PARALLEL PROCESSING FOR GEOMETRIC APPLICATIONS

by

Chandrasekhar Narayanaswami

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer and Systems Engineering

Approved by the
Examining Committee:

Wm. Randolph Franklin, Thesis Adviser

George Nagy, Member

Joshua Turner, Member

Mukkai Krishnamoorthy, Member

Rensselaer Polytechnic Institute
Troy, New York

December 1990
(For Graduation in May 1991)

To my parents.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to thank all those individuals who have contributed, both personally and professionally, to the completion of this thesis.

In particular, I gratefully acknowledge my advisor Professor Randolph Franklin for his support and guidance throughout the course of my thesis. His insight and suggestions have made graduate study a rewarding experience. I furthermore would like to thank my committee members George Nagy, Joshua Turner, and Mukkai Krishnamoorthy for their encouragement and timely suggestions.

I would also like to thank all my friends at Rensselaer who have helped me make it through the last four years. In particular, I would like to thank Amitava Maulik, Maharaj Mukherjee and Dz-Mou Jung for the stimulating discussions; Jun-ichi Kanai, Narendra Nanjangud and Mahesh Viswanathan for going through parts of the earlier drafts of the thesis; and Mohan Kankanhalli for maintaining the computers when they were needed most. All of them and the others in the Computational Geometry Laboratory have made my stay pleasant and memorable.

I would also like to thank my parents, brother, and sister for their love, encouragement, patience and support during the course of my graduate studies. They were a constant source of inspiration and many of my successes have to be attributed to them.

ABSTRACT

A framework is presented for the parallelization of a set of commonly encountered geometric problems. It combines the use of the uniform grid technique, parallel sorting, and data partitioning for parallelization. For many problems, the use of these techniques leads to algorithms whose complexity is linear in the sizes of the input, output, and intermediate results. In average cases, the sizes of the intermediate results are linear in some metric of the input and output. The data structures used by these techniques are simple to build, regularize memory access patterns, and promote locality of memory references in the parallel machine.

The framework is used to develop parallel algorithms for determining the convex hull of a set of points in the plane, the intersections between a set of segments in the plane, and the boundaries of the Boolean combinations of polygons and polyhedra. Approximate complexity analyses for the convex hull and Boolean combination algorithms are provided and compared with experimental results. The implementation on a shared-memory Sequent Balance 21000 shows speedups ranging from 9 to 12 with 15 processors. Close to linear speedups have been achieved in most phases of our algorithms. Implementation on the Intel iPSC Hypercube also shows respectable performance.

CHAPTER 1
INTRODUCTION

1.1 Motivation

The interactive nature of geometric applications and the large quantities of data involved make fast and efficient processing mandatory. However, it is becoming evident that significant increases in speed due to advances in uniprocessor design technology alone will soon be impossible to achieve. Fortunately, the nature of many geometric applications makes *parallel processing* an attractive alternative.

Researchers in computer graphics have concentrated on building special-purpose parallel machines for increasing the speed of specific, yet common, graphics operations such as scan conversion of polygons and rendering complex 3-dimensional scenes [3, 15, 22, 27, 33, 34, 35, 36, 45, 51, 58, 60, 70, 73, 75, 77, 78]. These algorithms operate in image-space and are often architecture-specific. They are useful mainly for purposes of visualization. These characteristics limit their use in general geometric applications. While visualization is an important application for which image-space results may be sufficient, there are many geometric applications which need results in object-space precision. Moreover, these applications deal with many different issues whose requirements cannot all be met efficiently by a special-purpose parallel machine. Therefore, to obtain faster performance for these applications, parallel object-space algorithms for general-purpose parallel machines need to be developed.

Important theoretical advances have been made in developing object-space parallel geometric algorithms in Aggarwal et al. [1], Yap [81], Goodrich [37], Atallah and Goodrich [11], and Miller and Stout [55]. Though a few general paradigms have been identified, research in this area is still in its infancy and there is a huge

gap between the development of algorithms and their implementation. Many of the above algorithms use the more general Parallel-Random-Access-Memory (PRAM) models of parallel computation and ignore some of the practical considerations such as data organization, resource allocation, load balancing, bus contention, and inter-processor communication. They also often assume an impractically large number of processors. In practical situations, the above issues *must* be considered [14, 23, 42]. One of the reasons for this gap between theory and practice is that while research in this area began in 1980, parallel machines such as the Sequent Balance 21000, NCUBE, and Connection Machine have been commercially available only since 1985. Poor environments for software development on parallel machines have aggravated this problem.

As an indication of the importance of research in this direction, we note that four of the eleven problems used as benchmark problems to evaluate parallel architectures for the DARPA Architecture Workshop Benchmark Study of 1986 [17] were problems encountered in geometric applications.

Thus, the important motivating factors for this research are the following:

1. The need for efficient and fast processing of large amounts of geometric data. Parallel processing appears to be a promising solution.

2. The need for object-space algorithms whose functionality is complementary to that of image-space algorithms.

3. The dearth of parallel object-space geometric algorithms which are sufficiently practical to be implemented on the currently available parallel machines.

1.2 Objectives

Many geometric applications, such as Solid-Modelers, Computer-Aided-Design and Geographic-Information-Systems require practical and implementable parallel algorithms. It is necessary to develop a simple yet sufficiently general framework which can be used to design parallel algorithms for these applications.

Since algorithms for these applications are expected to handle large amounts of data efficiently, it is desirable that the complexities of the uniprocessor algorithms be linear in the size of the input and output. It is also desirable that the parallel versions of these algorithms achieve close to linear speedups.

There is a need to analyze the experimental performance of parallel geometric algorithms. The complexities of the various phases in an algorithm have to be studied and phases which could be potential bottlenecks need to be identified. Such results will also help in predicting the performance of the algorithm for larger data sets, and for more massively parallel machines.

This thesis examines the issues outlined above. The results presented in this thesis will be useful for developing parallel algorithms for a variety of applications in the above-mentioned areas.

1.3 Problem Statement

A framework is presented for developing parallel algorithms for a set of geometric problems. This framework is used to design parallel algorithms for the planar convex hull problem, the determination of all the intersections of a set of segments lying in the plane, and the determination of the Boolean combinations of polygons and polyhedra.

The above set of problems have been chosen for the following reasons:

1. They are fundamental low-level operations that are encountered in many geometric applications. For example, they are all frequently encountered in the interference detection problem.

2. They differ in intrinsic complexity and test the framework more thoroughly than any one of them would.

3. They deal with geometric entities of different dimensionalities.

4. They differ in the balance of emphasis on geometric and topological aspects.

Each of the algorithms developed above is implemented on actual parallel machines and their performance is analyzed. The implementations are intended for demonstration of the basic ideas and their practicality, and not for working commercial systems.

1.4 Thesis Organization

Chapter 2 presents the framework and the techniques that are used for developing the parallel algorithms for the above-mentioned problems. The practical advantages of our techniques are identified and the suitability of different models of parallel computation for our algorithms is examined. A brief description of the parallel computers used for the implementation of our algorithms is also provided. Chapters 3-6 each deal with the development and analysis of one of the problems chosen for parallelization. In each of these chapters, the problem to be solved is defined, the previous solutions (if any) are briefly reviewed, and a description and analysis of the algorithm developed and implemented is given. Chapter 7 presents the contributions of this thesis, summarizes the results of our experimental investigations, and discusses avenues for further research.

CHAPTER 2
TECHNIQUES AND TOOLS

In a recent paper, Guibas and Stolfi [41] note that:

> Among the tools of computational geometry there seems to be a small set of techniques and structures that have such a wide range of applications that they deserve to be called fundamental, in the same sense that balanced binary trees and sorting are fundamental to combinatorial algorithms in general.

A framework is proposed for the parallelization of commonly encountered geometric problems. It uses the combination of data-parallelism, the *uniform grid* technique, and parallel sorting. To use the framework, it is first necessary to decompose the algorithm into a set of data-parallel phases. These phases operate on the elements of a tuple-set and generate the elements of another tuple-set. The uniform grid technique is used to prune the sizes of these tuple-sets and to generate sub-tasks. Parallel sorting schemes are used to sort the elements of the tuple-sets when required. The practicality and simplicity of these ideas are demonstrated in Chapters 3 to 6.

The suitability of shared/distributed-memory, SIMD/MIMD parallel machines, and pipelining/parallelism for the implementation of our algorithms is examined. The ease of parallelization of geometric and topological aspects of our problems is considered.

