F}(S)$ are called *regions*. Let b be the size of the largest element of $\mathcal{F}(S)$. If all of the elements of $\mathcal{F}(S)$ have size b, then we say that $\mathcal{F}(S)$ is *uniform*. Let $F \in \mathcal{F}(S)$ and $x \in S$ be given. Then we say that F *relies* on x, or x *supports* F, if $x \in F$. For each $R \subset S$, define $\mathcal{F}(R)$ to be $\{F \in \mathcal{F}(S) \mid F \subset R\}$. Finally, for each particular problem, we will define a *conflict* relation between S and $\mathcal{F}(S)$. If $R \subset S$, then $\mathcal{F}_0(R)$ will denote the set of $F \in \mathcal{F}(R)$ that conflict with no object. The nature of the conflict relation depends on the problem at hand, so the term "conflicts with" will have a different meaning for each problem.

For concrete example, for the convex hull problem, S is the set of points in d-space while $\mathcal{F}(S)$ is made up of half-spaces defined by sets of d points. Assuming that S has no degeneracies, $\mathcal{F}(S)$ is uniform, with $b = d$. Each set $F \in \mathcal{F}(S)$ of d points defines a hyperplane h_F. Assuming that we translate S so that the origin is inside the convex hull, we associate with each $F \in \mathcal{F}(S)$ the halfspace on the opposite side of h_F from the origin. A point x conflicts with $F \in \mathcal{F}(S)$ if it is contained in the corresponding half-space. Again, those regions in $\mathcal{F}_0(S)$ will define the convex hull. Figure 5.1a illustrates this case.

Similarly, in the Delaunay triangulation problem, S is the set of sites, each element of $\mathcal{F}(S)$ is a triple that defines a possible Delaunay triangle. Again, assuming no degeneracies, $\mathcal{F}(S)$ is uniform and $b = 3$. In an abuse of notation, we will identify each of these triples with the circumcircle that they define. Therefore, the conflict relation will specify that a site x conflicts with a region F if x lies within the disk defined by F. The goal of any algorithm that constructs the Delaunay triangulation is to build $\mathcal{F}_0(S)$. Figure 5.1b illustrates this case.

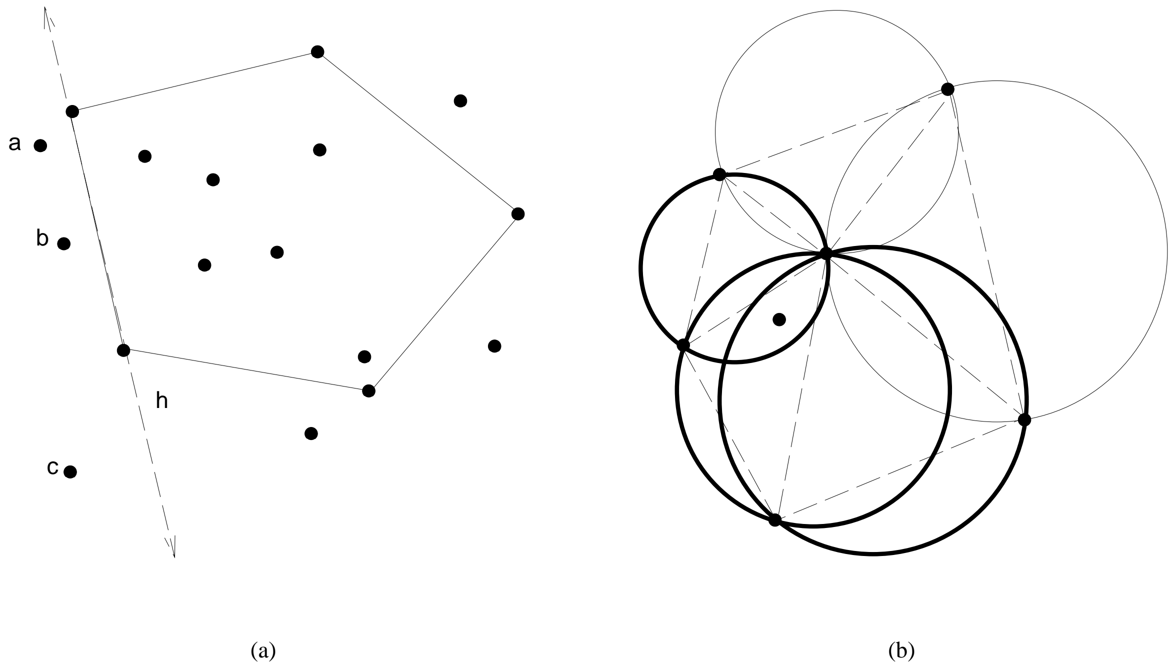


Figure 5.1: Objects, regions and the conflict relation. In (a), the three points are in conflict with the half-space. In (b), the site conflicts with each circle that it lies in.

5.5 Randomized Divide and Conquer

We can also use the Clarkson and Shor framework to develop divide-and-conquer algorithms based on random sampling. This is a generalization of the technique used in the sample-based sorting algorithms. The idea is to use a random sample R, of S to divide S into many independently solvable subproblems. After choosing R, we construct $\mathcal{F}_0(R)$ and for each region $F \in \mathcal{F}_0(R)$, let con(F) denote the number of objects in $S - R$ that conflict with F. The following result characterizes the performance of such algorithms. Define f_r to be the expected value of $|\mathcal{F}_0(R)|$ for a random subset R of S with $|R| = r$. In addition, let

$$F(r) = \max_{1 \leq z \leq r} f_z.$$

Theorem 5.1. Given S and R as above, with $r = |R|$, assume that $\mathcal{F}(S)$ is uniform with size b, and that $|\mathcal{F}(S)| \leq K\binom{n}{b}$ for some constant K. It follows that for any $q \geq 0$, there exists a constant $c_{\max}$ such that with probability $3/4 - e^{-q}$, the following conditions hold:

$$\sum_{F \in \mathcal{F}_0(R)} \text{con}(F) = O(n/r)F(r) \qquad \text{(T)}$$

and

$$\max_{F \in \mathcal{F}_0(R)} \text{con}(F) \leq c_{\max}(n \log r)/r \qquad \text{(L)}.$$

This is a somewhat restricted version of Corollary 3.8 from Clarkson and Shor's original paper [CS89]. We can use this result to analyze algorithms that fit the following generic framework.

Algorithm 5.1 (RD). Randomized divide and conquer.

1. Given the a set of S objects, the goal is to construct $\mathcal{F}_0(S)$. Choose a subset R of S of size r at random.

2. Construct $\mathcal{F}_0(R)$.

3. For each $F \in \mathcal{F}_0(R)$, let P_F be the subproblem for F.

4. For each $x \in (S - R)$, find every region $F \in \mathcal{F}_0(R)$ that conflicts with x and place x into P_F.

5. For each $F \in \mathcal{F}_0(R)$, recursively solve P_F and use the results to compute $\mathcal{F}_0(S)$.

Theorem 5.1 tells us that on average, the sample R can be used to split the original problem into equally sized subproblems whose total size is not much larger than the original. In the context of parallel algorithms, this result is especially interesting because for the problems that we are interested in, the subproblems are independent, and can be solved concurrently.

Reif and Sen [RS89] independently developed a refinement of this idea to design an efficient PRAM algorithm for constructing 3-D convex hulls. Their algorithm actually solves the dual problem, constructing an intersection of half-spaces. In this case, S is made up of half-spaces. Given a sample $R \subset S$, let $\mathcal{P}(R)$ be the intersection of the half-spaces in R, and assume that we know some point $p*$ in the interior of $\mathcal{P}(R)$. We can then partition $\mathcal{P}(R)$ into cones in the following way: take an arbitrary hyperplane h and cut the faces of $\mathcal{P}(R)$ into trapezoids. Then triangulate each of the trapezoidal faces, and connect the vertices of the triangles to $p*$. We will call the resulting set of cones $\Delta(R)$. For each $F \in \Delta(R)$, a half-space $s \in S$ conflicts with F if its bounding plane intersects F.

The following result follows directly from Theorem 5.1:

Lemma 5.2. Using the terminology above, there exist constants k_{total} and k_{max} such that the following conditions hold with probability at least 1/2:

$$\sum_{F \in \Delta(R)} \text{con}(F) \leq k_{\text{total}}(n/r)|\Delta(R)|$$

and

$$\max_{F \in \Delta(R)} \text{con}(F) \leq k_{\text{max}}(n/r)\log(r).$$

If a sample $R \subset S$ satisfies these two conditions, we will call it "good", otherwise, we will call it "bad."

Reif and Sen's algorithm follows the basic outline of Algorithm RD, but in order to achieve an optimal PRAM runtime, it must address the following issues that do not come up in a serial algorithm:

- The sampling algorithm must be able to choose a good sample with high probability in order to bound k_{max} and guarantee good load balancing in the recursive calls. To deal with this problem, Reif and Sen present a sampling scheme called "polling" that processes $O(\log n)$ samples in parallel in $O(\log n)$ time. It does this by checking each sample against

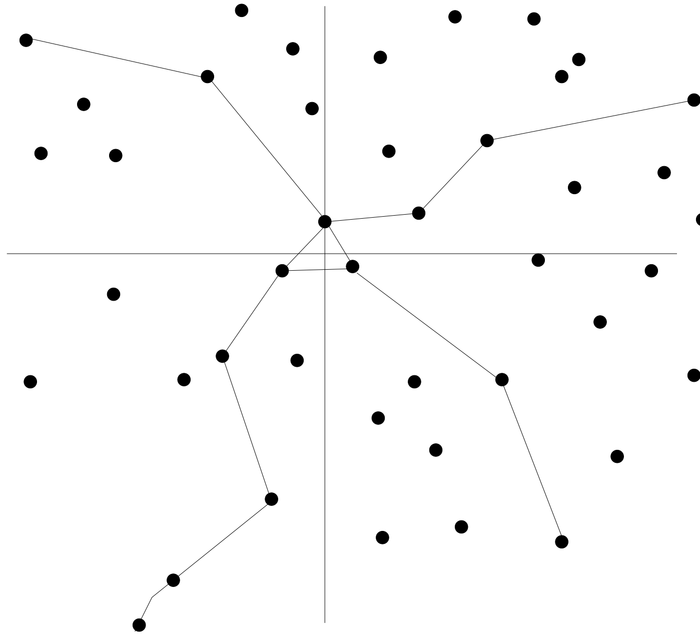


Figure 5.2: Dividing around a convex hull facet. This is just a schematic drawing, and isn't an exact representation of the execution of the algorithm.

a random sample of the input, rather than the entire set S. They then use Chernoff bounds [Che52, HR89] to show that will give an accurate estimate of the actual quality of the sample. Testing $O(\log n)$ samples guarantees that the algorithm will fine one "good" one with high probability. Polling is a generalization of the oversampling schemes that recent implementations of randomized sorting algorithms have used to guarantee good load balance [BLM$^+$91, PHR92]. We will see later that in practice, a similar, somewhat simpler scheme provides the same result.

- The algorithm must be able to compute the intersections between the cones and the halfplanes in $O(\log n)$ time, even though one plane may cut $O(n^\epsilon)$ cones, where $\epsilon > 0$. To do this, Reif and Sen take advantage of the fact that the random sample is small compared to n, and so they can use a trade time for processors and use a brute force approach.
- In a naive algorithm, the total size of the subproblems increases by a factor of k_{total} with each level of recursion. Reif and Sen present a scheme to filter redundant half-spaces out of recursive calls and show that this guarantees that the total number of processors needed for process the call tree will be $O(n)$.

Goodrich and Ghouse [GG91] use a somewhat different algorithm to compute the 3-d convex hull. Their algorithm picks a splitting point at random, and uses linear programming to find a facet of the hull above this point. The algorithm then uses this facet to split the original problem into four parts (see Figure 5.2). The partitioning routine works by projecting the sites onto the xz and yz planes and computing two two-dimensional convex hulls. All of these edges are edges in the 3-d hull, and when they are projected down to the xy plane they split the sites into four subproblems.

The algorithm then solves the subproblems recursively. Using some scheduling tricks that are similar to the ones in the MacKenzie paper, the authors prove a time bound of $O(\log^2 n)$ and a work bound of min($n \log^2 h, n \log n$) where n is the number of points and h is the number of hull facets. Each of these is a high probability bound. This is a parallel, randomized version of the 3-d hull algorithm due to Edelsbrunner and Shi [ES91] which was an extension the planar hull algorithm of Kirkpatrick and Seidel [KS82].

These algorithms again follow a familiar pattern. While the ideas underlying them are relatively simple, a large amount of complexity and overhead has been added in order to obtain "efficient' or "optimal" time and work bounds in the PRAM model. Therefore, while we cannot seriously consider implementing these techniques directly, we can use the basic ideas to design and evaluate simpler, possibly more practical algorithms.

5.6 A More Practical Randomized Parallel Algorithm.

We now turn to the question how to apply the theoretical ideas in the previous section into practical solutions to the problem. The goal is to be able to construct a parallel algorithm for constructing the planar Delaunay triangulation that exhibits a substantial performance advantage when compared to fast sequential algorithms on the same inputs.

Our algorithm will follow the style of Reif and Sen's algorithm because it has the following potential advantages over the other algorithms that we have discussed:

- The algorithm is guaranteed to achieve its expected performance with high probability independently of the distribution of the input.
- The algorithm is based on a simple framework. It is unclear how to simplify the divide and conquer algorithms without taking a substantial performance hit. In addition, the spiral search based algorithms tend to be be awkward because two separate algorithms are needed for the inner and outer diagrams.
- The principle advantage of the convex hull algorithm of Goodrich and Ghouse is that its work bound is output-sensitive. However, this is no advantage when computing Delaunay triangulation s, since we know that every projected site will be on the convex hull. Therefore, the extra complexity of linear programming, point location along the contours, and adaptive scheduling will gain us nothing.
- The Reif and Sen algorithm can be easily adapted to compute the Delaunay triangulation directly.
- Finally, recent results have already shown that randomized divide and conquer is extremely effective for constructing fast sorting algorithms. Thus, we might expect that the technique would be equally effective for geometric problems.

Our algorithm will follow the generic outline of algorithm RD. We will compute the Delaunay triangulation directly, rather than concentrating on computing convex hulls because the more general algorithm carries extra overhead that will only waste time. In addition, we will be comparing the algorithm against the incremental algorithm from Chapter 2, and we need to make the comparison a fair one.

The objects in S will be sites in the plane, the regions in $\mathcal{F}(S)$ will be the disks associated with Delaunay triangles, or equivalently, Voronoi vertices, in the current diagram. It will be more

convenient to describe this algorithm in terms of the Voronoi diagram, rather than the Delaunay triangulation.

For a set of sites S, Vor(S) will denote the Voronoi diagram of S, DT(S) will denote the Delaunay triangulation of S. For each site $s \in S$, $V(s)$ will denote the Voronoi region of s and DN(s) will denote the set of sites in S that are connected to s in DT(S). Our algorithm will work in the following three stages:

Algorithm 5.2 (RDDT). Randomized divide and conquer for constructing the Delaunay triangulation.

1. Let $k \geq 1$ be a constant that we will choose later. Assuming that $|S| = N$ and we have P processors, choose a sample $R \subset S$ of size kP. We do this by having each processor choose k sites, at random, from S. Construct Vor(R) using either a brute force parallel method or, if P is small, just use one processor to build the diagram sequentially. The idea of picking more than one sample per processor comes from recent experience with randomized sorting algorithms[BLM+91, PHR92]. The idea here is that while the size of any given subproblem may differ greatly from the expected value cn/kP, it is unlikely that the total size of k subproblems chosen at random will be far away from cn/P.

2. For each $r \in R$, let P_r be the subproblem of r. Then P_r must contain every site in $S - R$ that could change the structure of $V(r)$. For each site $s \in S - R$, search Vor(R) for circles that conflict with s. Each circle that we find will be associated with three sites in R. The algorithm places s in the subproblems of each of these sites.

3. For each $r \in R$, construct Vor(P_r) using an efficient sequential algorithm. Then remove edges and vertices in Vor(P_r) that lie outside the Voronoi polygon of r in Vor(R). We will see that these vertices are redundant, and do not belong in the final answer.

To prove that Algorithm RDDT is correct, we need to show how to correctly construct the subproblems in step 2, and that step 3 actually reports all of the valid vertices in Vor(S).

The algorithm for step 2 is more easily explained in terms of the Delaunay triangulation. Assume that we have constructed DT(R) and need to find all the triangles in this diagram that conflict with some site $s \in S - R$. We can do this with a simple modification to the incremental insert procedure in Chapter 2. Figure 5.3 shows the modified procedure.

The procedure `find-circles` starts by locating p in the DT(R). Obviously, p conflicts with the circumcircle of the triangle that it lies in. Thus, p goes into the subproblems of each site supporting this triangle. The routine then finds the rest of the conflicting circles using the same search loop as the the edge flipping loop from Chapter 2. The difference is that rather than flipping edges, and walking through the modified diagram, `find-edges` keeps a queue of edges that are suspect, and keeps walking until this queue is empty. The three edges making up the initial triangle are the original members of this queue. Let ABC be such a triangle with the point A opposite the new site p. If ABC fails the circle test, p goes into the subproblem of A, and the edges AB and AC go into the queue. We don't need to add p to the subproblems of B or C, since they must belong to a triangle that has already failed the circle test (see Figure 5.4)

The following result is not hard to prove.

Lemma 5.3. The routine `find-circles` visits exactly the same set of edges in DT(R) as the incremental insertion procedure would.

```
% Q is a queue whose element are edges.
% p is the point being searched.
find-circles(p)
{
  e := Locate(p);
  if (p == e.Org || p == e.Dest)
    return;
  if (is_on(p,e.Org, e.Dest)) {
    t = e.Oprev;
    e = t;
  }

  first = e.Org;
  init_queue(Q);

  do {  /* p conflicts with initial triangle */
    add p to subproblem for e.Org;
    enqueue(Q, e);
    e = e.Lnext;
  } while (e.Org != first);

  for(;;) {   /* Look for conflicting circles until done */
    e = dequeue(Q);
    t = e.Oprev;

    if (not CCW(t.Dest, e.Org, e.Dest) &&
        in-circle(e.Org,t.Dest,e.Dest, p)) {
      add p to subproblem for t.Dest;
      enqueue(Q, t);
      enqueue(Q, t.Lnext);
    }

    if empty(Q)
      return;
  }
}
```

Figure 5.3: Routine to find circles that conflict with a given site p. This routine is just like the incremental insertion procedure, but we don't change the current diagram at all. Instead, the queue keeps track of the edges that we need to look at.

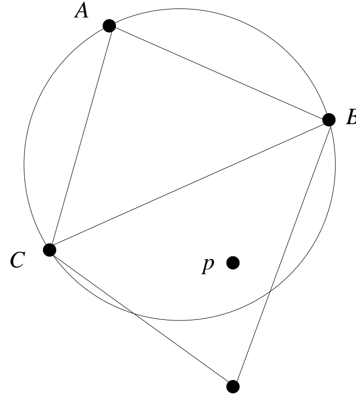


Figure 5.4: The triangle ABC fails a circle test, so p goes to A's subproblem.

Proof. The incremental algorithm checks the three edges of the initial triangle, and each edge of any triangle that fails the in-circle test. In `find-circles`, the search begins with the three edges of the initial triangle, and each time a triangle fails a circle test, the routine adds the two far edges to the queue. Therefore, we can conclude that it will check the same edges as the incremental algorithm does. □

Lemma 5.3 implies that `find-circles` will find all regions that conflict with a particular site. Therefore, we can conclude that we may use this routine to correctly construct the needed subproblems.

To prove the correctness of step 3, we switch back to looking at things from the perspective of the Voronoi diagram. Let $r \in R$ be given, let P_r be the subproblem of r, and let $V_R(r)$ be the Voronoi polygon of r in Vor(R). Let $c(v)$ denote the circle associated with the vertex v and define $\mathcal{C}(r)$ to be the union $c(v_1) \cup c(v_2) \cup \cdots \cup c(v_k)$ where each v_i is one vertex of $V_R(r)$ (see Figure 5.5). The following basic result allows us to show that Algorithm RDDT is correct.

Lemma 5.4. Let $v \in V_R(r)$ be given. Then the circle $c_r(v)$, centered at v and passing through r is included in $\mathcal{C}(r)$.

Proof. Let $v \in V_R(r)$ be given. It follows that $c_r(v) \cap R$ is empty. Let t be the neighbor of r that is closest to v, and let v_1 and v_2 be the two vertices from $V_R(r)$ whose circles pass through both r and t. If $c_r(v)$ is totally included in either of these circles, then we are done. If not, we can grow $c_r(v)$ by moving v directly away from r until the growing circle passes through both r and t. It follows that this new circle is contained in $c(v_1) \cup c(v_2)$. Therefore, we can conclude that $c_r(v) \subset \mathcal{C}(r)$. (see Figure 5.5). □

Lemma 5.5. Let v be a vertex of Vor(P_r). If $v \in V_R(r)$, then v is a vertex of Vor(S).

Proof. We have that $c(v) \cap P_r$ is empty. But Lemma 5.4 implies that that we can expand $c(v)$ until it touches r while staying inside $\mathcal{C}(r)$. Thus, $c(v) \subset \mathcal{C}(r)$. It follows that $c(v) \cap S \subset c(v) \cap P_r$, so no site in S can fall in $c(v)$. Therefore, v is a vertex of Vor(S). □

The following result implies that every valid vertex of Vor(S) will appear in the Voronoi diagram of at least one subproblem.

Lemma 5.6. If v is a vertex of Vor(S), then there exists $r \in R$ such that $v \in$ Vor(P_r) $\cap V_R(r)$.

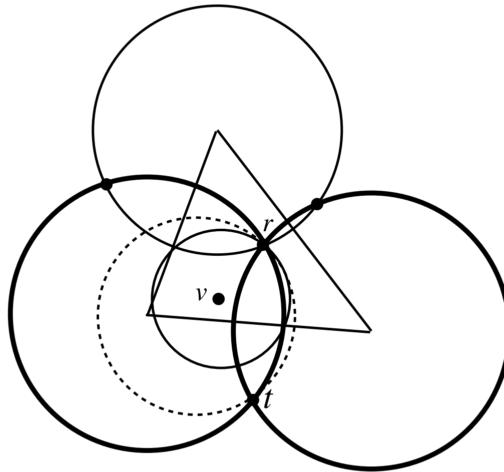


Figure 5.5: The Voronoi region of a point r, and the "flower" of circles surrounding it. Any circle centered in the region and passing through r must be contained in the flower.

Proof. Let $v \in Vor(S)$ be given. Then $c(v)$ passes through three sites $p_1, p_2, p_3 \in S$. We may choose $r \in R$ such that $v \in V_R(r)$. By Lemma 5.4, the circle centered at v and passing through r is a subset of $\mathcal{C}(r)$. Since $c(v)$ is included in this circle, $c(v) \subset \mathcal{C}(r)$ so it follows that $p_i \in \mathcal{C}(r)$, $1 \leq i \leq 3$. Therefore, each of these three sites will be in P_r, so v will be a vertex of Vor(P_r). □

These two lemmas imply that Algorithm RDDT will correctly compute all of the vertices in Vor(S). For each $r \in R$, it suffices to compute Vor(P_r) and report all the vertices in that diagram that fall inside $V_R(r)$. Vertices which appear more than once in the final list can easily be filtered out.

5.7 Analysis

Two factors are critical to the success of Algorithm RDDT. First, the average size of a subproblem must be small, i.e. no more than cn/p with p processors. Second, the ratio between the average subproblem size and the maximum subproblem size must be small. The first condition implies that the expected total work done by the algorithm will be cn, for some constant c. The hope is that c will compare well with the runtime constants of a serial algorithm. The second condition implies that efficiency will not be compromised by any one processor. Together, these conditions imply a fast parallel runtime and low parallel overhead, which should result in good parallel speedups.

We will use the following notation from Section 5.3. The objects in S will be the sites, the regions in $\mathcal{F}(S)$ will be triples of sites defining circles in the plane. A site $s \in S$ conflicts with $F \in \mathcal{F}(S)$ if it lies within the circle defined by F. Let $R \subset S$ be a random sample of S with $|R| = r$. The Condition (T) of Theorem 5.1 implies that the expected size of a subproblem in Algorithm RDDT will be $O(n/r)$.

The remaining problem is to show that the sampling scheme will produce evenly sized subproblems. Here we run into technical problems. As we saw earlier, randomized divide-and-conquer algorithms for constructing the planar Voronoi diagram are generally framed in terms of constructing a three-dimensional intersection of halfspaces. As in the Reif-Sen algorithm above, the divide step of these algorithms partitions the faces of a convex polyhedron into a set of cones, and creates one subproblem per cone. In the case of the Voronoi diagram, this action is equivalent to dividing

the Voronoi region of each sample point into triangular regions.

For the sake of simplicity, Algorithm RDDT does not create subproblems in this way. The routine `find-circles` creates one subproblem per sample point. This makes the algorithm easier to program and easier to understand. In addition, our algorithm has the advantage that it computes the Voronoi diagram directly, rather than going through the intermediate step of constructing an intersection of halfspaces. This is not only a potential performance advantage, but could also be important from the standpoint of numerical stability.

The problem with this scheme is that in theory, it makes the algorithm sensitive to the distribution of the input, since certain worst case inputs can create subproblems of size $O(n)$. The result is that condition (L) of Theorem 5.1 is of no help because Algorithm RDDT does not fit the conditions of the theorem.

To remedy this situation, we can either change the algorithm so that we can use Theorem 5.1 to analyze it, or we can leave the algorithm as it is, and use experiments to give us confidence that `find-circles` behaves as expected. The next section will present a series of experiments that suggests that Algorithm RDDT is not sensitive to the distribution of the input, and that the partitioning procedure almost always produces equally sized subproblems. Given these results, it would be more interesting to try and improve the analysis to account for this behavior, rather than modifying the algorithm to fit the current theory. The main reason for this conclusion is that any modifications would only make the algorithm more complex, and hence would further increase its runtime constants.

5.8 Experiments

Our theoretical analysis suggests that Algorithm RDDT will make effective use of current multiprocessor architectures. However, the analysis was not exact enough to tell us exactly what the runtime constants might be. In order to study the method in more detail, we will gather experimental data using a serial simulation of the bucketing process.

Simulations have long been used to predict the performance of real systems. The simulation that we will use here is somewhat different in that it will be used to measure *abstract*, as opposed to real system costs. McGeoch [McG86] uses simulations of this type to study the performance of many algorithms. She notes that it is often possible to construct simulation programs that accurately measure the abstract cost of an algorithm without actually executing the algorithm in question. For example, the cost of the partition step in quicksort is determined by the rank of the splitter. Thus, we can measure the cost of quicksorting without actually sorting keys.

In this section, we will use a simulation of this type to measure the performance of Algorithm RDDT. The performance of Algorithm RDDT is determined by the size of the subproblems created in step 2 of the algorithm. Therefore, our simulator will execute steps 1 and 2 of the algorithm, and then output the resulting problem sizes. We can then predict the performance of the last phase of the algorithm using the experiments and analysis in Chapter 2.

The implementation of the simulator follows the outline of Algorithm RDDT fairly closely. First, assuming P it picks a sample R of the input set, with $|R| = kP$ and constructs DT(R) using the incremental algorithm. It then uses the routine `find-circles` in Figure 5.3 to place each remaining site into the appropriate subproblems. Finally, assume that the subproblems are numbered $SP_1, SP_2, \cdots, SP_{kP}$. For each processor $1 \leq i \leq P$, the simulator adds up the sizes of $SP_{ik}, SP_{ik+1}, \cdots, SP_{(i+1)k-1}$ and reports the result as the total work needed for processor i.

We will examine two sets of experiments, one to observe the expected value of the work

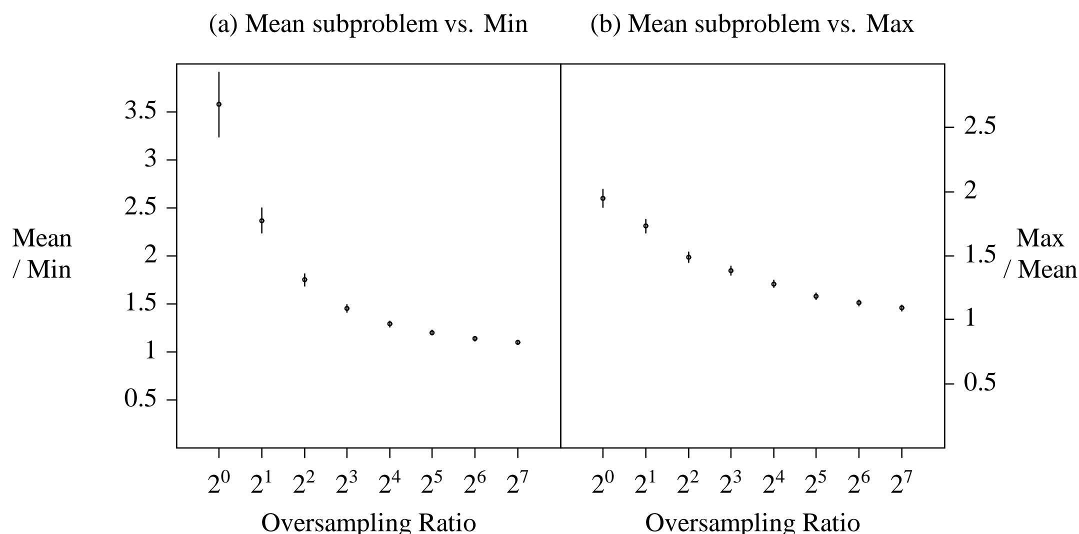


Figure 5.6: Load balancing results for 32 processors.

complexity of the algorithm, and another to observe the expected load imbalance.

Each set of experiments was run assuming a machine with 32 processors and a machine with 128 processors. The first matches the size of our target machine, while the second will give some idea as to how the method will scale to larger machines in the future. First, we will examine the effect of oversampling on the variance in subproblem sizes. For each machine configuration, 100 trials of the bucketing procedure were executed with an input size of 100,000 sites and sample sizes between one site per processor and 128 samples per processor. For this set of tests, the sites were generated from a uniform distribution in the unit square. Unless otherwise noted, we plot the sample mean along with the 90% confidence interval.

We will look at four sets of results. First, Figure 5.6a shows the ratio between the mean and minimum subproblem sizes over this range of inputs. In practice, this ratio is as important as the ratio of the maximum to the mean because a processor holding a small subproblem wastes cycles waiting for the rest of the machine to finish. The figure shows that small sample sizes behave badly in this regard, but that the oversampling scheme smoothes out the performance of the bucketing structure significantly. At an oversampling factor of 32, the ratio of the min to the mean is less than a factor of 1.5.

Figure 5.6b shows the ratio of the maximum bucket size to the mean. We will call this ratio the bucket expansion. The bucket expansion determines the efficiency of the algorithm both in terms of running time and processor utilization. Again, while the variance in subproblem sizes is rather large for small oversampling factors, at 32 samples per processor the average bucket expansion is less than 1.5.

Figure 5.7 shows the same two results for the larger simulated machine. The graphs show little change from the experiments with 32 processors. These results provide evidence that the performance of the sampling scheme should, for the most part, be independent of machine size.

The remaining question to address is whether the scheme will perform equally well with poor point distributions. To study this question, an additional set of trials was conducted using the same group of “bad” distributions from Chapter 2. For each distribution, 100 trials were run, with 32 simulated processors, a problem size of 20,000 sites and an oversampling factor of 32. Figure 5.8

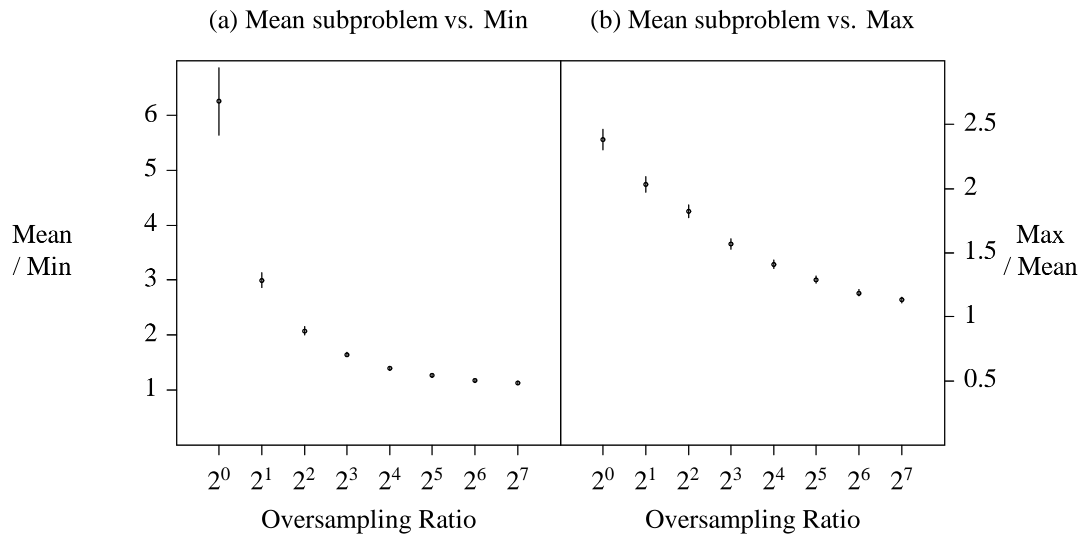


Figure 5.7: Load balancing for 128 processors.

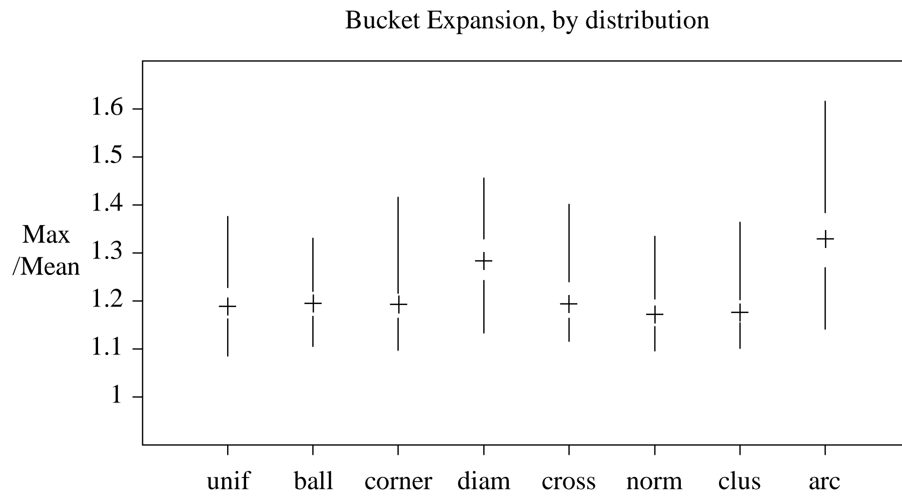


Figure 5.8: Bucket expansion with bad distributions for 32 processors.

summarizes these results with a box plot for each set of trials.

The bucket expansion incurred by Algorithm RDDT does show some sensitivity to the distribution of the input. In particular, notice that the behavior for the arc and diameter distributions is somewhat more erratic than the others. This is because in each of these cases, the subproblems associated with the sites "at infinity", which are added to the input as in the incremental algorithm, become somewhat large. Each of these cases models a particular worst case for Algorithm RDDT, where one isolated sample point has a disproportionately high degree in the Delaunay triangulation of the sample.

The above experiments indicate that the sampling scheme will be an effective way to manage load imbalances in Algorithm RDDT. The experiments provide evidence to support the following conjecture:

Conjecture 5.7. Let S be a set of sites that are identically and independently chosen from a distribution whose density function f satisfies $a \leq f \leq b$ for $a, b \geq 0$. Then the bucket expansion in Algorithm RDDT can be made less than 2 with high probability.

We will call distributions that satisfy these conditions "quasi-uniform," [BWY80, Dwy88]. The "bad" distributions that we used in our experiments all fit into this category, since even the most pathological of them still has a small amount of randomness. In addition, as we mentioned before, the oversampling scheme is directly related to the scheme used by Blelloch, et. al. in their sorting algorithm [BLM$^+$91], and is also similar to the polling technique introduced by Reif and Sen [RS92]. Theorem 5.1, the experiments, and the fact that the above algorithms all provide good load balanced with provably high probability are all strong evidence that a conjecture like Conjecture 5.7 can be proved.

Next, we move on to the total size of the subproblems created by the bucketing phase. To measure the expected total subproblem size, 30 trials were run at various problem sizes between 1K and 100K sites. For each trial, the bucketing procedure was executed, and the total size of the subproblems was tallied. Given the results above, these experiments were conducted using an oversampling factor of 32.

Figure 5.9a shows the results for the 32 processor machine, while Figure 5.9b summarizes the experiments for 128 processors. The graphs show that for small problem sizes, the total subproblem size is fairly small, while for large problems the total problem size approaches $6n$.

Finally, Figure 5.10 summarizes additional data from our earlier trials with various input distributions. For each distribution, the graph shows a boxplot summarizing distribution of the total problem size over the 100 trials. Again, these trials were run with 20,000 sites from each distribution, 32 simulated processors and an oversampling ratio of 32.

These experiments show that the total size of the subproblems comes close to $6n$ for a large variety of inputs. The actual work done by algorithm would be a small constant times the total subproblem size. Since the bucketing pass is rather expensive, and since the expected bucket expansion would be about 1.2 to 1.5, on 32 processors, this would mean speedups of 4 to 5, while on 128 processors, we might expect a speedup of 15 to 16.

In addition, neither the total subproblem size nor the bucket expansion seems to be overly sensitive to the distribution of the sites. Therefore, we would expect that the runtime of the algorithm would stay close to its expected value for many types of problems. However, it is apparent that this expected time has a constant that is high when compared to the sequential algorithms that we studied in Chapter 2.