The significant features of the Sequent Balance 21000 and the Intel iPSC1 parallel machines that are used for implementation, are discussed. The practical difficulties encountered in using these parallel machines are also reported.

2.1 Data Parallelism

Most graphics and geometric applications that justify parallelization deal with a large number of objects on which many common operations are performed. It is natural to attempt parallelization of these algorithms by distributing the vertices, edges, faces, or complete objects among the processors. This technique is known as *data partitioning.*

We demonstrate the application of this technique to geometric problems. In order to use this technique, the *attributes* of the *tuple-sets* on which the algorithm has to operate have to be identified. For example, a set of *(cell, edge)* tuples form a tuple-set. For parallelization, the elements of the tuple-sets are distributed among the processors. An element of the tuple-set is defined by a set of attributes. The attributes for the above example are *cell* and *edge.* The number of attributes needed to describe an element of the tuple-set is its *arity.* The number of tuples in the tuple-set is its *cardinality* or size. A tuple-set is also known as a *relation.* However, we find the term tuple-set more appropriate.

From a practical point of view, this technique has the following advantages:

1. ease of implementation,
2. ease of load-balancing among the processors, and
3. regular memory accesses and good use of hardware caching-mechanisms.

However, as the algorithms become more complex, vertices, edges, and faces are no longer sufficient attributes for the tuple-sets, and more complex tuple-sets which are necessary to solve the problem have to be identified. The arity, the attributes themselves, and the cardinality of the tuple-set are dependent on the problem and the cost of inter-processor communication on the parallel machine.

Problems with robustness and the presence of special cases can increase the arity of the tuple-set.

The identification and use of tuple-sets and data-parallelism in our algorithms are demonstrated in Chapters 3-6.

2.2 The Uniform Grid Technique

The *uniform grid* technique [28] (UGT) is a spatial subdivision scheme that divides the extent of a geometric scene uniformly into many smaller subregions. It was first used to develop an expected linear time *sequential* object-space hidden-surface algorithm [29] and *sequential* polyhedron intersection and union [30]. Other applications of this technique are given in [6]. In this thesis, we will demonstrate its use for developing *parallel* geometric algorithms.

The technique exploits the limited spatial extent of individual geometric entities while performing computations on them. Since geometric algorithms deal with large databases, the cardinality of the tuple-set on which the algorithm has to operate plays an important role in its efficiency. The uniform grid technique is used to reduce the cardinality of the tuple-sets in the average case.

This technique is also analogous to the *divide-and-conquer* technique, because problems of the same type but of smaller size are generated for each sub-region. The results for these sub-problems are then merged. The complexity of merging these results to obtain the global solution depends on the problem. The specific use of the uniform grid for the generation of sub-tasks is presented in the following chapters.

The uniform grid is also useful while searching for geometric entities in particular regions of a scene and hence it may be considered as an indexing scheme for locating objects. It improves the average-case query time for the point location problem too.

Data structures similar to the uniform grid, known as *buckets* in [9], have been used for a variety of problems in computational geometry. According to Asano [9], the popularity of bucketing schemes in computational geometry is due to the fact that theoretically better algorithms are often outperformed by more naive methods in many practical situations. Pullar [62] has experimentally shown that the sequential version of the uniform grid technique for determining the intersections between a set of segments lying in the plane is faster than the theoretically better plane-sweep technique for a variety of data sets. The overhead of maintaining the complicated data structure for the plane-sweep technique is cited as the reason for its inefficiency. As will be argued in the following section, in the parallel domain the plane-sweep technique is even less attractive when compared to the uniform grid.

2.2.1 Advantages of the Uniform Grid Technique

It is the following properties of the UGT that make it a useful and practical technique for parallel processing:

1. The reduction of the cardinality of tuple-sets, using uniform partitioning of the region under consideration, is simple and fast. The overhead for maintaining the uniform grid data structure is reasonable.

2. Some optimal geometric techniques such as the plane-sweep technique, used for solving many geometric problems, impose a temporal ordering of computation on the objects. The UGT does not impose one when none is strictly necessary. For example, the computation within the grid cells can be done in any order. Therefore, the uniform grid technique is much simpler to parallelize.

3. The uniformity of spatial partitioning makes it easy to map the sub-tasks onto the processors. A predetermined work partitioning scheme can be used. When more sophisticated schemes such as quadtrees and octrees are used [53, 69], the task of distributing the sub-tasks among the processors becomes more complicated. However, such schemes may achieve a greater cardinality reduction.

4. Unlike hierarchical and adaptive spatial partitioning schemes, the UGT is a single-level partitioning scheme. The data structure for the UGT exploits the large memories of modern machines and avoids indirect references through pointers to locate objects. This helps in avoiding log factors (due to tree searches) in the complexity of the algorithm. In a parallel processing scenario, hierarchical schemes which use a tree data structure will introduce bus congestion because the nodes have to be accessed through a common root.

5. The absence of pointers in the data structure for the UGT and the contiguous location of the tuples increase the locality of memory references and exploit the hardware caching-mechanisms. For algorithms in which objects in adjacent cells in the spatial partitioning interact, it is useful to transform adjacency in space to adjacency in the memory of the machine, in order to improve the memory access patterns. The data structure for the UGT preserves such adjacency. This is important in the context of shared-memory parallel machines because improper use of caching-mechanisms causes severe degradation of performance [25].

2.2.2 The Limitations of the Uniform Grid Technique

Though the uniform grid has many advantages, it has some important limitations:

1. In worst-case situations, when all geometric entities are concentrated in a few grid cells, there is no appreciable reduction in the cardinality of the tuple-set.

2. Uneven spatial densities of the objects reduce the efficiency of the uniform grid technique. However, experiments with the sequential segment intersection algorithm [57] have shown that this is not a serious problem. In the context of parallel computation, this may lead to problems in load-balancing. As shown in Chapters 3-6, the cells have to be assigned to the processors so that the workload is evenly distributed. One solution is to distribute contiguous cells from dense regions to different processors. In such situations, load-balancing and locality of memory references for proper use of caching-mechanisms are conflicting goals.

3. In some applications where separate data structures are used for each grid cell, the memory requirements of the uniform grid technique may be greater than that of other schemes. However, with the decreasing cost of memory, this may not be a serious disadvantage.

2.3 Sorting of Tuple-Sets

Many phases of our algorithms operate on elements of a particular tuple-set and generate the elements of another tuple-set. Before the next phase of the algorithm can proceed, the elements of the new tuple-set have to be sorted by one of their attributes. For example, in our edge intersection algorithm, the first phase uses the set of *(edge)* tuples, determines the cells through which the edges pass, and generates *(cell, edge)* tuples. The next phase determines all the intersections

by comparing the edges in each cell pairwise. Before this phase can proceed, the *(cell, edge)* tuples have to be sorted by the *cell* so that all *(cell, edge)* tuples for a particular cell are in contiguous locations. More examples can be found in Chapters 3 to 6. Since this operation is frequently used in our algorithms, an efficient and implementable parallel sorting scheme must be used.

Ajtai et al. [4] (AKS) and Cole [24] have developed optimal parallel comparison-based sorting algorithms that sort n elements in $O(\log n)$ time using n processors. These algorithms are extremely complicated and impractical. Leighton [50] points out that the constant of proportionality for the AKS algorithm is immense and that it would be slower than other parallel sorting algorithms unless $n > 10^{100}$. Cole's algorithm is believed to have a more reasonable constant. However, Blelloch [13] estimates that its implementation on the Connection Machine would be a factor of 4, and possibly a factor of 10, slower than the split-radix sort [13] and Batcher's bitonic sort [12].

The salient feature of our algorithms is that they very rarely require a parallel comparison sort. This is an advantage because a bucket sort is faster than a comparison sort for sorting records with integer keys that fall within a known range. Furthermore, in a practical situation, the bucket sort shows a better speedup than the comparison sort.

Parallel bucket sort and quicksort algorithms have been implemented for sorting the elements of tuple-sets.

2.3.1 Parallel Bucket Sort

This algorithm is the parallelization of the sequential bucket sort [2]. In order to handle uneven distributions of keys, an adaptive two-level bucketing scheme for sorting is used. At the first level, the complete range of keys is equally divided into buckets. At the second level, the buckets are adaptively divided into bins

depending on the number of elements in them. The range for each bin depends on the number of bins in the bucket to which it belongs. Though our technique has two levels of hierarchy, no pointers are used in the data structure, i.e., a contiguous set of memory locations are used. Hence, memory accesses are more regular.

In the first pass, the algorithm determines the bucket densities and the ranges for the bins in each bucket. Cumulative bucket frequencies are also determined for efficient parallelization of the last phase of the algorithm. This step is done in parallel by distributing the elements among the processors. Collisions in accessing the bucket data structure are resolved by using a suitable collision-resolving mechanism, such as atomic locks and semaphores.

In the second pass, the elements are inserted into the bins to which they belong. Collisions in accessing bins are resolved by the use of collision-resolving mechanisms. After all the elements have been inserted in the appropriate bins, each of the bins is sorted with an insertion sort. The bins can be sorted in any order and hence this step is parallelized by distributing the bins among the processors. The sorted elements in the bins are then copied back into a linear list. Cumulative bucket frequencies help determine the index in the list where the contents of the bins of each of the buckets have to be copied. The above step is done in parallel by distributing the buckets among the processors. There are no collisions in accessing the list in this phase of the algorithm.

Assuming that the adaptive subdivision of the range into buckets and bins limits the maximum number of elements in any particular bin, the complexity of this algorithm is linear in the number of elements.

Noga [59] has independently presented a similar bucket sorting algorithm called the *double distributive partitioning technique.* His technique makes use of sampling to estimate the densities in the buckets and bins.

2.3.2 Parallel Quicksort

This algorithm is the parallelization of the sequential quicksort algorithm [2]. The main idea in the parallelization is that once a pivot element has been located, the list of elements on either side of the pivot can be sorted in parallel. This step is applied recursively till it becomes impractical to create and manage small tasks. This is appropriate as we have a limited number of processors. The records to be sorted are entered into a global task queue. Each processor takes a block of records and either sorts it or subdivides it into simpler tasks and enters it into the global task queue. Thus this step uses *dynamic scheduling* to distribute the work among the processors. Quinn [64] provides a more detailed analysis of this technique and the upper bound of its speedup. For example, the best speedup we could expect with $n = 1028$ and $p = 8$ is 3.375, where n and p are the number of elements and processors, respectively.

2.3.3 Comparison of the Two Sorts

The parallel bucket sort obtains a much higher speedup than the parallel quicksort. The parallel quicksort, however, is an in-place sort and hence requires lesser memory than the parallel bucket sort. For large data sets, the bucket sort is faster than the quicksort by at least a factor of three. For small data sets (less than 500 elements for our implementation), the overhead of determining the densities of the bins makes the bucket sort less attractive. In cases where the elements to be sorted have to be compared explicitly, i.e., pair by pair, they cannot be assigned to buckets, the quicksort has to be used. Otherwise, the bucket sort can be used.

2.4 Practical Issues for using the Framework

2.4.1 SIMD vs MIMD

Since similar operations are performed on all the tuples, most of our algorithms can be implemented on SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data) parallel machines (see [64] for a taxonomy of parallel machines). Though the same operation is performed on the tuples, the complexity of the operation is data dependent. For example, consider the case where each processor is assigned an edge and has to determine the cells of the grid through which its edge passes. The complexity of this operation is dependent on the length of the edge. During each step, processors which deal with short edges have to be idle till the processor with the longest edge finishes. In the MIMD case, such differences are easier to average out over all the elements of the tuple-set. Hence, load-balancing is much more difficult on SIMD machines than on MIMD machines. Branching and random addressing are other reasons for poor processor utilization in SIMD machines. The small amount of memory available in each of the processors of SIMD machines is another disadvantage compared to MIMD machines. As a general rule, as algorithms become more complex, the MIMD model becomes more attractive. Similar experience by Fuchs et al. with the Pixel-Planes 4 machine [36] has resulted in incorporating these ideas in the development of the newer Pixel-Planes 5 machine [33]. As a consequence, all the algorithms developed in this thesis have been implemented on MIMD machines.

2.4.2 Pipelining vs Parallelism

Parallelization by pipelining is useful in cases where a particular phase of the algorithm can start before the preceding phase ends. Such situations arise when geometric entities are handled *independently* of each other as is the case in

some rendering applications in computer graphics. On the other hand, our geometric applications deal with the *interaction* between objects. As a consequence, in most cases all the elements of a particular tuple-set have to be determined before the algorithm can proceed with the generation of elements of the next tuple-set. Therefore, we have found it difficult to make extensive use of pipelining. Only the final stages of some of our algorithms, e.g., the polygon combination algorithm in Chapter 4, support pipelined parallelism.

2.4.3 Shared-Memory vs Distributed-Memory Machines

When algorithms that perform many complicated calculations on the same data are implemented on a distributed-memory machine, it is sufficient to transfer elements of the geometric database to the processors at the beginning of the algorithm. Hence, there is only a minimal need for inter-processor communication during the rest of the algorithm. Many image-processing algorithms fall under this category.

However, many geometric problems do not appear to belong to this class. Algorithms which use our technique perform calculations on elements of different tuple-sets. In a distributed-memory machine this implies that the tuples have to be transferred between the processors as the algorithm proceeds from one data-parallel phase to another. The sorting of tuples and redistribution of the tuples among the processors translates to *all-to-all personalized communication* among the processors. The inter-processor communication schemes available in commercial distributed-memory machines are currently slow and do not meet the requirements for the applications considered in this thesis.

Since inter-processor communication on distributed-memory machines is slow, it is advantageous to store complete entity descriptions in the tuples. For example, while it may be reasonably efficient for an attribute of a tuple-set to be a pointer to

a face in the shared-memory model, it is more advantageous to copy the complete description of the face in the tuple in the distributed-memory model. This causes the size of attributes of the tuple-sets to increase and this in turn increases the volume of inter-processor communication necessary.

Intuitively, the architecture of shared-memory machines bears a closer resemblance to sequential machines than distributed-memory machines do. Our experience also confirms that shared-memory machines are easier to program. Therefore, it is advantageous to implement the algorithms first on shared-memory machines to study their performance. On the other hand, these models do not support massive parallelism because with current technology the common bus used for inter-processor communication tends to saturate performance (speedup) after about 64 processors are introduced. Even with improved technology, this number is not expected to increase by orders of magnitude. Hence, distributed-memory machines will be more prevalent in the future and there is a need to study them.

Parallel machines using a rectangular mesh of buses instead of a single bus are a plausible answer to the above problems [10]. Usually, algorithms developed for PRAM models of computation are made architecture-specific by emulating shared-memory machines on these architectures [65]. This is in accordance with the objective of eventually making parallel algorithms architecture-independent and portable. An approach taken by a few researchers is to develop software environments for distributed-memory machines which make the distinction between the two models transparent to the programmer [72]. However, this research is in its inception and currently it is necessary to know the underlying architecture of the machine to obtain satisfactory performance.

Taking the above factors into consideration, we have implemented most of our algorithms on shared-memory machines. For completeness, we have implemented

the segment-intersection algorithm on a distributed-memory machine. The experimental results from Chapter 7 agree with the arguments put forth here in favor of shared-memory machines.

2.4.4 Topology and Geometry

It is known that the relative importance of *geometric* and *topological* issues for geometric algorithms may vary with the applications for which they are intended to be used. For example, in certain applications such as mass-property determination, interference detection, and viewing, complete topological specification is not essential.

If topological and geometric aspects are equally important, it appears that the geometric aspects have to be parallelized by breaking the topological relations across some geometric boundary. Our algorithm for determining the Boolean combination of polyhedra (Chapter 5) addresses this issue in detail. Since geometric aspects have a higher degree of data-parallelism, they are inherently more easily parallelizable than topological aspects. Topological aspects can exploit data-parallelism only in some cases. When data-parallelism cannot be used, efficient parallel algorithms for sorting and graph traversal are vital for parallelizing the topological aspects. As discussed in Section 2.3, the theoretically optimal parallel sorting algorithms are currently not practical. Though considerable research has been done on developing parallel algorithms for manipulation of graphs, the results are far from encouraging. For example, Reif [66] conjectures that the depth-first search algorithm is inherently sequential.

Algorithms that maintain complete topological relations in the intermediate stages of the algorithm do so by the use of Euler operators or their equivalents. Intuitively, it appears that these algorithms will be inherently sequential, because every Euler operator changes the description of the geometric model. In order to

guarantee consistency in the model, the Euler operators have to be implemented in critical sections. In our parallel algorithm, the topological relations are deduced in a batch mode from geometric results and appropriate additional information (see Chapter 5) derived in the geometric phase of the algorithm. This strategy allows us to use data-parallelism for parallelization.