The main problem with the algorithm is that it creates subproblems containing a large amount

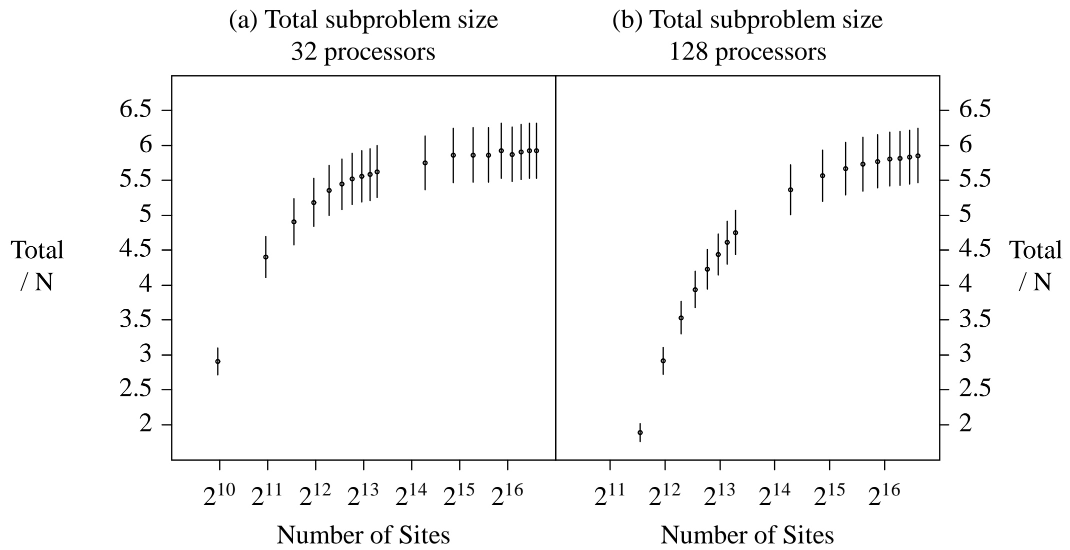


Figure 5.9: Total subproblem size for Algorithm RDDT.

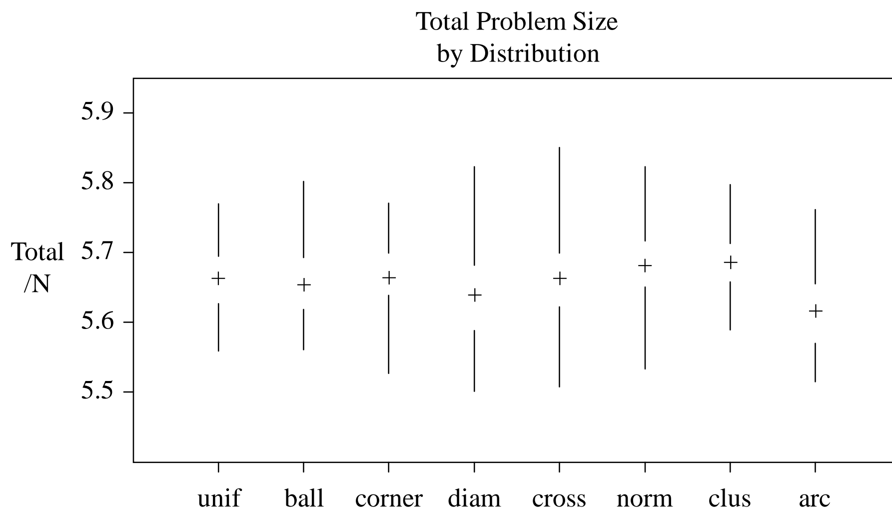


Figure 5.10: Total subproblem size by distribution for 32 processors.

of redundant information. The constant size blowup in total problem size is necessary to guarantee that all of the parallel subproblems can be solved independently. Perhaps more importantly, the algorithm would use a large amount of space to store and solve each subproblem. Since parallel machines tend to be used to solve larger problems, this extra space cost may become a concern even with the large memory sizes that are prevalent in modern machines.

One obvious way to reduce this overhead is to have Algorithm RDDT work in phases. After constructing the diagram of the sample, we can add the rest of the sites in small groups by running steps two and three of the algorithm several times. The problem with this scheme is that after each phase, the algorithm will need to run a fix-up procedure to filter out invalid Voronoi vertices. This filtering procedure will add both cost and implementation complexity to the final algorithm.

Another way to reduce the problem size blowup would be to only place a site s in the first subproblem found by `find-circles` and then move it to neighboring subproblems if there is not enough information in the initial one to correctly construct $V(s)$. We would expect that only a small fraction of all the sites would need to examine more than one subproblem, since for this to happen, a site s must lie close to the boundary of a Voronoi region in Vor(R). However, it is a relatively complex and costly task to determine, based on the solutions to the subproblems, exactly which sites need further processing, and exactly how to proceed once these sites have been found.

5.9 Summary

This chapter was the starting point of our study of parallel algorithms for Delaunay triangulation construction. After a survey of the literature, we introduced a framework for describing and analyzing randomized algorithms in Computational Geometry. In addition, we studied one possible parallel algorithm for constructing Delaunay triangulations that was based on randomized divide and conquer. The algorithm is a substantially simplified version of existing theoretical results. While the theoretical analysis of this algorithm was promising, experiments suggest that the constant factors in its runtime would be too high to make it effective.

Up to now,we have studied a number of different classes of sequential and parallel algorithms for closest point problems:

- "Classical" divide and conquer.
- Sweepline algorithms.
- Randomized incremental and divide-and-conquer algorithms.
- Incremental construction algorithms.
- Incremental search algorithms.

It is likely that the classic divide and conquer and sweepline schemes would be very difficult to implement in such a way as to take advantage of a large amount of parallelism. For machines with a modest number of processors, however, a straightforward parallelization of the divide and conquer algorithm could obtain reasonable performance on large problems. For P processors, such an algorithm would first divide and points into a tree of subproblems with P leaves. Then, in $\log P$ stages, the algorithm would merge P problems in $P/2$ subproblems using $P/2$ processors, then $P/2$ problems into $P/4$ using $P/4$ processors, and so on. As long as n is large enough, this algorithm will obtain high levels of efficiency.

This chapter has produced experimental evidence that randomized divide and conquer has a high runtime overhead when compared to current sequential algorithms. The high constants, and the relative complexity of the algorithm kept were the main reasons for not studying an actual implementation of the method. The method remains a strong possibility for applications where inputs are likely to be extremely unpredictable, however, since the experiments in this chapter have shown that the performance of the algorithm is generally uniform across a wide range of possible workloads.

In the next chapter, we will analyze and compare the other two possibilities: incremental construction and incremental search. The motivation for studying the latter is obvious. Concurrent local search proved to be a simple scheme with very good performance, and the incremental search algorithms for constructing Delaunay triangulations are straightforward extensions of that idea. Our interest in incremental construction is less obvious, since the incremental algorithm appears to be inherently sequential. In fact, we saw in Chapter 2 that with a random insertion order, incremental updates tend to effect only a small portion of the current diagram. Thus, it is likely that many such updates could be performed concurrently, assuming that we can devise a reasonable way to manage concurrent access to the data structure representing the diagram.

Chapter 6
Practical Parallel Algorithms for Delaunay Triangulations

Asymptotic analysis keeps the
student's head in the clouds,
while attention to
implementation details keeps his
feet on the ground.
—Jon L. Bentley

The shortest distance between
two points is under construction.
—Noelie Altito

The incremental algorithm has largely been ignored in theoretical studies of parallel methods for constructing the Delaunay triangulation. This is because there is no clear way to modify the algorithm to obtain the levels of concurrency needed for good results in the popular theoretical models of parallel computation. In practice though, obtaining such high levels of concurrency is usually not our main concern. Other issues, such as managing data movement, synchronization, and obtaining an abstract *work* bound that is as close as possible to good sequential time bounds are generally more important than being able to utilize n processors for a problem of size $O(n)$. As we saw in Chapter 4, simple data structures, a "coarse grained" parallel model, and careful engineering of implementations led to parallel programs with good performance.

In this setting, the incremental algorithm stands out as a natural extension to the concurrent local search method that we used in Chapter 4. Rather than looking for radically new methods that are asymptotically efficient, we will examine the practical issues involved in building concurrent versions of the simplest algorithms for constructing the Delaunay triangulation. These algorithms come in two flavors: incremental *construction* and incremental *search*. The former, like our sequential algorithm in Chapter 2 constructs the diagram by adding one site at a time. The latter, like the nearest-neighbor algorithm in Chapter 4 constructs the diagram one Delaunay triangle at a time using a variant on spiral search.

The remaining sections in this chapter will describe, analyze and compare the performance of these two algorithms. Finally, section Section 6.7 will sum up the chapter, and the lessons learned from implementing the algorithms.

6.1 Randomized Incremental Construction

Randomized incremental construction is yet another way to apply random sampling to the design of geometric algorithms. Randomized incremental algorithms, like the one in Chapter 2, and the

variant of quicksort, construct $\mathcal{F}_0(S)$ by adding the objects of S one at a time to a set R, and maintaining $\mathcal{F}_0(R)$. When a new object s is added to R, the algorithm adds new regions created by s, and removes regions in $\mathcal{F}_0(R)$ that conflict with s.

In their original framework, Clarkson and Shor present algorithms that use two data structures, one to represent the regions and one to represent the conflict relation. The conflict relation is represented as a bipartite graph, with edges between regions and objects. The main cost of the algorithms is updating this graph after each insertion. To do this correctly, and algorithm finds and deletes all the regions that the new object conflicts with and then creates a new set of regions to replace these. Finally, each edge (a, b) between an object a and a deleted region b must be replaced with an edge (a, b') from a to the appropriate new region b'.

For example, in the case of the Delaunay triangulation the conflict graph stores and edge (s, t) between a site s and any triangle t whose circumcircle contains s. When a new site q is inserted into the diagram, the algorithm must delete the triangles that q conflicts with and replace them with new triangle whose circumcircles do not contain q. In addition, if a site s conflicts with a deleted triangle t, the must examine each new triangle t' and add the edge (s, t') to the conflict graph if s falls in the circumcircle of t'.

To bound the expected cost of the incremental algorithm, it suffices to bound the expected cost of each insertion. Assume that we have chosen some subset R of S at random, with $|R| = r$, and we have built the Delaunay triangulation of the sites in R. Let f_r be the expected size of $\mathcal{F}_0(R)$ and let $F(r) = \max_{1 \leq z \leq r} f_z$. We know that for the Delaunay triangulation problem, $F(r) = O(r)$. The following lemma provides a bound on the expected cost of the last insertion. This is just a special case of Theorem 3.9 in Clarkson and Shor's paper [CS89].

Lemma 6.1. The expected cost of insertion step number $r + 1$ is $O(n/r)$.

Proof. Let s be the site inserted at step $r + 1$, and let some triangle $T \in \mathcal{F}(S)$ be given. Let I_T be a random variable that is 1 if $T \in \mathcal{F}_0(R)$ and s conflicts with T. Then I_T is 1 if and only if s is inserted after the three sites that are the vertices of the T, and before any other sites that conflict with T. Let $c(T)$ be the number of sites that conflict with T. The cost of the update step will be

$$\sum_{T \in \mathcal{F}(S)} c(T) I_T.$$

The probability that s is one of the $c(T)$ uninserted sites conflicting with T is $c(T)/(n - r)$. Therefore, the above sum is the same as

$$\sum_{T \in \mathcal{F}(S)} \frac{c(T)^2}{(n-r)} \text{Prob}\{F \in \mathcal{F}_0(R)\}.$$

Theorem 3.6 of Clarkson and Shor's paper shows that this is $O(F(r))(n - r)/r^2$. Since $F(r) = O(r)$, the whole expression is $O(n - r)/r$, which is $O(n/r)$. □

However, the incremental algorithm in Chapter 2 did not use a conflict graph to keep track of conflicts between sites and triangles. Rather, it used a combination of point location and circle testing to do the same job. In our implementation of the incremental algorithm in Chapter 2, the cost of each update was proportional to the number of triangles conflicting with the new site. Equivalently, the update time was proportional to the number of new triangles that the site created.

Let $R' = R \cup \{s\}$, then this cost is

$$\sum_{T \in \mathcal{F}(S)} \text{Prob}\{T \in \mathcal{F}_0(R') \text{ and } s \text{ is a vertex of } T\}.$$

The probability that s is a vertex of a given triangle T is $3/(r+1)$, so this sum is:

$$\sum_{T \in \mathcal{F}(S)} \frac{3}{r+1} \text{Prob}\{T \in \mathcal{F}_0(R')\} = 3F(r+1)/r = O(r)/r = O(1). \tag{I}$$

Thus, the expected number of updates that each new site incurs is a constant. We will use this result in the next section to design a an algorithm that constructs the Delaunay triangulation incrementally and concurrently. The intuition behind the algorithm is the same as we used before in Chapter 4. Since insertions into an existing diagram tend to cause a small number of local updates, many insertions will tend to be independent from each other. A parallel algorithm will be able to perform independent insertions concurrently. Therefore, as long as the current diagram is "large", a concurrent incremental algorithm will effective use of the parallelism available in the target machine.

6.2 Concurrent Incremental Construction

The concurrent incremental algorithm must be able to process independent insertions concurrently while maintaining a "consistent history" of intermediate diagrams.

To formalize what we mean by this, we'll borrow some terminology from the study of concurrent database processing [BHG87]. A *transaction* T_i is defined as a partial order $<_i$ where

1. $T_i \subset \{READ_i(x), WRITE_i(x)$ s.t. x is a data item $\} \cup \{ABORT, COMMIT\}$
2. $ABORT \in T_i$ if and only if $COMMIT \notin T_i$.
3. If t is $ABORT$ or $COMMIT$, then for all $p \in T_i$, $p <_i t$.
4. If $READ_i(x), WRITE_i(x) \in T_i$, then either $WRITE_i(x) <_i READ_i(x)$ or $READ_i(x) <_i WRITE_i(x)$.

Condition 1 specifies that a transaction is a sequence of machine instructions that read and write data from a central, shared database. Conditions 2 and 3 say that a transaction makes tentative updates to the database, and then either commits these updates permanently, or aborts and makes no permanent changes. Finally, condition 4 requires that reads and writes to the same data item be ordered.

We say that two operations p and q *collide* if they operate on the same data item. Normally, we could use the term *conflict* in this case, but since that term is already used in the context of randomized incremental construction, it would be confusing to overload its meaning. A *complete history* over a set of transactions $\{T_1, T_2, ..., T_n\}$ is a partial order $<_H$ with the following properties:

1. $H = \bigcup_{i=1}^{n} T_i$.
2. $<_H \supset \bigcup_{i=1}^{n} <_i$.
3. If $p, q \in H$ collide, then either $p <_H q$ or $q <_H p$.

Histories formalize the notion of defining an execution order on the operations of a set of transactions. In general, the operations in a set of transactions can be interleaved in an arbitrary fashion. However, some interleavings may not produce results that are consistent with sequential execution. A complete history is *serial* if for any two transactions T_i and T_j, either all the operations in T_i come before those in T_j or vice versa. The *completion* of a history H, denoted $C(H)$ is obtained from H by deleting all operations not belonging to transactions in H that have committed. A history is *serializable* if $C(H)$ is equivalent to some serial history, in the sense that it orders operations that collide in the same way. Serializable histories capture the intuitive notion of what should happen when a concurrent program produces an execution history that is consistent with a sequential program for the same algorithm.

To implement transactions that guarantee serializability, we use locks to synchronize access to shared data. It is well known that if a set of transactions utilizes *two-phase locking*, then any history over those transactions will be serializable. Two phase locking specifies that each transaction is divided into two phases, one where it obtains all of its locks, and one where it updates the database and releases all of its locks.

For the concurrent incremental algorithm, we would like to structure the insertion process as a set of transactions that update DT(R) in a serializable manner. Each insertion step will act on a centralized "database" representing the current triangulation.of R. As before, we will assume a shared memory programming model, with multiple threads, and three types of program statements:

`LOCK` Obtain a lock on an object s, or a region F, and its conflict list.

`UPDATE` Update a region F and the corresponding entries in the conflict graph.

`UNLOCK` Unlock a region F.

The code for `Insert-Site` is shown in Figure 6.1. The routine is split into three phases. The first phase looks much like the bucketing loop in the previous chapter. Assuming that we have inserted the sites in R, the routine searches DT(R) and marks the edges of each triangle that it tests for a conflict with a unique identifier i. This declares that this thread will attempt to obtain a lock on this edge. In addition, the first phase of the algorithm places all the edges examined by the search loop into a cache that will be used later to avoid repeated `in-circle` tests. Each entry in the cache is a pointer to an edge record, and a flag indicating whether the edge needs to be flipped. The algorithm puts all the edges that are examined into the edge cache, since it must check all of them later in the second phase of the locking scheme. This is necessary because the algorithm does not explicitly maintain a conflict graph, it uses the current state of the diagram to locate conflicts between new sites and existing triangles. Therefore, each thread must lock not only the edges it plans to flip, but also any edges it needs to read so that other updates do not invalidate any pre-computed circle tests.

The second phase of `Insert-Site` walks through the edge cache and checks to see if another thread with a lower priority number has marked any of the edges there. If so, this insertion must wait to try again in the next round. If all of the edges in the cache are marked with i, then it is safe to flip them all, which happens in the last phase of the algorithm. `Insert-Site`.

The Delaunay triangulation construction algorithm is then composed of some number of insertion *phases*. In each phase, each thread picks a site to insert and calls `Insert-Site`. We assume that the sites are randomly permuted before the the first insertion phase begins. If this isn't the case, permuting the input is a straightforward task.

```
Insert-Site (p) {
% The variable TID holds the thread ID of the current thread.
Phase 1:
  Locate an edge e, directly to the left of p,
    as in the sequential algorithm;
  initqueue(Q); initqueue(cache)  /* Init cache */
  do {  /* p conflicts with initial triangle */
    enqueue(Q, e); e = e.Lnext;
  } while (e.Org != first);
  for(;;) {   /* Look for conflicting circles until done */
    e = dequeue(Q); t = e.Oprev;
    enqueue(cache, e);                         /* put edge in CACHE */
    lock(e);                                   /* system lock */
    if (tid <= e.mark) {                       /* tag edge with ID */
      e.mark = tid; e.sym.mark = tid;          /* lock */
    }
    unlock(e);                                 /* system unlock */
    if (not CCW(t.Dest, e.Org, e.Dest) &&
        in-circle(e.Org,t.Dest,e.Dest, p)) {
      enqueue(Q, t); enqueue(Q, t.Lnext); set e.flip to true;
    }
    if empty(Q) break;
  }
Phase 2: /* Check to see if other threads are updating the same edges */
  Barrier;  private flag = 1;
  for each elt in cache {
    if (elt->e.mark != tid) flag = 0;
  }
Phase 3:
  if (flag) { /* Now make updates */
    do {  /* Connect new point */
      t = base.Sym; base = connect(e, t); e = base.Oprev;
    } while (e.Dest != first);
    for each edge e in cache {
      e.mark = e.sym.mark = infinity;              /* Unlock */
      if (e.flip) swapedge(e);
    }
    Barrier;
  } else abort;
}
```

Figure 6.1: The concurrent incremental algorithm.

By assigning priorities carefully, we can guarantee that the algorithm always makes progress, and that no thread tries repeatedly to insert a site without succeeding. The algorithm assigns priorities in the following manner: let S be the original set of n sites, and let $p = \{p_1, p_2, \ldots, p_n\}$ be a permutation on the integers between 1 and n. The permutation p defines the order in which the sequential randomized incremental would insert the sites. If a thread t is attempting to insert the site p_i, it is given the priority i.

This scheme, along with the locking protocol ensures that the algorithm will perform its insertions in an order which is consistent with what the original, sequential algorithm, assuming that each algorithm is given the same permutation. In addition, if we have P threads, the scheme ensures that no thread will have to wait for more than P phases in the worst case before its insertion succeeds.

Assuming a fixed number of processors, this algorithm has the same worst case $O(n^2)$ runtime as the standard incremental algorithm. Consider the situation where $n/2$ sites are placed along the x axis, and $n/2$ sites are placed in a circular arc that intersects the x-axis at $x = -1$. If we first insert all the sites in the arc, and then put in the other sites from right to left, each new insertion will cause $n/2$ edge flips. In addition, the sites along the x-axis will all need to be processed sequentially, since none of them update disjoint sets of existing edges. Therefore, in this case,the parallel algorithm is reduced to the worst case performance of the sequential algorithm.

Luckily, we can refer to many theoretical results that show that the average-case performance of this algorithm will be reasonable. We saw earlier that given a diagram DT(R) with $|R| = r$, if we insert site number $r + 1$ into R at random, then the expected number of updates the insertion would cause is a constant. Ideally, we would like to be able to prove that the probability of an insertion causing significantly more than the average number of edge flips is low.

The probabilistic analysis that we used before does not provide the exact result that we need here. The crux of the problem is bounding the sum that we derived in equation (I). If we knew that the probabilities in the sum were independent, then a simple tail bound would suffice to provide us with the needed result. But, this is not the case in general, and proving such a bound becomes much more difficult.

However, some useful bounds are available in the literature. First, the bound in condition (L) of Theorem 5.1 implies that with high probability, no site that has yet to be inserted conflicts with more than $O(\log r)$ existing triangles. In addition, Clarkson, et. al. show that the probability that the *total* number of updates performed randomized incremental algorithm is greater than k times the average is bounded by e^{-k} [CMS92]. The first bound gives us a rough ceiling on the maximum number of updates needed for a given insertion, but it isn't as exact as we would like. The second bound is tighter, but only holds in an amortized sense, not for any given insertion.

Outside the realm of randomized algorithms, Bern, et. al. show that under a restricted model of the input, the expected maximum degree of a vertex in the triangulation with r sites is $\Theta(\log r / \log \log r)$ [BEY91]. In this paper, the point set S is modelled as a unit-intensity Poisson process in the plane. The authors then restrict their attention to the portion of the Delaunay triangulation of such a set of points that lies within a square with side length $\sqrt{n}$. This model is used to avoid dealing with anomalies near the edges of the square, and thus does not correspond to point sets generated from a finite distribution. However, this result is good evidence that when the input to our algorithm is generated from the uniform distribution in the unit square, very few sites will cause expensive insertions to take place. This is because in the incremental algorithm, the cost of an insertion is proportional to the degree of the new site in the modified diagram.

Finally, we can supplement current theoretical thinking with experiments. For this purpose, we will depend on a simulation of the parallel algorithm that executes on a conventional workstation. Such a simulation can provide relatively precise answers for a wide range of questions that we are interested in, and for a large range of problem instances. Good experimental results, along the current set of similar, but not identical theoretical results listed above will all provide evidence that the result that we want is, in fact, true.

6.3 Simulations

To collect data for our experiment, a simulator was constructed from the program for the incremental algorithm used in Chapter 2. The simulator was designed to measure the abstract cost of Algorithm CRIC, not to try and estimate the runtime of the algorithm on any real system. Therefore, the simulator mimics multiple "threads" of control, executing synchronously, by using `for` loops. That is, whenever a statement in Algorithm CRIC is executed in parallel, by multiple threads, the simulator wraps the statement in a `for` loop, with one iteration per thread. Thus, the threads move forward through the program in lock step. Of course, this is a gross simplification of a real system, but since the simulation is not attempting to provide an accurate estimation of real performance, this is not a concern.

The simulator was instrumented to keep track of the maximum number of edge flips over a block of insertions. The size of the block is a parameter to the simulation. We will present results for blocks of 32 and 1024 insertions. The simulator runs the sequential algorithm, and keeps track of the maximum and average number of edge flips needed in each block of insertions. The 32 processor simulation ran on inputs of 10^3 sites, while the 1024 processor simulation ran on inputs of 10^5 sites. Each experiment used inputs from each of the nine input distributions used Chapter 2.

Figure Figure 6.2 shows the results of this experiment. For each run in each distribution, the box plot summarizes the ratio of the maximum number of edge flips in any given block to the mean in that same block. Thus, for the 32 processor case, the plot summarizes about 300 trials, while the 1024 processor plot summarizes 100 trials. In each case, we can see that the maximum number of edge flips is never more than a small constant times the mean. There does seem to be a small dependency on problem size and block size, since the values are uniformly higher in the 1024 processor plot. But, for practical problem sizes, this small factor does not appear to be significant.

These experiments give us confidence that for the range of problem sizes we are studying, (i.e. up to, say, 200K sites), and on machines with less than 1000 processors, the maximum number of edges flipped by any insertion will be a small constant times the average. If this is the case, and $|R| = r$, the probability that s conflicts with any given triangle in the current diagram is $O(c/r)$, for some constant $c > 0$. Now suppose we have P sites that we wish to insert into R. We will say that two insertions *collide* if their update sets intersect. Using the same analysis as in Section 4.3, we can see that the average number of insertions that collide with each other is $O(P^2/r)$. Therefore, as long as $r = O(P^2)$, the expected number of collisions in one insertion phase is a constant. Therefore, for n sites, there will be $O(n/P)$ phases. If we use the same point location structure as in Chapter 2, the expected runtime of each phase will be a constant, so the expected cost of processing insertions will be $O(n/P)$ time.

The remaining cost of the algorithm will be in managing the insertion queue, managing locks, memory system overhead, and the cost of barrier synchronization. All of these factors are machine dependent constants, and for fixed P, they will not affect the asymptotic runtime of the algorithm. However, they play a large role in determining the actual performance of an implementation, so we

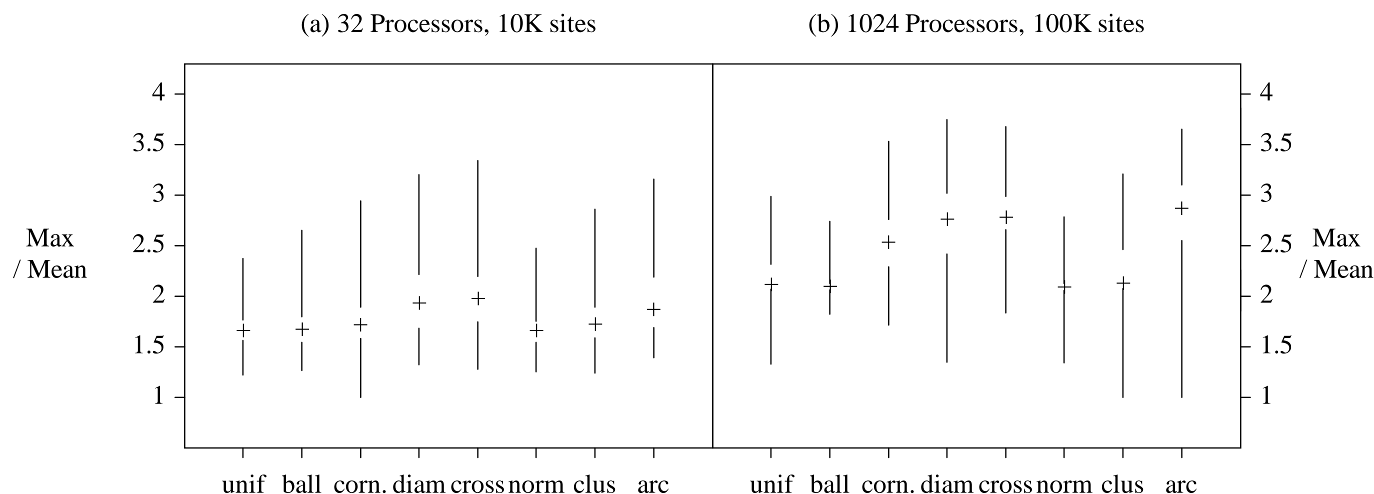


Figure 6.2: Load balancing in the concurrent incremental algorithm.

will deal with these issues at that stage of the discussion.

Now we look more closely at the the expected cost of processing insertions in Algorithm CRIC. We will examine two aspects of Algorithm CRIC: how much concurrency is actually available, and how much total work, in terms of circle tests and edge tests, does the algorithm perform.

The first question to consider is whether or not the algorithm actually achieves low contention and high concurrency. To examine this, the simulator was programmed to keep track of the number of insertions that succeeded in each iteration. These "concurrency profiles" were collected for a wide range of problem sizes between 1,024 and 131,072 sites, and for simulated machines with between 4 and 128 processors. The inputs were generated from the uniform distribution in the unit square. Each simulation was run with an initial sample sizes of between one sample and 32 samples per processor. We will examine the results for one sample per processor, since larger initial samples only increase the potential concurrency in the algorithm.

Figure 6.3 shows the results of this preliminary experiment. For each machine size between 4 and 128 processors, each graph shows one simulated run of the algorithm, on $1024P$ sites generated uniformly in the unit square. The "Time" axis of the plot represents the forward motion of the algorithm through each insertion phase. The simulation breaks the run of the algorithm up into groups of 20 insertion phases. For each phase, it records the number of threads in that phase that successfully inserted a site. This number reflects the number of threads doing useful work, and thus the processor utilization for that phase. After each group of 20 insertion phases, the simulation stops and computes the average number of active threads for that group. This value is shown on the graph. All of the profiles display the same general pattern. Once a sufficient number of sites have been inserted into the diagram, the algorithm can support as many active threads as is needed.

With these encouraging results in hand, we now take a more detailed look at the algorithm's performance. As before, the relevant measures are the total number of circle tests the algorithm performs and the number of edge tests the algorithm needs for point location.

For each statistic, ten trials were for problem sizes between 1,024 and 131,072 sites, and with simulated machine sizes of between 4 and 128 processors. As before, the inputs were generated from the uniform distribution in the unit square. The size of the initial sample was chosen to be one sample per processor. The graphs show box plots that summarize the distribution of the trials

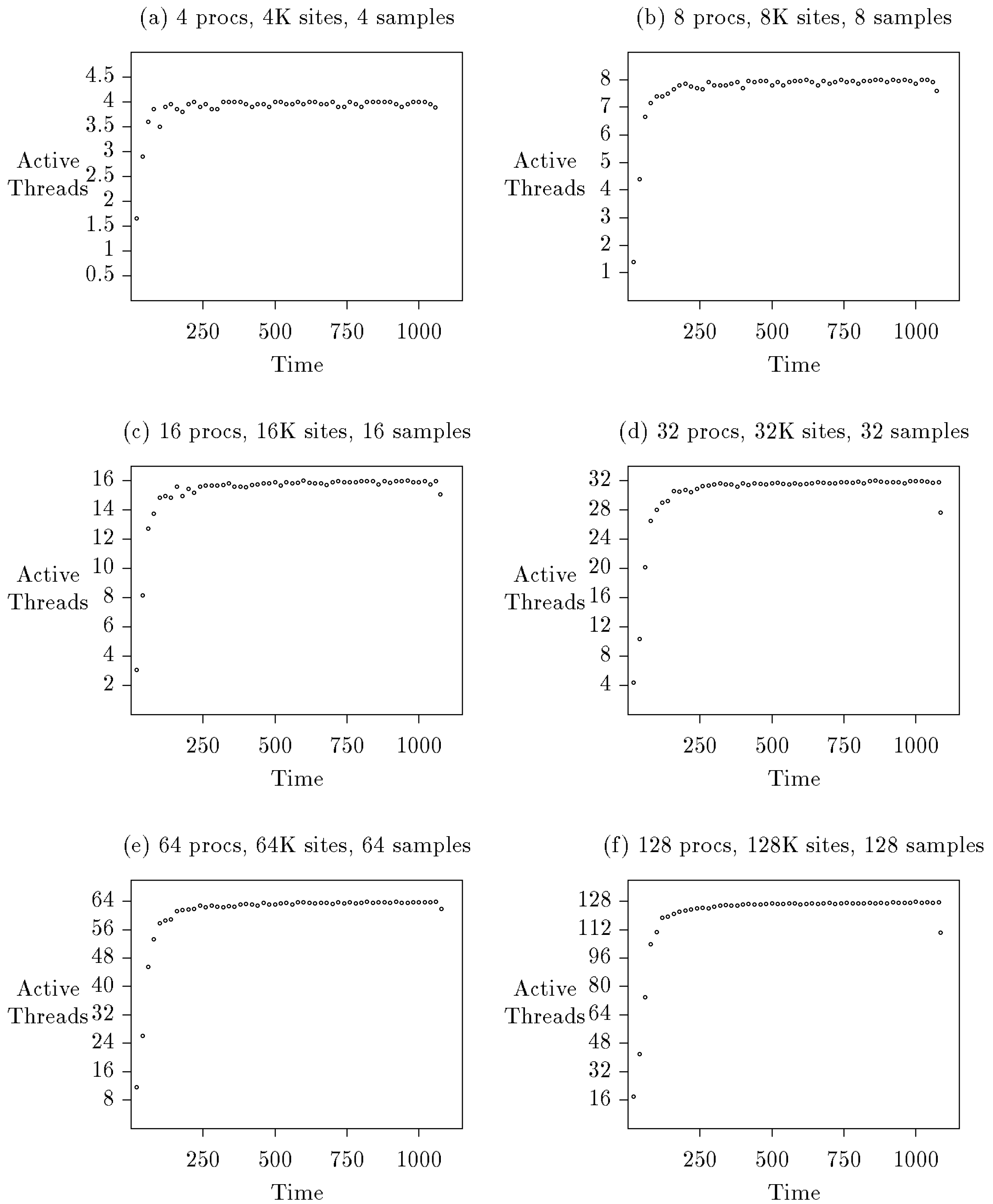


Figure 6.3: Concurrency Profiles for various machine and problem sizes.

at each problem size.

Figure 6.4 shows that for large problems, the parallel incremental algorithm does only slightly more circle tests, on average, as the serial algorithm did. For small problems, there is more contention, so more transactions must be aborted and retried. This causes extra circle tests to be performed. These results are consistent with our analysis, which said that the algorithm would perform well as long as $N >= P^2$.

Figure 6.5 shows the performance of the point location algorithm. These graphs show the same pattern as the results for circle testing. For large problem sizes, the performance of the algorithm is very close to that of the serial algorithm. For smaller problem sizes, the algorithm does somewhat worse, but as long as $N > P^2$, the expected cost of this phase of the algorithm, as measured by the simulator, is never more than a factor of two worse than the serial algorithm.

These results imply that in the absence of other system overhead, Algorithm CRIC will perform very well when compared to the serial incremental algorithm. Even though the simulations were idealized, they provide good evidence that the runtime constants in the algorithm are low enough to not effect on the performance of an implementation. Thus, this algorithm is a good candidate for a practical solution to the problem of constructing Delaunay triangulations in parallel.

6.4 The Implementation

Of course, any real implementation will need to contend with the overhead present in real systems. To examine these issues, we will discuss an implementation of Algorithm CRIC on the KSR-1. We choose the KSR-1, rather than the Cray for several reasons:

- The algorithm is not suited to a vector style of programming. We saw earlier that vectorizing spiral search, which is considerably simpler than Algorithm CRIC, was a very intricate process. So, while it is in principle possible to translate the algorithm to a vector style, it is much more natural, and convenient to take advantage of the multithreaded, shared memory programming model that the KSR provides.

- The KSR-1 provides more potential concurrency at the processor level, so we can better study how well the algorithm will scale to larger numbers of processors. The Cray has a high level of potential instruction level parallelism, but only for programs that vectorize. Otherwise, only 4 processors are available on the Cray Y-MP that we have access to.

- Finally, the irregular and dynamic nature of Algorithm CRIC will put maximum stress on the KSR-1 memory system. Therefore, the performance of the memory system on this test will provide insight as to the practicality of large, distributed memory, cache coherent multiprocessors.

The implementation of Algorithm CRIC follows Figure 6.1 fairly closely. Each thread repeatedly calls `Insert-Site`, retrying aborted insertions as needed. The sites to be inserted are kept in a centralized queue, out of which idle threads fetch new work to do. In addition, a shared version of the point location data structure is maintained and updated in parallel. Since the insertion phases of the algorithm occur synchronously, after each phase the algorithm checks to see if the bucket table needs to be expanded. If so, a master thread performs the expansion, and then all threads rebucket the sites that have been inserted.

In the implementation, our main concern is reducing the cost of data management and synchronization. The KSR parallel runtime libraries [KSR91] contain routines for barrier synchronization

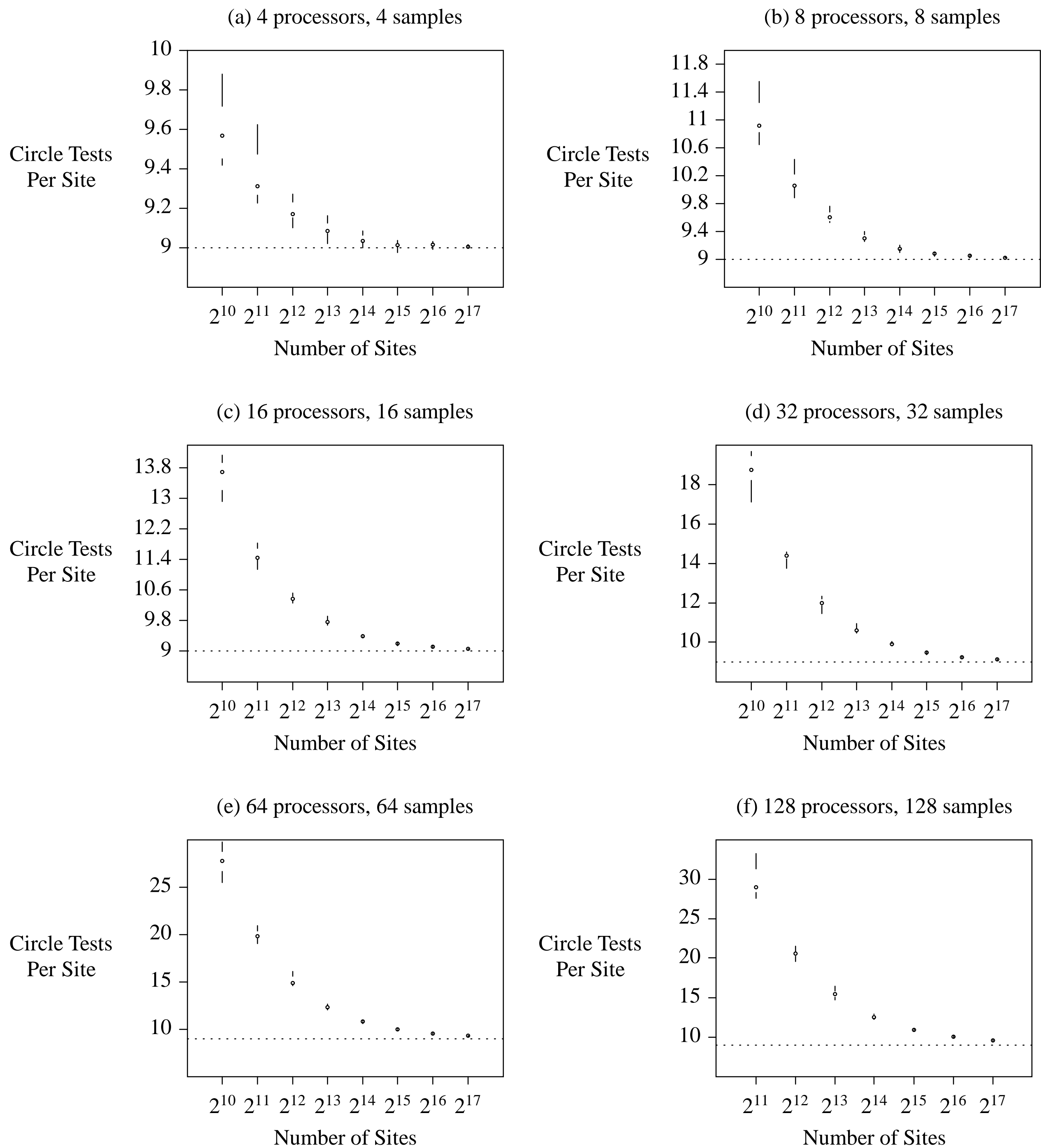


Figure 6.4: Circle tests per site for Algorithm CRIC.