This approach is practical and attractive in the absence of special cases due to coincident geometric entities, and problems due to numerical errors. However, when they are present, one consistent way to solve these problems is to make use topological information to avoid possible contradictory decisions arrived at by the use of geometry alone. For example, if an edge of a polyhedron lies on a face of another polyhedron, all the faces adjacent to this edge have to reach the consistent decision that they are incident on the face of the other polyhedron. One way to do this is to check for consistency in decisions across processors that are involved in this computation. This implies additional inter-processor communication to resolve ambiguous decisions. Another approach would be to detect the special cases and handle them sequentially by an algorithm which uses topology to make consistent geometric decisions. This strategy ensures that the large number of unambiguous cases are handled rapidly in parallel while the few special cases are handled sequentially. This approach is reasonable as long as there are only a few special cases in any given instance of the problem.

2.5 Two Commercial Parallel Computers

The system configuration, software development environment and the advantages and disadvantages of the Sequent Balance 21000 and the Intel iPSC1 parallel computers are now discussed.

2.5.1 The Sequent Parallel Computer

The Sequent Balance 21000 system is a multiprocessor that incorporates multiple identical processors and a common memory. It can include 4 to 30 processors. It can be configured with 4 to 28 Mbytes of memory and provide 16 Mbytes of virtual address space per process. In addition, each processor has 8 Kbytes of local RAM and 8 Kbytes of cache RAM so that accesses to the system memory are minimized. All the processors access the system memory through a common high-speed bus. Communication between processors is through the shared memory. Process synchronization and avoidance of memory access conflicts are achieved by using semaphores and locks.

The system allows both *function partitioning* and *data partitioning*. Function partitioning refers to the splitting of the tasks to be performed among the processors so that every processor is assigned a separate task. This technique is useful for applications which either use pipelining or need completely independent processes. Function partitioning was not found to be useful in the parallelization of our algorithms.

Microtasking is one of the data partitioning methods available. It is a shared-memory parallel programming model that has a master thread of execution, and zero or more slave threads. The master thread runs the sequential parts of the application, and at appropriate points, causes all the slaves to work with it in parallel. Microtasking programs usually create multiple independent processes to execute loop iterations in parallel. The microtasking support routines provided are suited to data partitioning in a homogeneous application where all processes perform similar work.

The system allows both static and dynamic scheduling in programs using data partitioning. In static scheduling, the scheduling of tasks is known *a priori* before the program execution. In dynamic scheduling, the scheduling of tasks is

determined at execution time. Our algorithms use both forms of scheduling.

The main advantage of the Sequent Balance parallel machine is that it is simpler to program than other parallel machines. Results in Chapters 3 to 5 indicate that it is possible to obtain close to linear speedup for problems in which a high degree of parallelism can be identified. Its main disadvantages are the following:

1. Process creation is expensive. This leads to sub-linear speedups for small problem sizes.

2. The common high-speed bus is used by all processors. This slows down the accesses to the shared-memory as the number of processors increases.

3. The number of processors available is small (16 in the one we used).

4. The grain of parallelism available is limited. For example, the microtask fork utility can only place a single process on a processor. In addition, microtask forks cannot be nested.

2.5.2 The Hypercube Parallel Computer

The hypercube computer is a MIMD, coarse grained, message-passing parallel computer. An n-cube has 2^n node processors. Commercial hypercubes, such as the Intel iPSC and the NCUBE, have up to 1024 node processors. Each node processor has 512 KBytes of memory of which about 280 KBytes are available to the user. In addition to these processors, there is a central processor called the *host* which is used as a controller for the node processors and also serves as an I/O processor. Separate programs need to be written for the host and for the nodes.

The node processors are connected by a hypercube (an n-dimensional cube) interconnection network which is used for exchanging information between them. Software support facilities exist to communicate messages between any two node

processors and between a node and the host. The host and the nodes can communicate directly to each other. This speeds up the broadcasting of programs and data from the host to the nodes.

The hypercube connection is very versatile. The network is completely symmetric with respect to all its vertices and edges. It has good fault-tolerance properties because there are n different paths between any two nodes on an n-cube. Many other network topologies such as a ring, 2D mesh, 3D mesh, and binary tree can be embedded on a hypercube. Since many algorithms have been developed for these topologies, the hypercube can be used to simulate these topologies to implement and verify these algorithms.

The diameter (maximum distance between any two nodes in a graph) of this network is $\log n$. This is important for developing *efficient* parallel algorithms and for efficient communication between any two random processors in the network. An *efficient* parallel algorithm is one which takes $O(\log^c n)$ time with a polynomial number of processors, where c is a constant. For example, sorting n numbers on a $\log n$ cube takes $O(\log^2 n)$ time.

While the theoretical properties of the hypercube network are very good, commercially available hypercube computers have many shortcomings, some of which are expected to disappear as technology improves. Some of the shortcomings of the Intel iPSC1 hypercube which is used to implement the parallel segment intersection algorithm in Chapter 6 are as follows:

1. The lack of a debugging environment makes it very difficult to develop any complex application.

2. The user has the additional burden of having to route messages. This problem can be alleviated by providing the user with a standard set of utilities for commonly encountered communication patterns.

3. The size of the messages is restricted to 16 KBytes. This complicates the message-passing algorithms when larger messages have to be sent. The message-passing mechanism sends 1 KByte even if only one byte has to be sent. This makes short messages uneconomical.

4. A long start-up time to initialize the nodes and a long transit time for messages make the hypercube very slow for solving problems of small sizes. These factors essentially limit the speed of computation on the hypercube. Better performance can be achieved by using the more sophisticated routing algorithms discussed in [43].

5. The sizes of the arrays are restricted to 64 KBytes. This puts a severe restriction on the amount of data that can be handled. This is contradictory because parallel computers are intended for efficient handling of large databases.

2.5.3 Practical Difficulties in Using Parallel Machines

The poor software development environments on most existing parallel machines make the task of implementing a parallel algorithm formidable. Deficiencies include poor error messages from compilers and poor debugging facilities to track program execution. Improved compiler messages about conflicts in using shared variables, race conditions, process synchronization, etc., would reduce programmer time and fatigue. Debugging facilities also have to be enhanced to help detect the above-mentioned problems. Lastly, since geometric applications often involve complex data structures with pointers, graphical debugging facilities will be of immense value to speed up the parallel software development process.

The small amount of memory available on the machines and the large sizes of the geometric databases require the user to design data structures carefully. In some cases the sizes of the arrays have to be changed depending on the number

of processors used, and the program has to be recompiled. This time-consuming process makes measurement of algorithm performance more tedious.

On most message-passing distributed-memory parallel machines, the software support for communicating between the processors is inadequate. Much time is spent in correcting errors which arise from improper synchronization in the message passing protocol.

The scarcity of parallel machines and the immense interest in them results in high demand. The poor support facilities cause the interested user to spend more time on debugging than on useful software development. Finally, since the hardware and system software of parallel machines are complex and fewer people understand it, system downtime is higher.

Fortunately, many of these problems are now being addressed; some require further research.

2.6 Summary

A framework for developing parallel geometric algorithms was proposed. The practical advantages of our framework were identified and the suitability of different models of parallel computation for our algorithms were examined. A brief description of the parallel computers used for the implementation of our algorithms was also provided. A short description of the practical problems of using these machines is also given.

The advantage of this framework is that for many problems its use results in an algorithm whose complexity is linear in the size of the tuple-sets generated in the course of the algorithm. In the average-case, the size of the tuple-sets is linear in the sizes of the input and output of the algorithm.

The disadvantage of these techniques is that the data structures and the representation schemes require more memory than the conventional representation

schemes used on sequential computers. This is the price that has to be paid for incorporating parallelism. However, with the cost of memory is going down, this disadvantage may soon be outweighed.

The following chapters describe the algorithms developed using the ideas presented in this chapter. They show that these ideas can be used to develop parallel algorithms for a variety of applications. These algorithms are sufficiently simple for implementation on general-purpose parallel machines.

CHAPTER 3
PARALLEL CONVEX HULL DETERMINATION IN THE PLANE

We now illustrate the use of the framework presented in the previous chapter for developing a parallel algorithm for the determination of the *convex hull* of a set of points in the plane. As will be shown in the algorithm, this problem has both geometric and topological aspects. It is the simplest of the problems for which we a provide parallel algorithm.