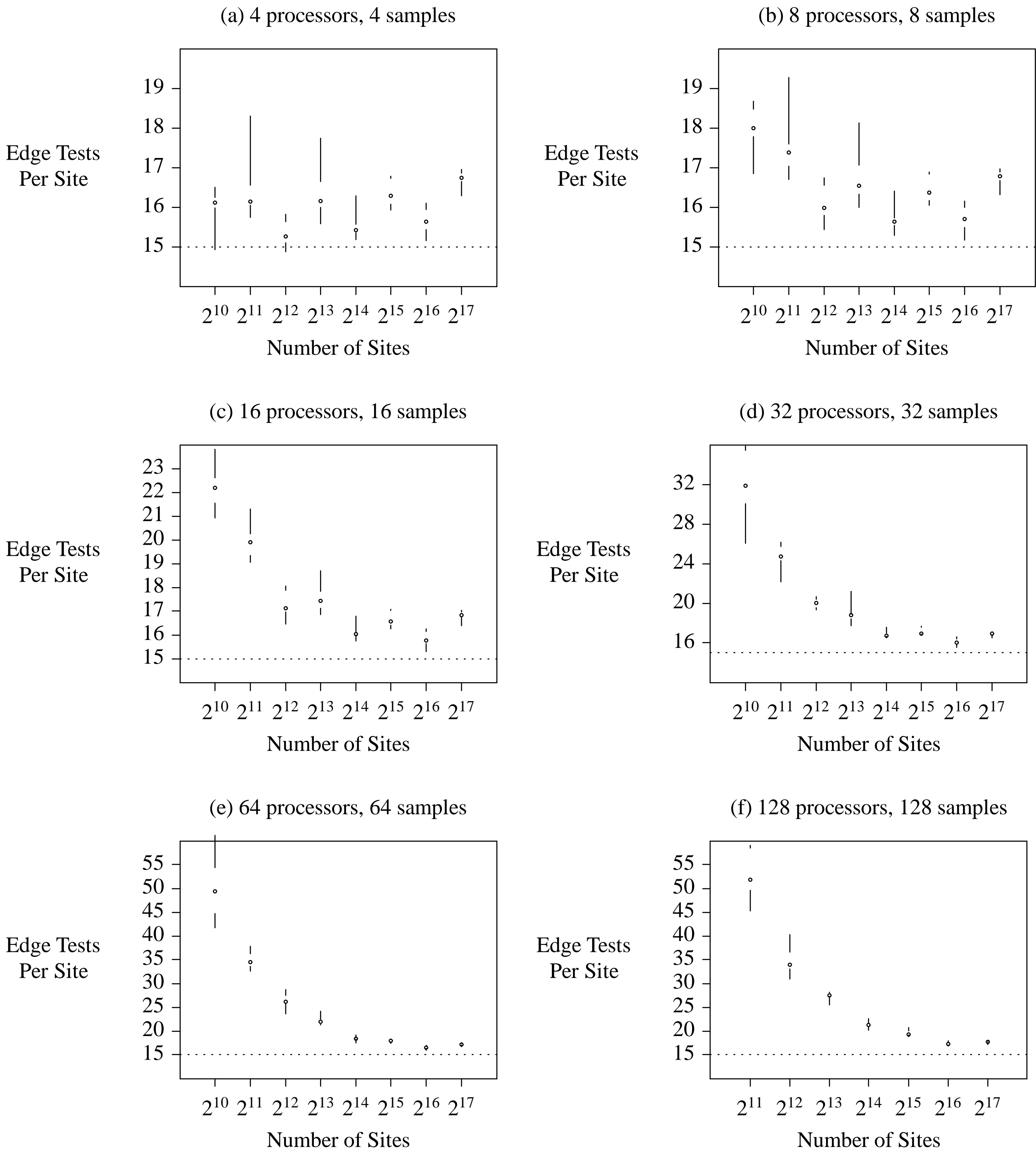


Figure 6.5: Edge tests per site for Algorithm CRIC.

and queue locks, but the overhead of using these routine is too high. Therefore, the simple locks needed in Figure 6.1 were implemented using the `_gspwt` instruction. This instruction attempts to obtain exclusive access to a cache block and halts the processor until the block is available. This mechanism is similar to the QOSB instruction described by Goodman [GVW89]. The `_rsp` instruction is used to release a lock.

Barriers were implemented using Mellor-Crummey and Scott's arrival tree barrier [MCS91]. This barrier algorithm is attractive because it uses less communication than standard counting barriers. This is an important consideration on the KSR, since communication is expensive compared to local processing. The routine was ported to the KSR from its original implementation for the Sequent Symmetry.

With these mechanisms in place, the main source of overhead becomes the cost of managing the distributed quad-edge data structure. Because the algorithm must update the diagram after each insertion phase, it incurs network traffic to fetch data from remote memories and to invalidate copies of cache blocks that may have been replicated. In Chapter 3 we saw programs that exhibit this pattern of memory access introduced a significant runtime overhead. The unstructured permutation benchmark from that chapter is representative of this sort of code. On that benchmark, eight processors running in parallel achieved little speedup over the sequential code, except for very large vectors. Given these results, we would expect that the direct cost of maintaining the shared data structure will limit the absolute speedup attainable by this algorithm.

However, non-uniform memory access times also produce an unexpected indirect effect on the program's performance. The unpredictable performance of the memory system introduces a large amount of variance into the relative runtimes of the threads in the parallel algorithm. As a result, the arrival times of threads at the barrier points in the algorithm becomes very unpredictable. The performance of the barrier algorithms is good when the all the threads reach them at roughly the same time, but when the arrival times are spread over a large range, fast threads spend a significant amount of time spinning inside the barriers.

In order to combat this effect, we will use two *ad-hoc* devices. First, we process insertions in batches to amortize the overhead of the barriers over a larger number of transactions. This idea was motivated by the fact that one of the main goals of data parallel language compilers for MIMD machines is to reduce the synchronization overhead of data parallel programs by combining individual instructions into larger blocks of code that could be run asynchronously by multiple threads[Cha91, QHJ88]. Similarly, Algorithm CRIC is defined as a set of synchronous phases, with one phase per site per processor. However, the concurrency profiles indicate that moderately sized problems can support a large amount of virtual concurrency. Therefore, it makes sense to take advantage of this, and simulate many insertions using one processor. The effect is to increase the grainsize of each insertion phase, and spread the cost of synchronization across many insertions.

The second device that we will use in the implementation is motivated by the "guided self-scheduling" (GSS) technique of Polychronopoulos and Kuck [PK87]. Guided self-scheduling was designed to facilitate the scheduling of do-loop iterations across multiple processors in a shared-memory environment. Information about iterations still needing service is stored in a central queue. When a thread becomes idle, it examines this queue and pulls off some fraction of the remaining work. Polychronopoulos and Kuck describe how threads can make a set of local, independent scheduling decisions in such a way as to minimize the spread of arrival times at the barrier marking the end of the loop.

Original:

```
for each thread
   for each site in current batch
     Insert-Site (site)
   grab new sites
end
```

Changes to:

```
place new batch of points in PQ
for each thread
  while (PQ is not empty)
    pull off a set of sites using GSS
    run phase 1 of Insert-Site
  end
  barrier;
end
place current batch of points in PQ
for each thread
  while (PQ is not empty)
    pull off a set of sites using GSS
    run phase 2 of Insert-Site
  end
  barrier;
end
place current batch of points in PQ
for each thread
  while (PQ is not empty)
    pull off a set of sites using GSS
    run phase 3 of Insert-Site
  end
  barrier
end
```

Figure 6.6: Interchanging loops to use GSS.

Restructuring Algorithm CRIC to use GSS would have required major surgery to the existing program. To make the scheme work, we would need to interchange the inner loops of the algorithm according to the scheme in Figure 6.6. The new code interchanges the order of the loops, bringing the internal phases of `Insert-Site` to the outer level of the loop nest and pushing the loop over each batch of sites into the inner level. This allows use to use GSS to schedule the phases of `Insert-Site` in a way that minimizes the overhead of the barriers. However, a change of this magnitude should not be undertaken without looking for a simpler method first.

Profiles of the program showed that the major source of overhead was the barrier after phase 1 of `Insert-Site`. Therefore, the algorithm was modified so that the first thread to reach this barrier would use a global flag to inform the other threads to finish up. When other threads detect that this flag is set, they break out of the phase 1 loop and enter the barrier as soon as possible.

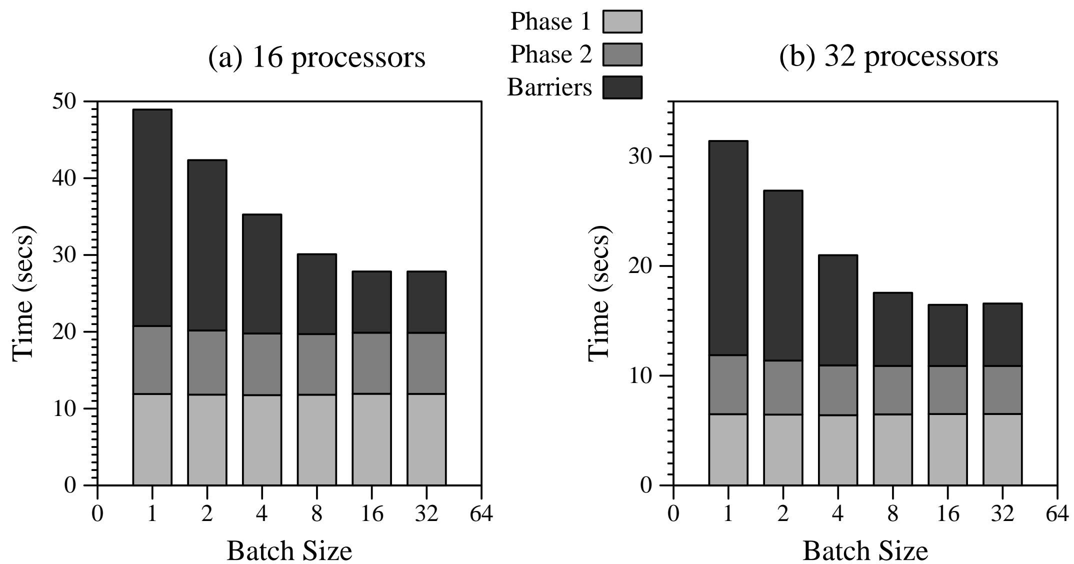


Figure 6.7: The effect of large batch sizes for 16 and 32 processors.

Figure 6.7 shows the combined effect of these two schemes on 16 and 32 processors. For these tests, batch size was varied from 1 site per iteration to 32 sites per iteration. At each batch size, five trials were run and the average value of each component of the total runtime was calculated. The graph summarizes the results of five trials at each batch size on a problem with 100K uniformly distributed sites. As the batchsize becomes larger, the contribution of barrier overhead to the total runtime of the program shrinks significantly, resulting in a larger improvement in overall performance.

While this simple scheduling trick works well for relatively small numbers of processors, the use of the global flag clearly limits the usefulness of this algorithm to relatively small machines. On large machines, the expense of the the broadcast operation will eventually outweigh any of the gains that better scheduling may have made. Making the implementation use more sophisticated scheduling techniques, such as GSS, and examining its scalability in that context is a promising avenue for future research.

Even with these extra refinements, the implementation of Algorithm CRIC behaves much like its simulation. Figures 6.8 and 6.9 show the cost of the algorithm in terms of circle and edge tests. For each measure, the algorithm was run on 4, 8, 16 and 30 processors, and on problem sizes between 1,000 and 200000 sites generated from the uniform distribution in the unit square.

As in the simulations, on small problem sizes, contention on the shared database causes many retries. But, as the problems become larger, the work that algorithm CRIC does is very close to that of the original incremental algorithm. The actual values are somewhat different from those in the simulation because the simulation did not mimic batched insertions or the simple-minded loop scheduling that the final implementation performed. Both of these mechanisms are machine dependent optimizations that are not appropriate to study in a simulation concerned with abstract costs. Even with these weaknesses, comparing the output from the simulations with the measurements from the implementation shows that the original experiments provided reasonably accurate upper bounds on the actual performance of the algorithm.

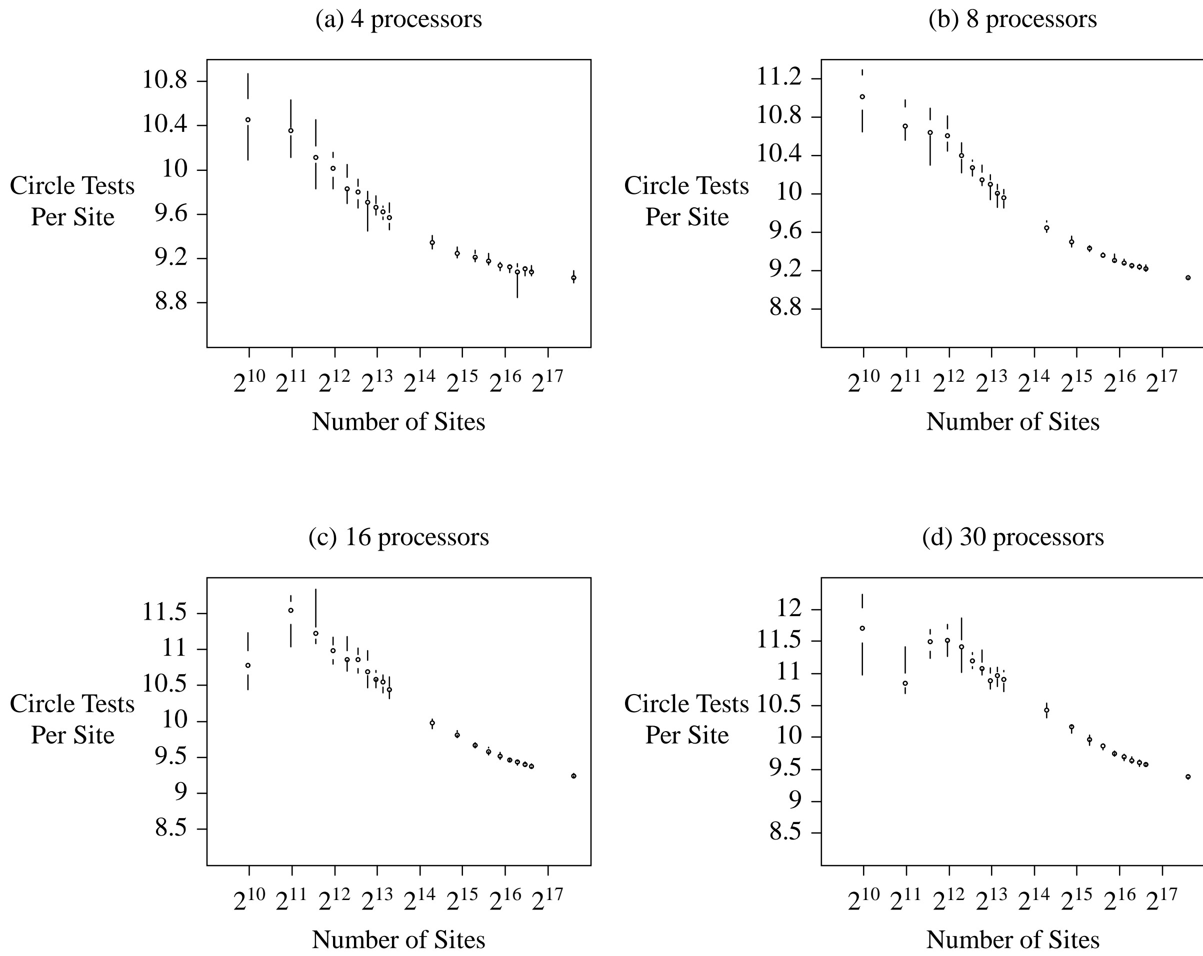


Figure 6.8: Circle tests per site for Algorithm CRIC.