3.1 Introduction

The convex hull is a general construct that is used in many geometric applications. For example, it is useful in obtaining a triangulation of a set of points, topological feature recognition, shape decomposition in pattern recognition, and testing linear separability [61]. The convex hull algorithm is also a *benchmark* algorithm in computational geometry as it is used as a preprocessing step in many geometric problems such as determination of the diameter of a polygon [61].

The following definitions introduce the notion of the convex hull.

Definition 3.1 *Given a set of n distinct points $S = \{p_1, p_2, \cdots, p_n\}$, the set of points*

$$p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3 + \cdots + \alpha_n p_n$$

such that

$$\alpha_j \in \mathcal{R}, \alpha_j \geq 0, \sum_{j=1}^{j=n} \alpha_j = 1$$

is the convex set generated by $p_1, p_2, \cdots, p_n$, and p is a convex combination of $p_1, p_2, \cdots, p_n$.

Definition 3.2 *Given an arbitrary set of points $S = \{p_1, p_2, \cdots, p_n\}$, the convex hull conv(S) of S is the smallest convex set containing S.*

Thus in 2-D, the convex hull is the smallest convex polygon containing the point set S. Most solutions to this problem report the edges of the convex hull in order.

3.2 Preliminaries

The problem of developing a parallel convex hull algorithm has been studied by a number of researchers, and many parallel algorithms have been developed. However, most of these algorithms are not easy to implement. This section surveys some of the existing parallel algorithms for determining convex hulls. Sequential algorithms for this problem are given in [61].

In one of the earliest works in parallel computational geometry, Chow [21] describes a $O(\log^2 n)$ algorithm using $O(n)$ processors for a Concurrent-Read-Exclusive-Write (CREW) Parallel-Random-Access-Memory (PRAM) model of computation. Akl [5] describes a constant-time parallel convex hull algorithm using both $O(n^3)$ or $O(n^4)$ processors and $O(n^3)$ space. Even with such a large number of processors, it just identifies the points which lie on the convex hull and does not order them. Chazelle [19] shows how to solve the problem on a linear array of n elements in a systolic fashion in $O(n)$ time, which is optimal for this model of computation. The *processor* $\times$ *time* $= P \times T$ product for this computation is n^2, which is worse than the sequential bound of $n \log n$, and hence this does not appear to be a good model for determining convex hulls. All of the above algorithms are sub-optimal.

Optimal parallel algorithms, using $O(n)$ processors and achieving the time bound of $O(\log n)$, for determining the convex hull on a CREW PRAM model of computation have been independently developed by Aggarwal et al. [1] and

Goodrich [37]. Both algorithms have a preliminary sorting step and use either the AKS [4] or Cole's [24] optimal parallel sorting algorithm. They then use the $\sqrt{n}$ divide-and-conquer technique to merge the partial results. As mentioned earlier in Section 2.3, the parallel sorting algorithms used in the above algorithms are not practical from the point of view of implementation.

Goodrich [37, 38] also discusses the determination of the convex hull of a sorted point set in parallel. His CREW PRAM algorithm uses a *hull tree* data structure and the $\sqrt{n}$ divide-and-conquer technique, and runs in $O(\log n)$ time using $O(n/\log n)$ processors which is optimal. However, no results from implementation are given.

Miller and Stout [55] provide parallel convex hull algorithms for hypercubes, meshes, pyramids, etc., and also for the PRAM model of computation. Most of their algorithms are optimal for either that architecture or model of computation.

The above algorithms represent some of the interesting theoretical advances made in parallel computational geometry. They have resulted in the development of optimal algorithms on abstract models of parallel computation. However, there are hardly any accompanying experimental results for these algorithms. This indicates that they are not easy to implement on parallel machines available today. The scant experimental results for these algorithms make it difficult to compare them with the more practical algorithms that have been implemented.

The following are some attempts to implement parallel convex hull algorithms. Evans and Mai [26] discuss two parallel implementations for this problem. Their first algorithm uses a preliminary sequential sorting step followed by a parallel divide-and-conquer step. Their second algorithm is a parallel version of the quickhull technique and is based on rejecting points inside the convex hull. Some of their strategies for parallelization are not scalable with the number of processors. They mention that the parallel control overhead of their second algorithm is very

high and also report problems with load-balancing. They have implemented their algorithm on very small data sets (up to 2048 points) using a 4-processor machine. Their implementations have significant sequential sections, and since the fraction of sequential computation which can be tolerated in a parallel algorithm decreases as the number of processors increases (Amdahl's Law [8]), it is not clear how well their algorithm would perform on a machine with more processors.

Blelloch and Little [14] have implemented some algorithms on an n-processor Connection Machine. A modified version of the $\sqrt{n}$ divide-and-conquer algorithm, described by Goodrich [37, 38] and Aggarwal et al. [1], has an asymptotic complexity of $O(\log n)$ using $O(n)$ processors, but has a large constant. According to them it is not the most practical algorithm. Their parallel version of Graham's scan requires $O(\log^2 n)$ router cycles for n points. The Jarvis march requires $O(h)$ time for h hull points but has a small constant; the quickhull technique runs in $O(\log h)$ time for h hull points for well distributed hull points, and has a worst case running time of $O(h)$. In one example consisting of 1000 points, their Graham scan implementation took 150 milliseconds (ms) and the Jarvis march took 12 ms on a 64K processor Connection Machine. Only results on larger data sets would enable us to compare their results with those in Section 3.3.

Cohen et al. [23] have implemented parallel versions of the quickhull technique on the Intel iPSC1 hypercube machine. Their results show close to optimal speedups for each of the iterations of the quickhull. They give the time taken for each iteration of the quickhull but do not give the total time taken by the convex hull algorithm. Also, they do not show the fraction of the total time spent in communication. It is not clear to us whether their timings include the time spent on communication. The absence of such information makes it difficult to compare their results with ours. However, even after accounting for differences in hardware and architecture, their results still seem to indicate that their algorithm

is considerably slower than the algorithm presented in this chapter, for the same data sets.

The algorithm presented in this chapter has been implemented on an actual parallel machine and has been tested on large data sets and on different point distributions. Its minimal reliance on parallel sorting makes it more practical. A major advantage of our technique when compared to the parallelization of the quickhull technique is that the preprocessing phase of our algorithm can be implemented more easily in hardware. Another advantage due to the regularity and simplicity of our approach is that load-balancing and resource allocation are much easier in our algorithm than in the quickhull technique. Finally, it is faster to determine the grid cell to which a point belongs rather than to check whether it is inside a quadrilateral. The algorithm is also robust and does not require points to have distinct x-coordinates, unlike some of the existing algorithms. Thus our algorithm is *practical* in the sense that it is both implementable and robust.

3.3 The Algorithm

We are given a set of n points, $S = \{p_1, p_2, \cdots, p_n\}$ in the plane. We are also given a commercially available tightly coupled parallel machine containing m processors and adequate shared-memory. Note that $m << n$, and so our model is equivalent to a CREW PRAM with a small number of processors.

Our parallel algorithm is based on the fact that optimal parallel comparison sort algorithms are currently impractical, and that for speed and efficiency, an implementable parallel convex hull algorithm should have minimal reliance on a comparison-based sort. The main steps in our parallel algorithm are the following:

1. Fast rejection of interior points.

2. Computation of the convex hull of the remaining points.

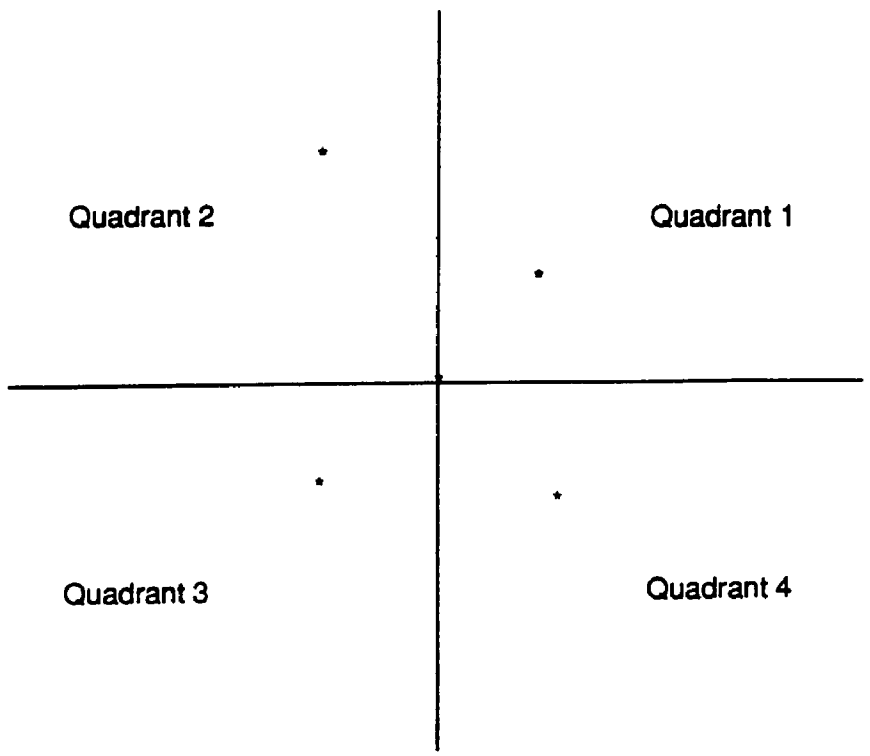


Figure 3.1: The Four Quadrants for a Point. The point at the origin is inside the convex hull.

3.3.1 Rejection of Points Inside the Convex Hull

This step systematically eliminates points inside the convex hull. We use the simple observation that if we orient a set of axes about any point p_i and make p_i the origin of this coordinate system, and if each of the four quadrants in this coordinate system contains points from the input set, then p_i is inside the convex hull (see Figure 3.1). This problem is related to the problem of determining *dominances.* A point p_i *dominates* point p_j in the third quadrant, denoted as $p_i \; dom^3 \; p_j$, if p_j lies in the third quadrant in a coordinate system of which p_i is the origin. Dominances in other quadrants are defined in a similar manner. Thus, points which dominate points in all four quadrants are inside the convex hull. A point which is on the

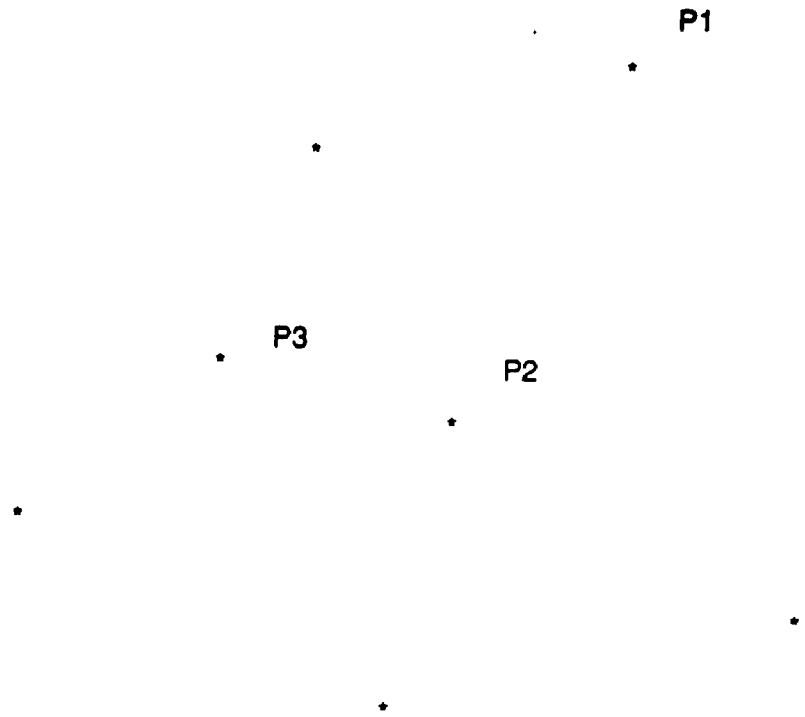


Figure 3.2: An Example Case for Interior Point Elimination

convex hull has the property that it does not *dominate* any other point in at least one quadrant, i.e., there is at least one empty quadrant. However, note that not all such points lie on the convex hull. Figure 3.2 shows an example case. In this example, no point dominates p_1 in the third quadrant, and p_2 is inside the convex hull, since it dominates points in all four quadrants. Point p_3 is not on the convex hull though it does not dominate any point in the second quadrant.

Counting the number of points dominated by each of the points (2-D dominance problem), in each of the four quadrants to determine the interior points is naive, expensive, and unnecessary.

Instead of solving the 2-D dominance problem we cast a uniform grid of $G \times G$ square cells on the scene and eliminate clusters of points at once by performing the dominance computations only for the grid cells. Figure 3.3 shows the four

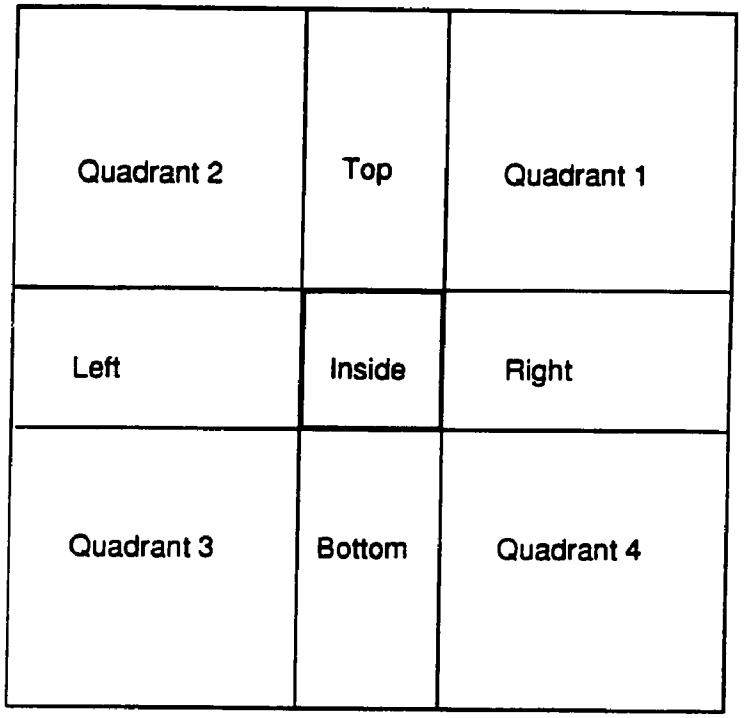


Figure 3.3: The Different Regions for a Cell. Figure shows the regions to the right, left, top, and bottom of a cell. The four quadrants for a cell are also shown.

quadrants for a cell $C_{i,j}$, (i increases from bottom to top and j increases from left to right in the grid), and the regions which are to the top, bottom, right, and left of it. The dominance relations for the cells are analogous to those for the points, e.g., if cells $C_{i,j}$ and $C_{k,l}$ both have some points in them and $i > k$ and $j > l$ then $C_{i,j}$ dominates $C_{k,l}$ in the third quadrant. The dominance value of a cell $C_{i,j}$, for a particular quadrant is 1 if $C_{i,j}$ dominates a cell in that quadrant. We compute the dominance values for the cells and determine the *interior cells.* Interior cells are those cells that dominate other cells in all the four quadrants. Figure 3.4 shows an example case where a 10×10 grid has been used.

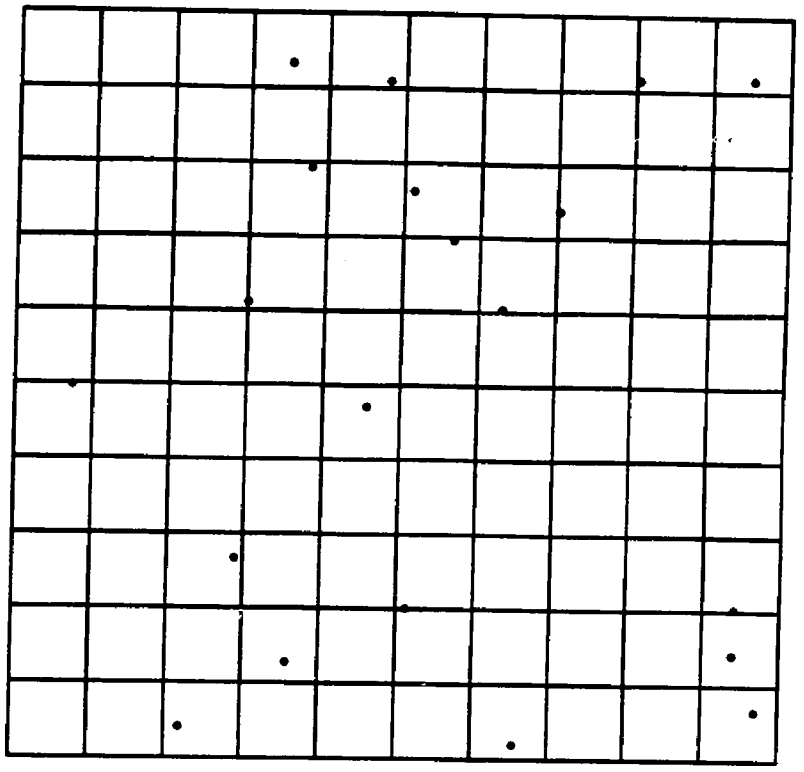


Figure 3.4: Example Point Set for Convex Hull Determination

The main steps in computing the dominance values for the cells are as follows:

1. Determine the cells which contain points; $in(C_{i,j})$ is 1 if there is at least one point in it. To ensure that each point is stored in exactly one cell, only the bottom and left boundaries of a cell belong to the cell. A *predetermined partitioning* scheme is used to distribute the points among the m processors.