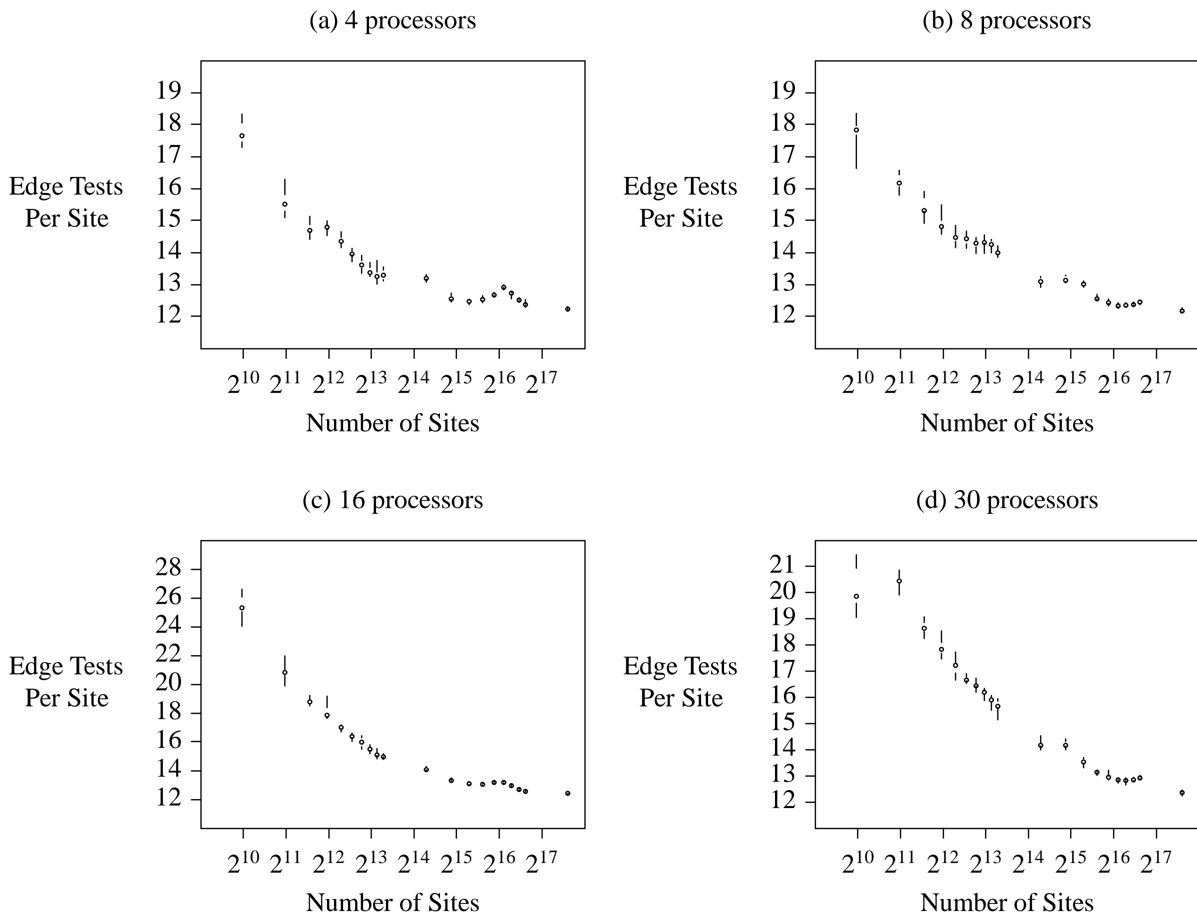


Figure 6.9: Edge tests per site for Algorithm CRIC.

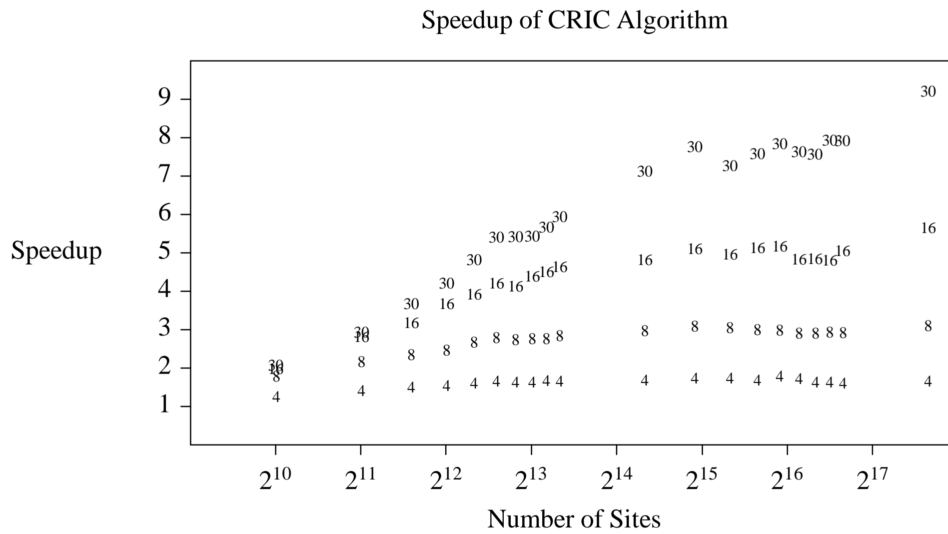


Figure 6.10: Speedup of Algorithm CRIC vs. the standard incremental algorithm on one KSR processor.

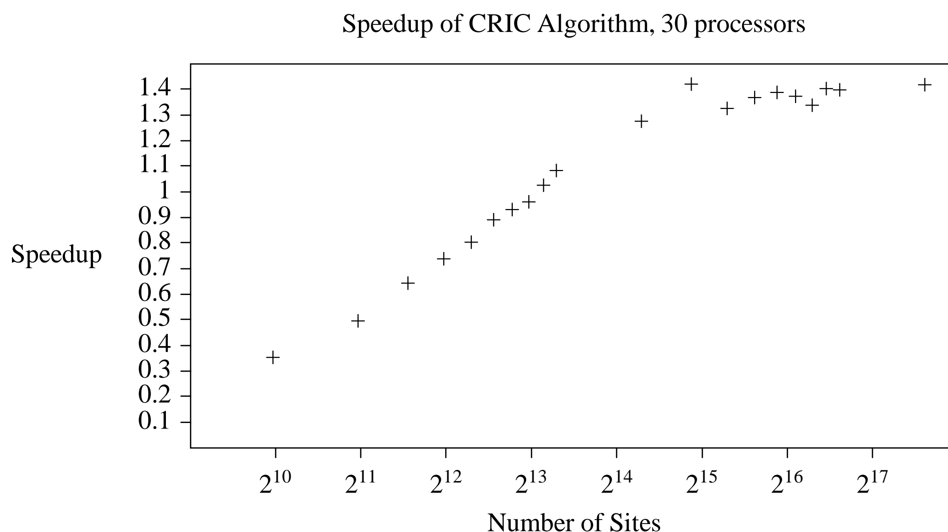


Figure 6.11: Speedup of Algorithm CRIC with 32 threads vs. the standard incremental algorithm on a Sparc 2.

The final performance measure of interest for Algorithm CRIC is, of course, speedup. The speedup plots show the average speedup over ten trials of the parallel algorithm for problem sizes between 1,000 and 200,000 sites. The inputs were again generated from the uniform distribution distribution in the unit square. Figure 6.10 compares the runtime of the concurrent algorithm with that of the *original* sequential algorithm running on one KSR processor. The graph indicates that while it takes four processors running algorithm CRIC to match the performance of the single, sequential node, the algorithm does achieve reasonable levels of relative speedup as we add processors: up to 9 with 30 threads. Two factors handicap the performance of algorithm CRIC when compared against the original incremental method. First, the KSR memory system is at its worst when dealing with unstructured read/write access patterns. We observed this behavior before in our study of the `permute` benchmark, presented in Chapter 3. Second, Algorithm CRIC makes two passes over the current diagram for each insertion, one checking for conflicts and one flipping edges. The original incremental algorithm only needs one pass to perform both tasks. While Algorithm CRIC caches the results of circle tests and point location, there is still extra work related to managing queues, checking locks and so on.

Both of these factors, and the fact that the KSR node processor is much slower than current off-the-shelf parts result in the speedup curve shown in Figure 6.11. Here, the runtime of Algorithm CRIC using 30 processors is compared with the runtime of the serial algorithm on a fast workstation. For this range of problem sizes, the workstation has the clear advantage in cost/performance. On the other hand, as problems become larger, the parallel algorithm will be able to take advantage of much more memory than the workstation. One indication of this effect is the small upward jump that the speedup curves in Figure 6.10 take for the largest problem sizes. In general though, the cost of maintaining all intermediate diagrams in a shared memory is too high for Algorithm CRIC to compete with current workstations.

On the face of it, the performance of this algorithm in absolute terms is somewhat disappointing. However, our experiments also showed that the algorithm makes effective use of multiple processors when they are available. The relative speedup exhibited by the algorithm on up to 30 processors is

typical of this, and other irregular applications with high communication requirements. Chatterjee's thesis contains many other benchmarks of this type [Cha91], though none are as complex as the construction of the Delaunay triangulation. Thus, the real advantage of the parallel algorithm is the fact that with many processors available, it will be able to solve problems that are too large to fit into the memory of current workstations. The multiple CPUs, and the memory management hardware available on machines like the KSR-1 make it possible to work with very large data sets efficiently.

6.5 Concurrent Incremental Search

Next, we turn to a parallel implementation of Algorithm IS from Chapter 2. We will call the new algorithm "Algorithm CIS", for Concurrent Incremental Search. Recall that this algorithm constructs the diagram on triangle at a time using a search process that is much like spiral search. In the parallel version of the algorithm, we perform many site searches in parallel rather than one at a time. In order to make this work, we need to construct concurrent versions of three data structures: the bucket grid, the edge dictionary and the edge queue. Chapter 4 showed how to construct the bucket grid in parallel using a simple locking protocol. The implementation of the edge dictionary is similar. The edge dictionary is also represented using a hash table with external chaining. Therefore, when inserting a new edge record into the dictionary, each thread first locks the bucket that the record falls in, adds the record to the bucket, and then unlocks the bucket. Unlike the bucket grid, the edge dictionary is modified as the algorithm progresses. Therefore, searching the dictionary, threads must also lock any buckets that they scan in search of an edge record. Since edges are mapped to buckets a uniform manner, contention on the bucket locks should not be a problem, so, as before, the edge dictionary will not be a major bottleneck in the performance of the parallel algorithm. Figure 6.12 shows pseudo-code that summarizes the implementation of the concurrent edge dictionary.

Handling the edge queue is a slightly more subtle problem. The edge queue is implemented as a circular queue embedded into an array. A straightforward way to make the queue work concurrently is just to have threads lock the head and the tail when queueing or dequeuing an edge. However, early runs of the program showed that contention was limiting the performance of the parallel algorithm. Therefore, the final implementation removes this contention by having each thread keep a local queue of edges, and threads only place edges onto the global queue when they run out of local space. In addition, to make sure that no thread terminates prematurely because its local queue empties out, we make sure that each thread puts at least a fixed fraction of its edges onto the global queue to keep everyone busy. The algorithm can then be tuned by adjusting the local queue size and the frequency of global inserts to optimize load balance and contention. For the experiments in this chapter, the local queue size was set to n/P, and no extra global inserts are performed. In practice, the premature termination of threads did not happen often enough to have a significant impact on the performance of the program. Figure 6.13 shows pseudocode for the queue handling functions in our implementation.

Except for code to create threads and synchronize between the bucketing and searching phases of the algorithm, the program for the concurrent incremental search algorithm is identical to its sequential counterpart.

6.6 More Experiments

To test the effectiveness of the parallel algorithm, I ran the standard set of experiments on an implementation for the KSR-1. I tested the program on point sets generated at random from the

```
// node(org,dest) == the edge record corresponding to (org,dest)
// link[bucket]   == pointer to first edge record in this chain.
//
Insert-edge (org, dest)
{
  bucket = hash(org,dest)

  lock(link[bucket])

  if node(org,dest) already exists in bucket
     unlock(link[bucket])
     return node(org,dest);

  newnode = construct node(org,dest);

  newnode.next = link[bucket];
  link[bucket] = newnode;

  unlock(link[bucket])
  return newnode;
}

Find-edge (org, dest)
{
  bucket = hash(org,dest)
  answer = nil

  lock(link[bucket])

  if node(org,dest) already exists in bucket
     answer = node(org,dest)

  unlock(link[bucket])
  return answer;
}
```

Figure 6.12: Code for a concurrent edge dictionary.

```
// qnodes    == array for global queue
// pqnodes   == array for local queue
// qhead     ==  head of global queue
// qtail     ==  tail of global queue
// pqhead    ==  head of local queue
// pqtail    ==  head of local queue

enq_edge (e)
{
  temp = (pqtail + 1 ) % num_pqnodes
  if (temp != pqhead || !should_insert_globally()) {
    pqtail = temp;
    put e into pqnodes[pqtail];
    return;
  }
  // local queue is full
  lock(qlock);
  qtail = (qtail + 1) % num_qnodes;
  if (qtail == qhead) {
    fprintf(stderr,"Queue overflowed, try again\n");
    exit(1);
  }
  put e into qnodes[qtail];
  unlock(qlock);
}
deq_edge()
{
  // check local queue
  if (pqhead != pqtail) {
    pqhead = (pqhead + 1) % num_pqnodes
    e = pqnodes[pqhead].e
    return e
  }
  // look in global queue
  lock(qlock)
  if (qhead == qtail) {
    unlock(qlock)
    return nil;
  }
  qhead = (qhead + 1) % num_qnodes
  e = qnodes[qhead]
  unlock(qlock)
  return e
}
```

Figure 6.13: Code for the concurrent edge queue.

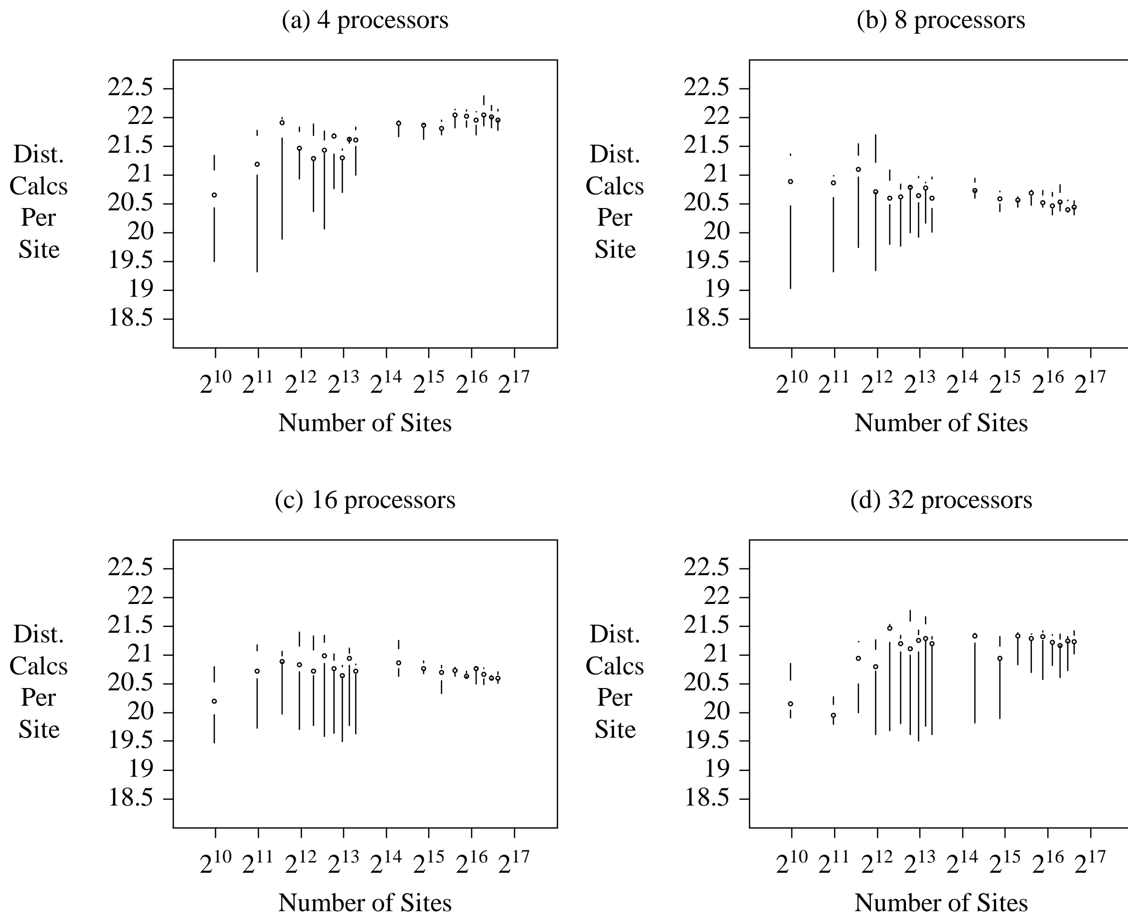


Figure 6.14: Distance calculations per site for the concurrent incremental search algorithm.

uniform distribution in the unit square. The program was tested using between four and thirty-two processors. Five trials were executed for point sizes between 1,000 and 100,000 sites. For comparison, the sequential version of the program was run on one KSR node, and on the Sparcstation used in the earlier tests. Finally, as usual, I used the sequential incremental algorithm running on a Sparc 2 as a baseline for an absolute performance comparison with a fast workstation. I did this mainly for consistency with my other comparisons. While it is true that Dwyer's algorithm is somewhat faster the difference on the Sparc is only about 10%.

Algorithm CIS performs almost exactly the same searches as the sequential one. One situation where it might perform a redundant search is if one thread is about to mark an edge as finished just after another thread has pulled it off the global event queue and called `Site-search`. However, the use of local queues makes this possibility relatively remote. Figures 6.14 and 6.15 provide experimental evidence for this fact. Recall that in Chapter 2 we saw that the performance of Algorithm IS was largely determined by the number of buckets it examined and the number of distance calculations it performed. The figures show box plots summarizing the behavior of the two metrics in this set of experiments. It is apparent from these trials that the parallel algorithm performs very little extra work when compared to the sequential code.

We saw in Chapter 2 that Algorithm IS was significantly slower than the incremental algorithm.

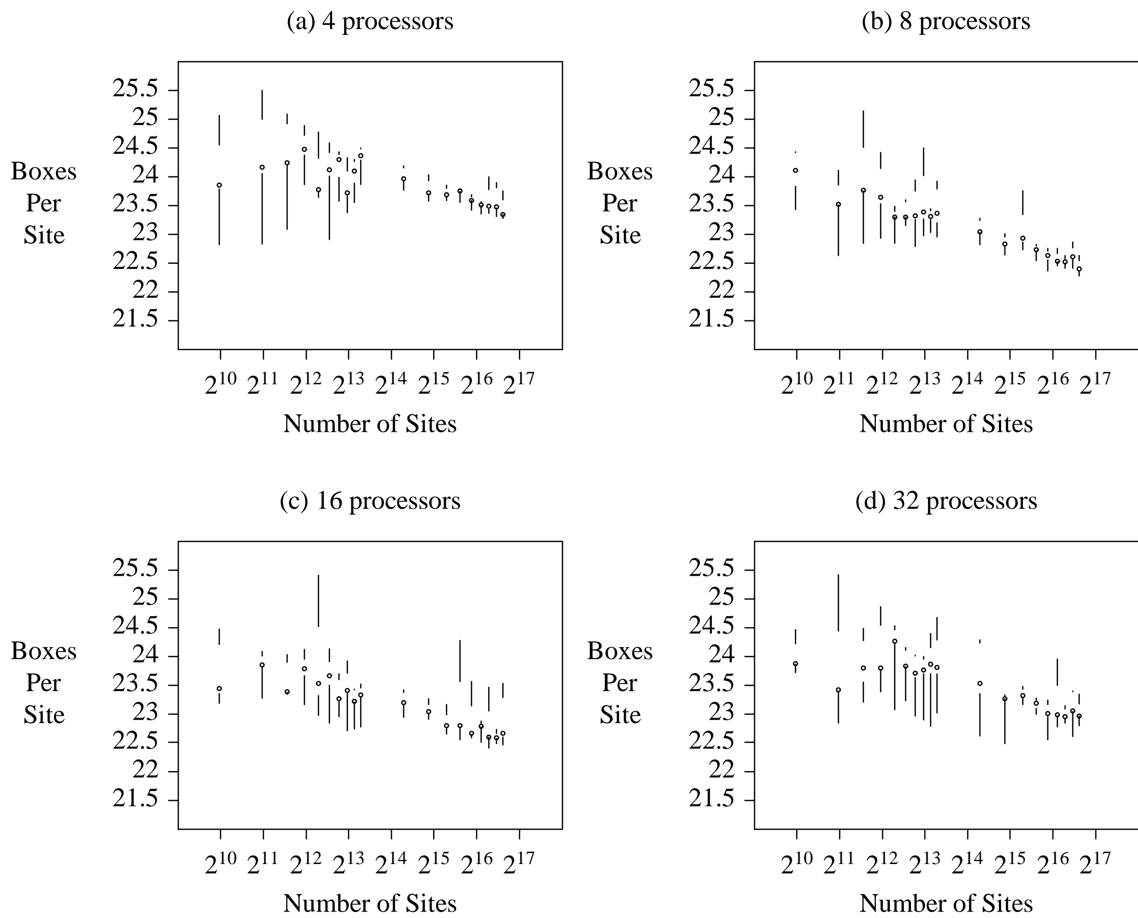


Figure 6.15: Buckets examined per site for the concurrent incremental search algorithm.

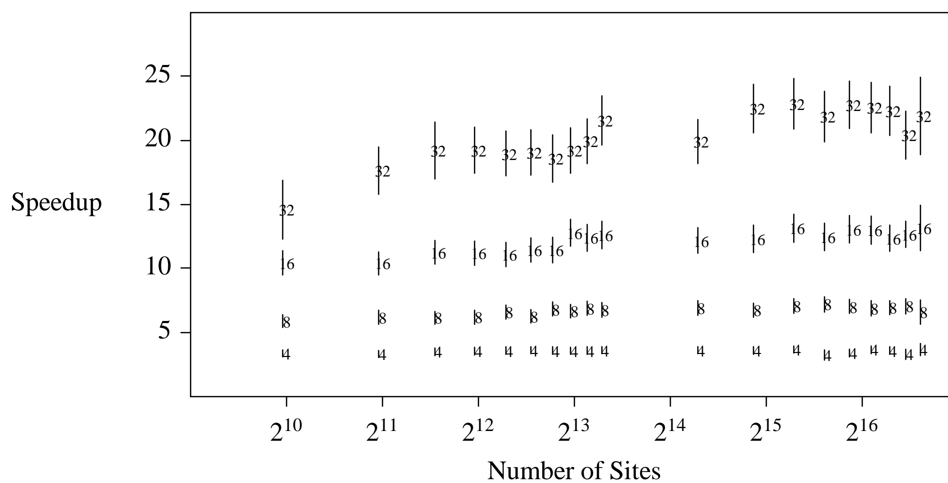


Figure 6.16: Speedup of Algorithm CIS relative to one KSR node.

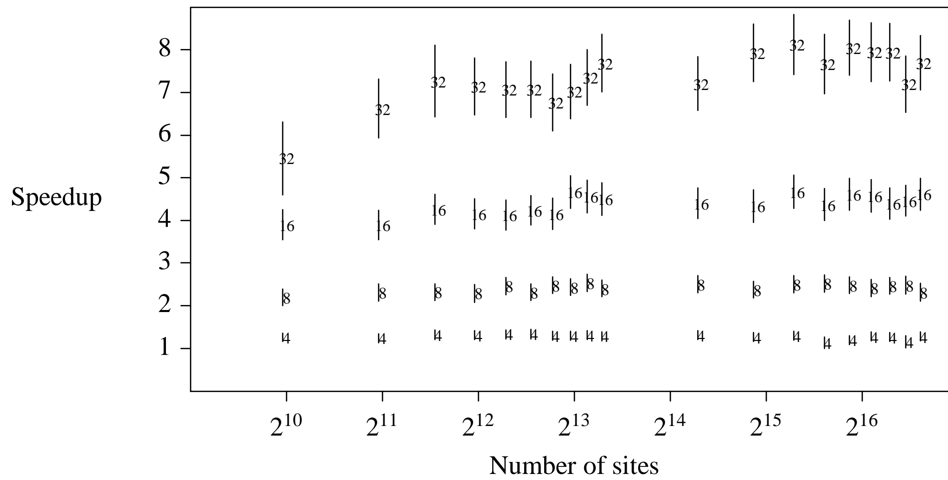


Figure 6.17: Speedup of Algorithm CIS relative to the SparcStation 2.

In its concurrent incarnation, the situation is reversed. The combination of simplicity, lack of global synchronization, and lack of contention on shared data structures allows Algorithm CIS to effectively utilize much more parallelism than Algorithm CRIC. The following figures summarize the speedup in three different ways. Each shows the average speedup obtained in the five trials along with the 90compares the runtime of Algorithm IS on one KSR processor to the runtime of Algorithm CIS on multiple KSR processors. The figure clearly shows that the Algorithm CIS is making much better use of extra processors than Algorithm CRIC did. Algorithm CIS behaves much like the parallel all-nearest-neighbors code did in Chapter 4. Since it mainly depends on searching the bucket grid in parallel rather than searching and updating it, the KSR memory system supports concurrent reads using the local cache of each processor. Algorithm CIS is not quite as scalable as the earlier all-nearest-neighbors algorithm because its searches cover a larger area, making the cache less effective, and because the algorithm needs to maintain the shared edge dictionary.

Figure 6.17 compares the Algorithm CIS running on the KSR to Algorithm IS running the Sparcstation 2. It is evident that the KSR has a significant performance advantage over the workstation in this case. This advantage remains even when we compare Algorithm CIS to the original incremental algorithm (see Figure 6.18). These two graphs provide solid evidence that within the limitations that we noted in Chapter 2, Algorithm CIS shows that it is possible to use parallelism effectively in the construction of the Delaunay triangulation.

Finally, Figure 6.19 shows a direct comparison between Algorithm CIS and the Algorithm CRIC on uniformly distributed sites. Both algorithms were run with 32 processors on problems of between 1K and 100K sites. The graph shows the ratio between the mean runtimes of each algorithm. Even though these results are easily derivable from the previous graphs, it helps to see them directly. The graphs show that for smaller problems, Algorithm CIS is more than five times faster then CRIC. For large problems, Algorithm CRIC improves, but Algorithm CIS is still more than twice as fast.

6.7 Summary

This chapter presented two practical algorithms for the parallel construction of the Delaunay triangulation with extensive experimental analysis of their performance. Each of the algorithm constructs

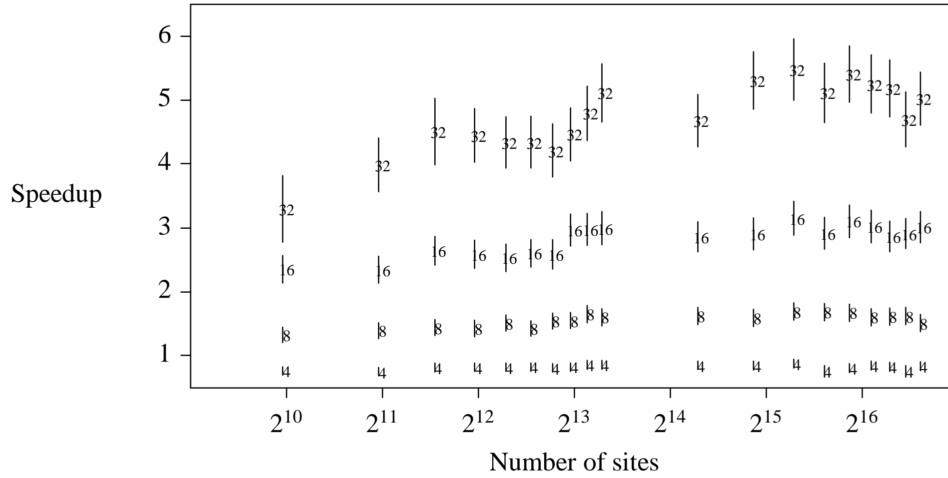


Figure 6.18: Speedup of Algorithm CIS relative to the sequential incremental algorithm running on a SparcStation 2.

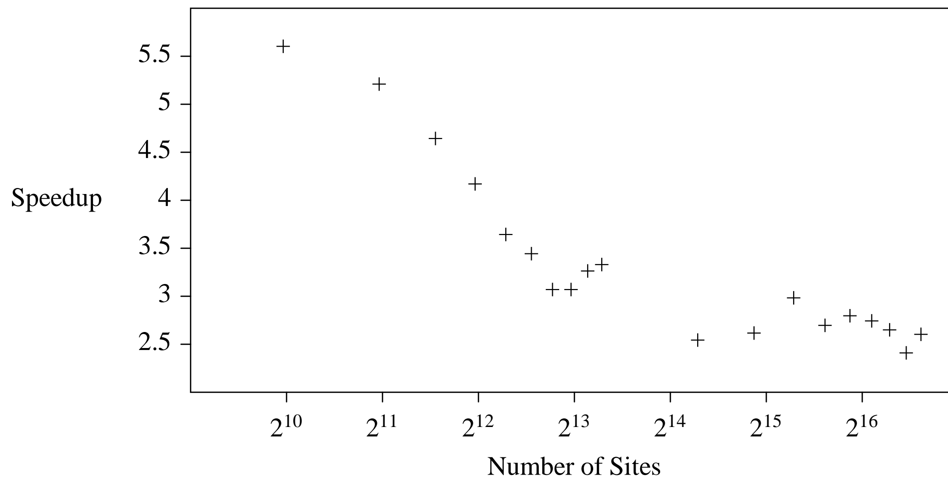


Figure 6.19: Speedup of Algorithm CIS relative to Algorithm CRIC on 32 procs.

the Delaunay triangulation in an concurrent, incremental fashion. Algorithm CRIC is a concurrent version of the randomized incremental algorithm that we studied in Chapter 2. Algorithm CIS is a concurrent version of the incremental search algorithm that we studied in Chapter 2. The experiments in this chapter, and in Chapter 5 led to some important observations about the practical behavior of parallel algorithms for Delaunay triangulation construction. We summarize these observations below.

- Using a random sample to divide and conquer produces independent subproblems that appear to be of roughly equal sizes. However, experiments with uniform inputs also showed that this technique adds a large multiplicative constant to the work bound of the algorithm.
- The incremental algorithms that we first discussed in Chapter 2 provide a practical basis for parallel algorithms for constructing the Delaunay triangulation.
- Simulations showed that on uniform inputs, the incremental construction algorithm has the potential to support a high level of concurrency, assuming that synchronization is not a bottleneck.
- The implementation of the concurrent incremental construction algorithm showed performance is poor due to high synchronization and memory management overhead.
- The incremental search algorithm avoids much of the overhead of the incremental construction algorithm, and thus exhibits much better performance in a parallel implementation.

Like the experiments in Chapter 2, our experiments with parallel algorithms also illustrate some principles that should be of general interest. We already covered some of these in Chapter 2, but we repeat them here in the new context of parallel algorithms design.

Keep it simple, stupid (KISS). After studying many elegant, but complicated methods for constructing the Delaunay triangulation in parallel, the simplest methods proved to be the most practical. The algorithms that I finally implemented were the oldest and simplest available.

Abstraction. Designing and analyzing parallel algorithms at a high level is possible both in theory and in practice. In Chapter 4, we were able to derive very accurate performance models from a combination of high level analysis and machine benchmarking. In this chapter, we used our earlier analysis of the sequential algorithms, and the high level primitives that they were based on, to obtain at least a qualitative feel for the performance of the final programs. This was more successful for Algorithm CIS than Algorithm CRIC since the core of Algorithm CIS is nearly identical to its sequential counterpart and therefore no new analysis was needed.

Concurrent Data Structures. Being able to utilize concurrent versions of common data structures makes the implementation of data parallel algorithms much easier. The algorithms in Chapter 4 and Algorithm CIS in this chapter use concurrent hash tables and queues. Algorithm CRIC uses a concurrent version of the quad-edge data structure. The availability of good, efficient, data structures allows the programmer to work at a high level of abstraction while maintaining efficiency. It also makes it much easier to develop programs and prove that they are correct.

Experimental Analysis. We have depended heavily on experiments to characterize the expected runtime of algorithms over a wide range of possible input distributions. This is crucial in those cases where a mathematical characterization of performance is not available, or is not exact enough for practical purposes.

Algorithm Animation. The animation of parallel algorithms is just as useful for debugging and visualization as it was for sequential algorithms.

Concurrent Read is Easier. Current parallel architectures seem to deal with read-shared data structures much more effectively than read/write shared data structures. This is reflected in both of our target architectures. The Cray Y-MP memory system only supports one write port per CPU, while each CPU has two read ports. The poor performance of Algorithm CRIC was in part due to the large amount of read/write sharing that it utilized. In contrast, the all-nearest-neighbors algorithm and Algorithm CIS both performed well on the KSR, and both utilize much more read sharing than write sharing.

Chapter 7
Summary and Future Work

This is the most difficult part: to summarize...
—C. A. Hoare

This thesis examines the construction of efficient programs for solving proximity problems. While the more novel results have been in the area of parallel algorithms, the real focus of the work is to combine theoretical and experimental results to create practical solutions. The contributions that this thesis makes are in three areas: algorithms, experimental analysis, and implementations.

7.1 Algorithms

This dissertation presented four case studies on sequential and parallel algorithms for solving the all-nearest-neighbors problem and for constructing Voronoi diagrams and Delaunay triangulations (see Chapter 1). The case studies presented four novel algorithms along with mathematical and experimental analysis of their performance:

- A faster version of the randomized incremental construction algorithm of Clarkson that combines randomization with buckets to obtain an average runtime of $O(n)$ steps. The algorithm is simple to implement and its performance is competitive with all other known methods (see Chapter 2).
- A parallel algorithm for the all-nearest-neighbors problem that is demonstrably efficient on both traditional vector processors and on newer cache-coherent multiprocessors. On each machine, the algorithm achieves good speedups over sequential code on current workstations (see Chapter 4).
- A concurrent incremental algorithm for constructing the Delaunay triangulation that uses a transactional abstraction to dynamically maintain the subdivision. The algorithm has the advantage of being more "on-line" and could be extended to support concurrent deletion as well. However, the overhead of maintaining a complicated shared data structure limits the algorithm's performance (see Chapter 6).
- A concurrent incremental search algorithm that constructs the Delaunay triangulation one valid triangle at a time using a series of nearest-neighbor-like queries. The algorithm uses a bucket grid to accelerate the search process and obtains reasonable speedups over fast sequential algorithms running on current generation workstations (see Chapter 6).

7.2 Models and Experimental Analysis

The case studies in this thesis illustrate the utility of using abstract models that are high level enough to be machine independent, but flexible enough to be tailored to particular applications. Such models have been used before in the study of parallel sorting, and the work presented here extends their usefulness to the realm of computational geometry. Our models are *data parallel* in the sense that they focus on performing concurrent operations on aggregate data structures rather then specifying large systems of independent processes. At the lowest level, the models are based on three classes of vector operations: elementwise arithmetic, routing and parallel prefix operations (scans). In Chapter 3 we made the following arguments in favor of this style of programming model:

- The models are portable in the sense that the low-level primitives are easy to implement efficiently on a wide variety of machines. Chapter 3 benchmarks example implementations of the primitives on the Cray Y-MP and the KSR-1.
- The models are simple but expressive. By providing powerful primitives, such as scans, the models allow programmers to express and analyze algorithms in high level terms.
- The models are realistic. The models do not lose sight of real-world costs since they can easily be parameterized to account for the cost of each of the primitives on a given architecture.
- The models are flexible. If the need arises, new, higher level primitives can be substituted for the low level ones as long as it is clear that the new primitives can be implemented efficiently.

In Chapter 4 we saw how to combine the expressiveness of the vector models with realistic machine parameters to obtain accurate estimates of the expected performance of a non-trivial algorithm.

The other case studies utilize a higher level version of the basic models, where the vector primitives are augmented with operations such as the `in-circle` test, orientation tests, and distance computations. By concentrating on high level, abstract operations, experiments can be used to make general observations about the behavior of an algorithm. Some observations that we have made are as follows:

- Incremental construction is a simple, effective method for building the Delaunay triangulation when combined with a dynamic, bucket-based data structure for accelerating point location (see Chapter 2).
- On uniform inputs, the incremental search algorithm appears to run in $O(n)$ expected time, but it is somewhat slower than the other algorithms due to higher constant factors (see Chapter 2).
- On sets of n uniformly distributed sites, the frontier in Fortune's algorithm has an expected size of $O(\sqrt{n})$ edges. The expected size of the event queue in the algorithm is also $O(\sqrt{n})$ (see Chapter 2).
- On uniformly distributed sites, circle events in Fortune's algorithm cluster near, and move with, the sweepline (see Chapter 2).
- Using a heap to represent the event queue in Fortune's algorithm improves its performance on large problems by a small amount. Most of the cost in maintaining a heap in Fortune's algorithm is incurred by `extract-min` (see Chapter 2).

- Dwyer's algorithm is the strongest overall for the range of problems that we tested (see Chapter 2).
- Using random samples for divide and conquer is a viable basis for a parallel algorithm to construct the Delaunay triangulation but suffers from large constant factors in its empirical runtime (see Chapter 5).
- In the randomized incremental construction algorithm, most insertions update independent areas of the current diagram, and thus can be processed concurrently (see Chapter 6).
- In parallel, the incremental search algorithm performs better, even though its abstract constants are higher, because it avoids high-overhead operations such as synchronization and fine-grained read/write sharing of data structures (see Chapter 6).

Finally, the data parallel models allowed us to implement our algorithm in terms of high-level data structures and appropriate primitive operations. The cost of using such data structures could be incorporated into the parameters of the model. The sequential algorithms in Chapter 2 used such structures as buckets, heaps, queues, dictionaries, and the quad-edge data structure. The parallel algorithms utilized concurrent versions of these same structures. This use of data abstraction greatly aided the implementation and analysis of both the sequential and parallel algorithms.

7.3 Implementations and Benchmarks

The final contribution that this dissertation makes is a suite of implementations of both sequential and parallel algorithms for closest point problems. The sequential suite collects code from several sources and adds the implementations the I have constructed myself. The parallel suite is made up of the four programs, two for all-nearest-neighbors and two for Delaunay triangulation construction that were used for the experiments in Chapters 4 and 6.

The parallel programs form the basis of a new set of benchmarks for studying the performance of parallel computers on programs that solve proximity problems. Such a benchmark suite will be useful to those evaluating the possible use of large machines on the many applications that use these algorithms. In addition, our algorithms exhibit behaviors that are typical of many irregular and dynamic programs. Therefore, they can be used in the further study of architectural and language support for such programs.

7.4 Future Work

The experience that we have gained from designing and constructing implementations of these algorithms leads to many questions for future consideration.

7.4.1 Algorithms.

In Chapter 2 and Chapter 4, we were able to match experiments results with known analytical results from the literature in computational geometry. While the experimental results in Chapters 5 and 6 are fairly conclusive, no formal proofs were presented to back them up, only heuristic arguments. Therefore, an interesting open problem is to match the mathematical analysis of these algorithms to their performance in practice.