2. Determine whether there are points which lie in the cells to the left, right, top, and bottom of each cell. For example, $right(C_{i,j})$ is 1 if there is at least one point in all the cells in that row which are to the right of $C_{i,j}$. The recurrence equations are as follows ($\vee$ is the Boolean OR operator).

$$\begin{aligned} right(C_{i,j}) &= right(C_{i,j+1}) \vee in(C_{i,j+1}) \\ left(C_{i,j}) &= left(C_{i,j-1}) \vee in(C_{i,j-1}) \\ top(C_{i,j}) &= top(C_{i+1,j}) \vee in(C_{i+1,j}) \\ bot(C_{i,j}) &= bot(C_{i-1,j}) \vee in(C_{i-1,j}) \end{aligned}$$

This evaluation is done in parallel by distributing the rows and columns of the grid among the processors. Further, each of these four relations can be determined simultaneously in parallel.

3. Compute the dominance value for each interior cell $C_{i,j}$, using the following recurrence equations.

$$\begin{aligned} dom^1(C_{i,j}) &= dom^1(C_{i+1,j+1}) \vee right(C_{i+1,j+1}) \vee top(C_{i+1,j+1}) \vee in(C_{i+1,j+1}) \\ dom^2(C_{i,j}) &= dom^2(C_{i+1,j-1}) \vee left(C_{i+1,j-1}) \vee top(C_{i+1,j-1}) \vee in(C_{i+1,j-1}) \\ dom^3(C_{i,j}) &= dom^3(C_{i-1,j-1}) \vee left(C_{i-1,j-1}) \vee bot(C_{i-1,j-1}) \vee in(C_{i-1,j-1}) \\ dom^4(C_{i,j}) &= dom^4(C_{i-1,j+1}) \vee right(C_{i-1,j+1}) \vee bot(C_{i-1,j+1}) \vee in(C_{i-1,j+1}) \end{aligned}$$

The above equations need minor modification for border cells. An examination of the equations reveals that the dominance value of a cell depends on quantities that have been computed previously, i.e., it is a recurrence equation. The only dependence of the dominance value of a cell with that of the dominance value of other cells occurs in a diagonal fashion. Hence this step is done in parallel by scanning the grid of cells in a diagonal fashion by interlacing the diagonals among the processors. Further, each of these 4 sets of dominance values can be computed in parallel. Fig 3.5 shows how the computation of dom^3, the dominance value for the third quadrant, is done. The

identifier of the processor assigned to a diagonal is indicated at the starting end of the diagonal. Section 3.5 shows that this method distributes the load fairly evenly among the processors.

4. Eliminate points in all the interior cells. Gather all the points in the cells that are not interior cells. This step is parallelized by using a technique which is similar to parallel array contraction [47]. Figure 3.6 shows the points that remain after the above step for the example depicted in Figure 3.4.

3.3.2 Parallel CH Determination of Remaining Points

Our parallelization of this phase is *not optimal* and so any one of the optimal parallel convex hull algorithms mentioned in the literature review could be used (when they become practical) at this stage to determine the convex hull of the remaining points.

Allison and Noga [7] have determined that in the sequential case, the Graham scan outperforms the Jarvis march and quickhull techniques when the points are in an annulus. Since the points which remain at this stage are in the boundary cells, we use a parallel version of the Graham scan technique. The parallel scheme is iterative and can in the worst case exhibit poorer performance than the sequential Graham scan. Our experimental results indicate that this does not happen for most commonly encountered data sets, and that this step takes a small fraction of the total time. Hence it does not affect the efficiency of the complete algorithm drastically.

The points $\{p_i\}$ which remain after the point rejection step are sorted angularly around a pivot I_p, which is inside the convex hull, to form a star polygon. The angular displacement of a point is the counter-clockwise angle between the horizontal axis and the line joining the pivot and the point. For reasons of numerical stability we never determine the angles by invoking a trigonometric function.

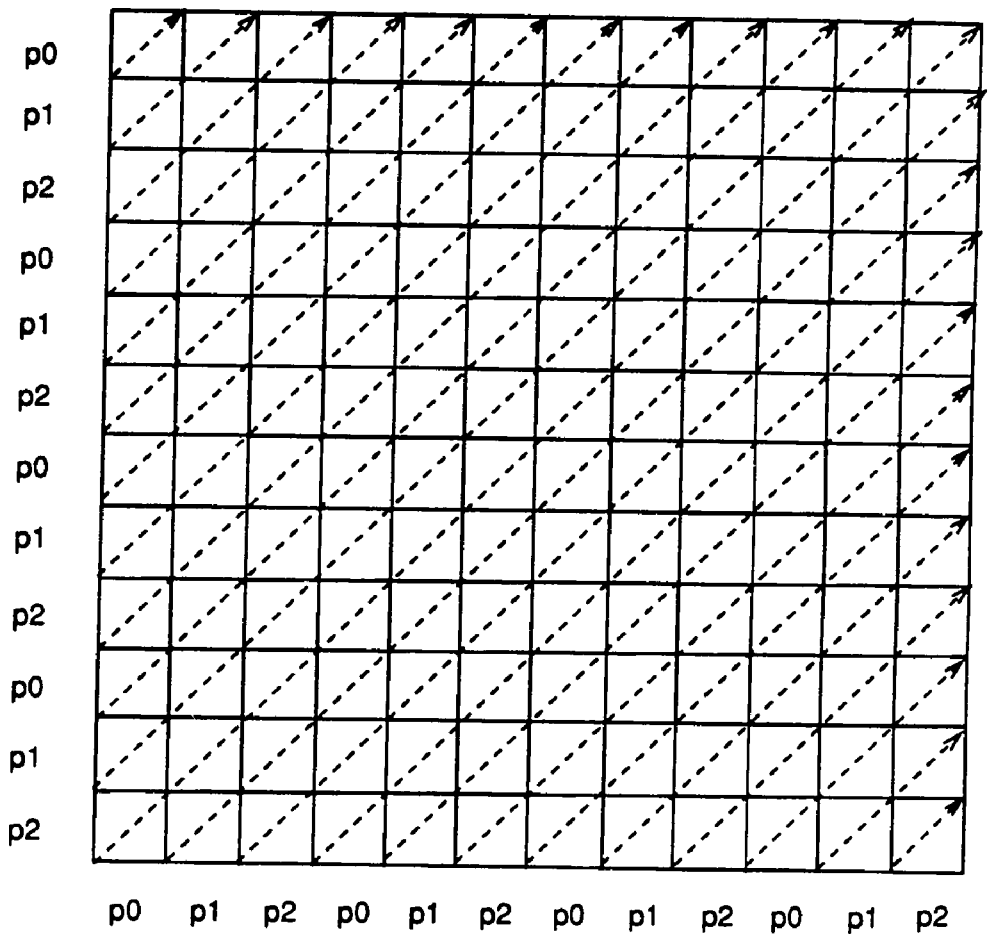


Figure 3.5: Parallel Dominance Computation (third quadrant) for Cells. Figure shows scanning of the grid in a diagonal fashion to determine dominance values in the third quadrant. Processors assigned to the diagonals are shown at their leading end.