In addition, many improvements and extensions to the algorithms in Chapter 6 are possible. The main bottleneck in Algorithm CRIC is memory management and synchronization overhead resulting from the algorithm's need to maintain a complicated shared data structure. In particular, the

algorithm displays a large amount of read/write sharing, which on the KSR means that it generates a large amount of invalidate traffic on the network. Newer shared memory machines, with faster CPUs and higher performance networks and cache protocols could improve the absolute performance of the algorithm substantially. But the experience we have gained with this algorithm suggests that some algorithmic refinements would have a larger impact. Modifying the algorithm to use a better scheduling scheme, like Guided Self-Scheduling could potentially reduce synchornization overhead in the insertion loops.

In addition, a more sophisticated locking protocol could remove the need for barriers in the inner loops of the algorithm. This would reduce the runtime of the algorithm by another 20% to 30%. However, it is unclear how to construct such a protocol in such a way as to avoid deadlock and starvation, especially because the algorithm accesses the quad-edge structure in an unpredictable fashion, making deadlock avoidance difficult.

Recently, Herlihy and Moss [HM92] have proposed an extension to standard cache coherency schemes, called *transactional memory* that would, to a large extent, alleviate these problems. Transactional memory allows the programmer to construct custom, multi-word read-modify-write operations that act atomically on shared data structures. This mechanism would allow us to define `Insert-Object` transactionally without the use of barriers or complicated locking schemes. A restructured version of Algorithm CRIC that takes advantage of this mechanism is currently under construction on a simulator for Transactional Memory.

The incremental search algorithm could also be improved by incorporating a better data structure to support site searches. Bentley has proposed some novel data structures for supporting these types of queries in two and higher dimensions [Ben90]. It would be interesting to explore how to use these structures in a concurrent environment. In addition, using techniques from Chapter 4, the incremental search algorithm could be vectorized, resulting in even better performance on machines like the Cray and the new CM-5 that have hardware to take advantage of vector code. In particular, I am investigating the implementation of this and other algorithms in Blelloch's NESL language [Ble93], which will allow a vector style implementation to run on many platforms.

Finally, it should be possible to apply the methods used in these two algorithms to other problems and applications. The randomized incremental framework can be applied to a wide variety of problems. In particular, I would like to explore segment intersection and higher dimensional convex hull and Delaunay triangulation construction algorithms. These problems have applications that are potentially large enough to support the use of a multiprocessor. In addition, exploring algorithms for the many applications of closest-point problems, such as the TSP problem, minimum spanning trees, clustering, and so on, would lead to many more insights about the practical nature of parallel algorithms.

7.4.2 Programming Parallel Algorithms.

Parallel algorithms need to be easier to program. Each algorithm in this thesis was painstakingly built by hand. This limited the scope of our investigation to a small number of algorithms for a restricted class of problems. If more comprehensive studies of practical parallel algorithms are to be feasible, a portable, high level library implementing basic parallel primitives must be built. The current trend is toward machines that are even more difficult to program than current architectures, making such a library even more necessary.

Data parallelism is about data abstraction more than anything else. By packaging a large set of useful, data-oriented parallel primitives together and providing efficient access to them, data

parallel languages provide users with a high level of abstraction at a modest cost. Currently, the data parallel style is associated with aggregate operations on collection-oriented structures such as vectors. I would like to study the mechanisms necessary to further expand the data abstractions available to the programmer of parallel architectures. In particular, developing a transactional style of programming, similar to the one used in Chapter 6 would be a promising avenue for further work. This style has two critical characteristics:

Shared memory Programming in a single address space is easier than programming in multiple address spaces and explicitly placing and moving data. For complicated data structures with irregular and dynamic reference patterns, shared memory models provide a large gain in programmer efficiency in exchange for a modest loss in runtime efficiency.

Atomic Operations Herlihy and Moss [HM92] show that simple extensions to current multiprocessor memory and cache management schemes can provide the programmer with an extendible set of atomic operations that can be used to build transactional data abstractions without complex locking protocols.

In order to make these idea usable in practice, much needs to be accomplished. Machines must be built with suitable cache-management primitives. Notations for specifying concurrency and concurrent operations on data structures must be designed and incorporated into languages. Languages based on such notations must be translated by new compilers into a runtime environment that is suitable for the task. Algorithms that utilize the transactional style must be developed to test and tune the primitives offered by such a system.

Finally, the high level tools that are available on workstations must migrate to multiprocessor machines. High level programming languages, program instrumentation, profilers, data analysis tools, algorithm animators, and debugging tools are all needed on the new machines. This thesis has concentrated on the use of many of these kinds of tools and an ad-hoc fashion usually off-line from the actual target machine. The future challenge is in integrating these tools together into a coherent, efficient and systematic programming environment for multiprocessors.

7.4.3 Simulation and Performance Analysis.

In this thesis, simulations played a large role in the design and evaluation of algorithms. In addition, execution-driven machine simulation is the main vehicle for studying questions about parallel architecture. These simulation systems have the potential to allow algorithms designers to easily observe the behavior of their programs in a relatively realistic setting. More importantly, simulators would free designers from needing to develop many machine dependent versions of an algorithms in order to evaluate its performance across a wide range of architectures. Also, the use of parallel architectures to run simulations would allow large applications to be prototyped and tested in a simulated environment, thus giving more realistic results.

Having new applications available on simulators would also help the architecture and systems community. The study of novel parallel algorithms can have a large impact on the design of future machines, since many architectural decisions are made by studying the behavior of standard suites of benchmarks. These suites need to be expanded to include a more diverse set of algorithms, especially those that, like Algorithms CRIC and CIS, exhibit irregular and dynamic patterns of read/write sharing. Ideally, one would like to accurately characterize the bottlenecks of more irregular programs, and then design and evaluate new mechanisms for relieving these bottlenecks.

With such simulation systems available, many other problems become possible areas of

research. These include managing the data produced by large experiments, querying such a large databases for useful information, visualizing performance trends, or dynamic and time-dependent program behavior, and feeding performance information back into abstract machine models to accurately predict the performance of related algorithms or machines.

Simulations could also be used in concert with high level performance models and compilers to help programmers tune applications for better performance. A combination of static flow analysis and dynamic information collected at runtime or through a simulation could be used to, for example, tune data placement to increase locality or restructure programs to decrease synchronization costs. Profiler/compiler feedback loops have been used before to restructure sequential program in various ways.

References

[ACF90] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. *ACM Symposium On Theory Of Computing*, pages 600–608, 1990.

[ACG+88] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

[ACG89] M. J. Attallah, R. Cole, and M. T. Goodrich. Cascading divide and conquer: A technique for designing parallel algorithms. *SIAM Journal On Computing*, 18(3):499–532, 1989.

[ACS89] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in pram computations. *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–22, 1989.

[ACS90] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of prams. *Theoretical Computer Science*, 17:3–28, 1990.

[AGSS89] A. Aggarwal, L. Guibas, J. Saxe, and P. W. Shor. A linear time algorithm for computing the voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4:591–604, 1989.

[AJF91] A. Aggarwal, J. Jubiatowicz, and C. Fields. Limitless directories: A scalable cache coherence scheme. In *Proceedings of the Fourth ASPLOS*, pages 224–234, 1991.

[Aur91] F. Aurenhammer. Voronoi diagrams-a survey of a fundemantal geometric data structure. *Computing Surveys*, 23(5):345–405, 1991.

[Bar93] C. Brad Barber. *Computational geometry with imprecise data and arithmetic*. PhD thesis, Princeton, 1993.

[Ben80] J. L. Bentley. Multi-dimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[Ben82] J. L. Bentley. *Writing Efficient Programs*. Prentice-Hall, Inc., 1982.

[Ben89] J. L. Bentley. Experiments on travel salesman heuristics. *ACM-SIAM Symposium on Discrete Algorithms*, pages 91–99, 1989.

[Ben90] J. L. Bentley. K-d trees for semidynamic point sets. *Proc. 6th Annual ACM Symposium on Computational Geometry*, pages 187–197, 1990.

[Ben91] J. L. Bentley. Tools for experiments on algorithms. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, chapter 5. ACM Press, 1991.

[BEY91] M. Bern, D. Eppstein, and F. Yao. The expected extremes in a delaunay triangulation. *International Journal on Computational Geometry and Applications*, 1(1):79–91, 1991.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Ble89] G. Blelloch. Scans as primitive parallel operators. *IEEE Transactions On Computers*, 38(11):1526–1538, 1989.

[Ble90] G. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Ble93] G. E. Blelloch. Nesl: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, CMU, 1993.

[BLM+91] G. Blelloch, C. Leiserson, B. Maggs, G. Plaxton, S. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine cm-2. *Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[BWY80] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6(4):563–580, 1980.

[CBF91] S. Chatterjee, G. Blelloch, and A. L. Fisher. Size and access inference for data parallel programs. Technical report, CMU, 1991.

[CBZ90] S. Chatterjee, G. Blelloch, and M. Zagha. Scan primitives for vector computers. *Proceedings of Supercomputing '90*, pages 666–675, 1990.

[CCL88] M. Chen, Y. Choo, and J. Li. Compilig parallel programs by optimizing performance. *The Journal of Supercomputing*, 2:171–207, 1988.

[CF89] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a numa multiprocessor. exerience with platinum. In *Proceedings of the Twelfth SOSP*, pages 32–43, 1989.

[CGO90] R. Cole, M.T. Goodrich, and C. O'Dunlaing. Merging free trees in parallel for efficient voronoi diagram construction. *LNCS 443*, pages 432–445, 1990.

[Cha91] S. Chatterjee. *Compiling Data-parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, CMU, 1991.

[Che52] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[Cho80] A. Chow. *Parallel Algorithms for Computational Geometry*. PhD thesis, Univ. of Illinois, 1980.

[CKP+92] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. Technical Report UCB/CDS 92-713, U.C. Berkeley, 1992.

[Cla84] K. L. Clarkson. *Algorithms for Closest Point Problems*. PhD thesis, Stanford Univeristy, 1984.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press/McGraw Hill, 1990.

[CMS92] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental construction. *Annual Symposium on Theoretical Aspects of Computer Science*, 1992.

[CMSS92] E. Cohen, R. Miller, E. M. Sarraf, and Q. F. Stout. Efficient comvexity and domination algorithms for fine- and medium-grain hypercube computers. *Algorithmica*, 7:51–75, 1992.

[Cor92] T. H. Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992. Available as Technical Report MIT/LCS/TR-559.

[CS89] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, ii. *Discrete and Computational Geometry*, 4:387–421, 1989.

[CZ89] R. Cole and Ofer Zajicek. The apram: Incorporating asynchrony into the pram model. *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[Deh89] F. Dehne. Computing digitized voronoi diagrams on a systolic screen and applications to clustering. *Optimal Algorithms*, pages 14–24, 1989.

[DMT90] O. Devillers, S. Meider, and M. Tellaud. Fully dynamic delaunay triangulation in logarithmic expected time per operation. Technical Report 1349, INRIA, 1990.

[DS90] D. Dobkin and D. Silver. Applied computational geometry: Towards robust solutions of basic problems. *Journal of Computer and System Sciences*, 40:70–87, 1990.

[Dwy87] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[Dwy88] R. A. Dwyer. *Average-case Analysis of Algorithms for Convex Hulls and Voronoi Diagrams*. PhD thesis, CMU, 1988.

[Dwy91] R. A. Dwyer. Higher-dimensional voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6:343–367, 1991.

[EK89] S. Eggers and R. Katz. Evaluating the performance of four snooping cache-coherency protocols. In *Proceedings of the Sixteenth ISCA*, pages 2–15, 1989.

[ES91] H. Edelsbrunner and W. Shi. An $o(nlogsup2h)$ time algorithm for the three-dimensional convex hull problem. *SIAM Journal on Computing*, 20(2):259–269, 1991.

[FJL+88] G. C. Fox, M. A Johnson, G. A. Lyzgenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[For87] S. Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[For89] S. Fortune. Stable maintenance of point-set triangulations in two dimensions. *IEEE Symposium on Foundations of Computer Science*, pages 494–499, 1989.

[For92] S. Fortune. Numerical stability of algorithms for delaunay triangulations and voronoi diagrams. *Annual ACM Symposium on Computational Geometry*, 1992.

[GG91] M. Ghouse and M. Goodrich. In-place techniques for parallel convex hull algorithms. *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 192–201, 1991.

[Gib89] P. Gibbons. A more practical pram model. *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[GKS92] L. Guibas, D. Knuth, and M. Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[GS77] P. Green and R. Sibson. Computing dirichlet tessellations in the plane. *Computing Journal*, 21:168–173, 1977.

[GS85] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):75–123, 1985.

[GVW89] J. R. Goodman, M. Y. Vernon, and P. J. Woest. Efficient syncrhonization primitives for large-scale cache-coherent multiprocessors. *Proceedings, Third ASPLOS*, pages 64–75, 1989.

[GW88] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the Fifteenth ISCA*, pages 422–431, 1988.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in fortran d. Technical report, Rice Univ., 1991.

[HM92] M. Herlihy and J. B. Moss. Transactional memory: Architectural support for lock free data structures. Technical report, DEC Cambridge Research Lab, 1992.

[HR89] T. Hagerup and C. Rub. A guided tour of chernoff bounds. *Information Processing Letters*, 33:305–308, 1989.

[HR91] T. Heywood and S. Ranka. A practical hierachical model of parallel computation: The model. Technical report, Syracuse University, 1991.

[JL90] C. S. Jeong and D. T. Lee. Parallel geometric algorithms on a mesh connected computer. *Algorithmica*, 5:155–177, 1990.

[KK87] J. Katajainen and M. Koppinen. Constructing delaunay triangulations by merging buckets in quad-tree order. Unpublished manuscript, 1987.

[KMR90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 177–186, 1990.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *The Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[KRS90] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[KS82] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *20th Annual Allerton Conference On Communication*, pages 35–42, 1982.

[KSR91] Kendall Square Research. *KSR C Programming*, 1991.

[Lem85] Stanislaw Lem. *The Cyberiad: Fables for the Cybernetic Age*. Harvest/HBJ, 1985. Translated by Michael Kandel.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[LKL88] C. Levcopoulos, J. Katajainen, and A. Lingas. An optimal expected-time parallel algorithm for voronoi diagrams. *SWAT*, pages 190–198, 1988.

[LLGG90] D. Lenoski, J. Laudon, K. Gharachorloo, and A. Gupta. The directory-based cache-coherency protocol for the dash multiprocessor. In *Proceedings of the Seventeenth ISCA*, pages 148–159, 1990.

[LLJ+92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Implementation and performance. In *Proceedings of the Ninteenth ISCA*, pages 92–103, 1992.

[LM88] C. E. Leiserson and B. M. Maggs. Communication efficient parallel algorithms for distributed random access machines. *Algorithmica*, 3:53–77, 1988.

[LS80] D.T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *Int. J. of Information Science*, 9(3):219–242, 1980.

[Lu86] M. Lu. Constructing the voronoi diagram on a mesh connected computer. *IEEE Conference on Parallel Processing*, pages 806–811, 1986.

[Mau84] A. Maus. Delaunay triangulation and the convex hull of n points in expected linear time. *BIT*, 24:151–163, 1984.

[McG86] C. McGeoch. *Experimental Analysis of Algorithms*. PhD thesis, School of Computer Science, CMU, 1986.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[Mer92] M. L. Merriam. Parallel implementation of an algorithm for delaunay triangulation. *First European Computational Fluid Dynamics Conference*, pages 907–912, 1992.

[Mil88] V. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. Technical report, School of Computer Science, CMU, 1988.

[MNP+91] P. Mills, L. Nyland, J. Prins, J. Reif, and R. Wagner. Prototyping parallel and distributed algorithms in proteus. Technical report, Duke University, 1991.

[MS90] P. D. MacKenzie and Q. Stout. Ultra-fast expected time parallel algorithms. *ACM-SIAM Symposium On Discrete Algorithms*, pages 414–423, 1990.

[Nat90] L. Natvig. Logarithmic time cost optimal parallel sorting is *not yet* fast in practice! *Supercomputing*, pages 486–494, 1990.

[NV91] M. H. Nodine and J. S. Vitter. Greed sort: An optimal external sorting algorithm for multiple disks. Technical Report CS-91-20, Department of Computer Science, Brown University, 1991.

[OBS92] Atsuyuki Okabe, Barry Boots, and Kokichki Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, 1992.

[OIM84] T. Ohya, M. Iri, and K. Murota. Improvements of the incremental method for the voronoi diagram with computational comparison of various algorithms. *Journal fo the Operations Research Society of Japan*, 27:306–337, 1984.

[PHR92] J. Prins, W. Hightower, and J. Reif. Implementations of randomized sorting on large parallel machines. *Annual ACM Symposium On Parallel Algorithms And Architectures*, 1992.

[PK87] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[PS85] F. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[QHJ88] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais. Compiling c* programs for a hypercube multicomputer. *Proceedings ACM/Sigplan PPEALS*, pages 57–65, 1988.

[RS89] J. Reif and S. Sen. Polling: A new randomized sampling technique for computational geometry. *Symposium on Theory of Computation*, 21, 1989.

[RS92] J. Reif and S. Sen. Optimal parallel randomized algorithms for 3-d convex hulls and related problems. *SIAM Journal on Computing*, 21(3):466–485, 1992.

[RSW90] M. Rosing, R. Schnabel, and R. Weaver. The dino parallel progamming language. Technical report, Univeristy of Colorado, 1990.

[San76] L. A. Santaló. *Integral Geometry and Geometric Probability*. Addison-Wesley, Reading, MA, 1976.

[SH75] M. I. Shamos and D. Hoey. Closest-point problems. *Proc. Sixteenth FOCS*, pages 151–162, 1975.

[SH86] G. L. Steele and D. Hillis. Data parallel algorithms. *Communications Of The ACM*, 29(12):1170–1183, 1986.

[Sto86] I. Stojmenovic. Computational geometry on a hypercube. *International Conference On Parallel Processing*, pages 100–103, 1986.

[SY91] M. Sharir and E. Yaniv. Randomized incremental construction of delaunay diagrams: Theory and practice. *Annual ACM Symposium on Computational Geometry*, 1991.

[TOO83] M. Tanemura, T. Ogawa, and N. Ogita. A new algorithm for three dimensional voronoi tessellation. *Journal of Computational Physics*, 51:191–207, 1983.

[TSBP93] Y. A. Teng, F. Sullivan, I. Beichl, and E. Puppo. A data-parallel algorithm for three-dimensional delaunay triangulation and its implementation. submitted manuscript, 1993.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[VS92] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. Technical Report CS-92-04, Department of Computer Science, Brown University, August 1992. Revised version of Technical Report CS-90-21.

[VVM92] B. Venmuri, R. Varadarajan, and N. Mayya. An efficient expected time parallel algorithm for voronoi construction. *Annual ACM Symposium On Parallel Algorithms And Architectures*, pages 392–401, 1992.

[Wei78] B. Weide. *Statistical Methods for the Analysis of Algorithms*. PhD thesis, School of Computer Science, CMU, 1978.

[WGWR93] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *DAGS '93*, June 1993.

[WL92] A. W. Wilson and R. P. LaRowe. Hiding shared memory reference latency on the galactica-net distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15:351–367, August 1992.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT, 1989.

[Yap90] C. K. Yap. A geometric consistency theorem for a symbolic pertubation scheme. *Journal of Computer and System Sciences*, 40:2–18, 1990.

[ZB91] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. *Proceedings Supercomputing'91*, pages 712–721, November 1991.