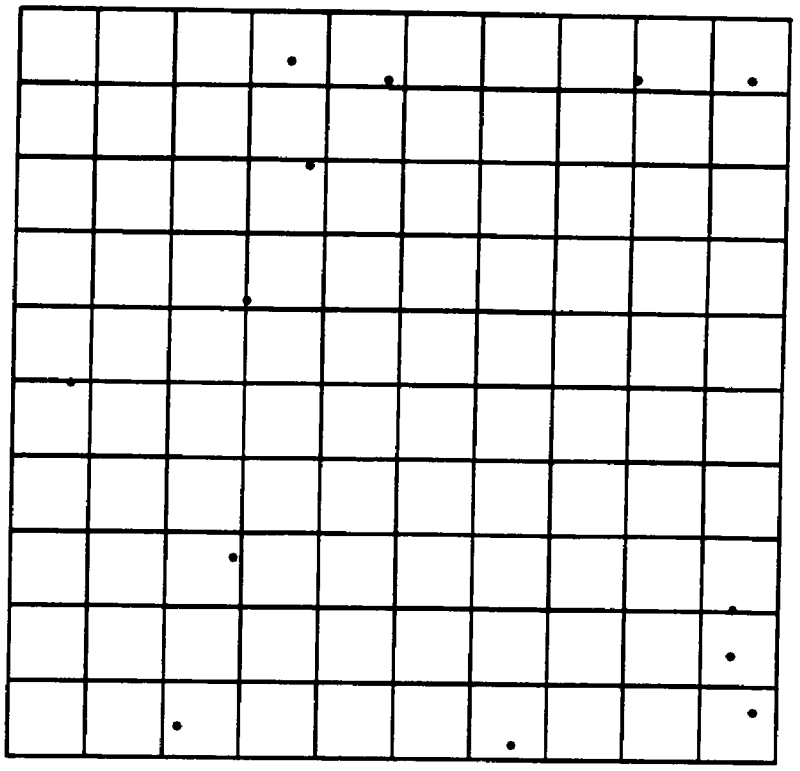


Figure 3.6: Points Surviving Preprocessing Phase for Example Point Set

Instead we compare the relative positions of two points by using vector algebra and simple arithmetic comparisons. Preliminary experiments showed that computing angles and sorting was faster than the above method but was less robust. We have sacrificed speed and overall speedup for robustness in this case. The list is pruned so that if two or more points have the same angular displacement about the pivot, only the farthest among these points is retained. Figure 3.7 shows an example.

The angularly sorted list of points (stored in a shared array) is distributed equally among the m processors so that each processor gets a chain of the star polygon formed by the above step. If very few points are left after the preprocessing phase, the number of processors is adjusted so that each processor gets a chain

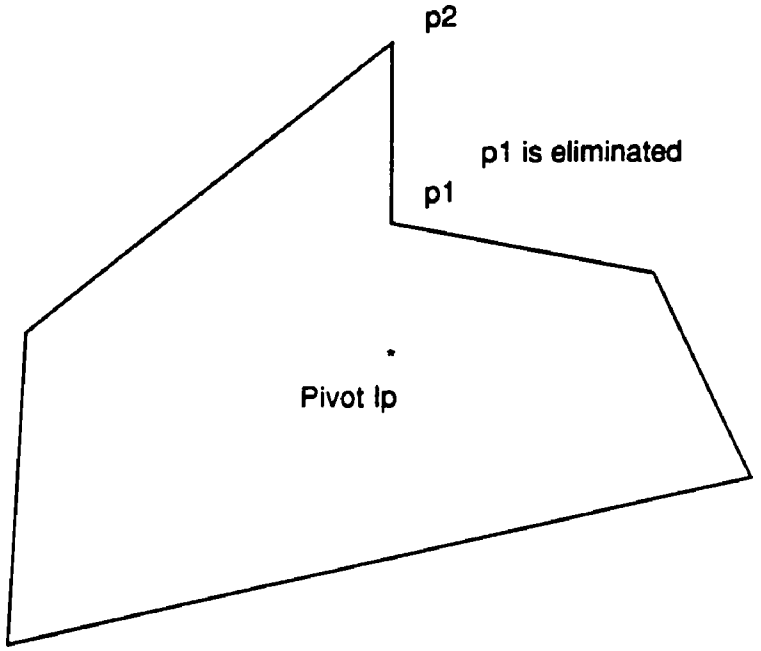


Figure 3.7: Elimination of Radially Closer Points. I_p, p_1, and p_2 are collinear. Point p_1 is eliminated and point p_2 is retained

which has at least 3 points. Each processor then sequentially converts its simple chain into a convex chain. The first three points in the chain are labeled as p_1, p_2, and p_3 respectively. If $\angle p_1 p_2 p_3$ is reflex, then p_2 cannot be on the convex hull, and it is eliminated and p_2 and p_3 are advanced along the chain. If $\angle p_1 p_2 p_3$ is not reflex, all three points p_1, p_2, and p_3 are advanced along the chain. This procedure is repeated till the last point in the processor's chain becomes the current p_3. Figure 3.8 shows the surviving points at the start and end of the first iteration for an example case using 3 processors (I_p is the pivot around which the candidate points have been sorted). It also shows the assignment of the chains to the processors. Every processor also checks whether the joints between its chain and those of its successor and predecessor are convex. If this is not the case, the points causing the concavities are eliminated. Figure 3.9 shows an example. This check is crucial to avoid a possible false termination of the algorithm. Once every processor has

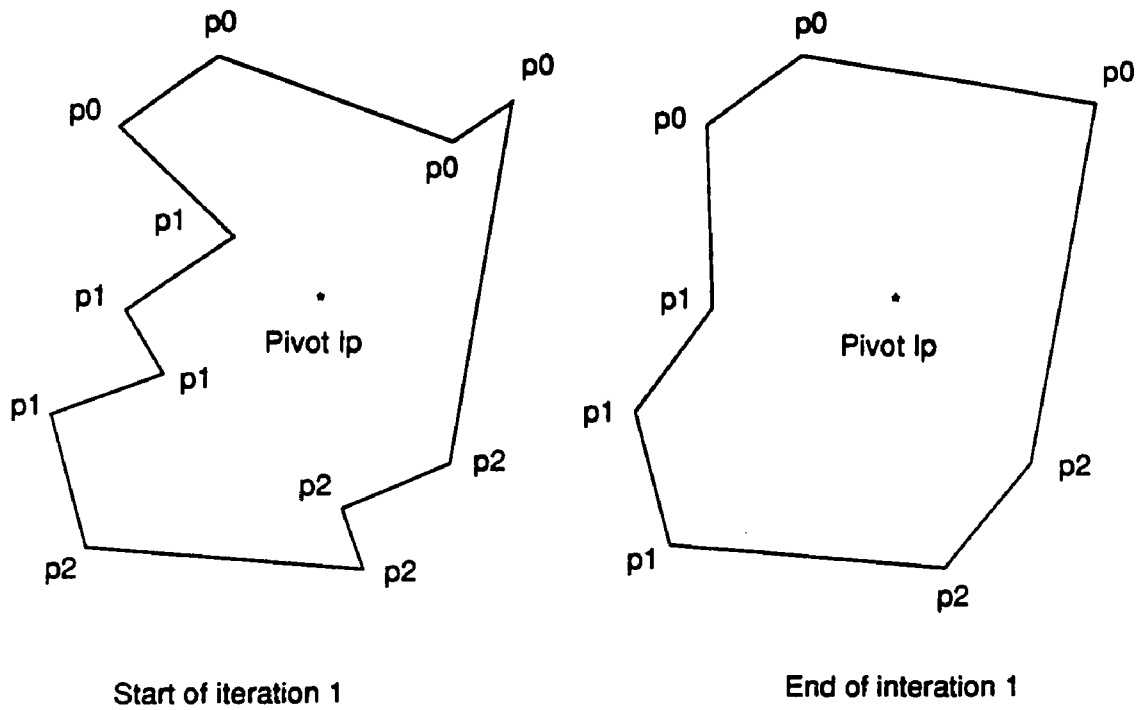


Figure 3.8: Example Case for Parallel Graham Scan. The three processors used are $p0$, $p1$, and $p2$. The processors assigned to points are shown beside them. I_p is the pivot about which points are sorted.

determined the convex hull of the set of points assigned to it, the points which have not been eliminated are contracted into the original array used at the start of the iteration. The index in the array at which each processor has to start writing its points is determined by a parallel prefix computation [47]. The algorithm terminates when the number of points remaining in two consecutive iterations is the same. The points which still remain at the end of this stage form the convex hull of the given set of points.

3.3.3 Location of a Pivot

The centroid of the triangle formed by any three distinct non-collinear points p_a, p_b, and p_c from the set S can be used as the pivot I_p, the point about which the candidate points have to be sorted. I_p will definitely be inside *conv* (S) because

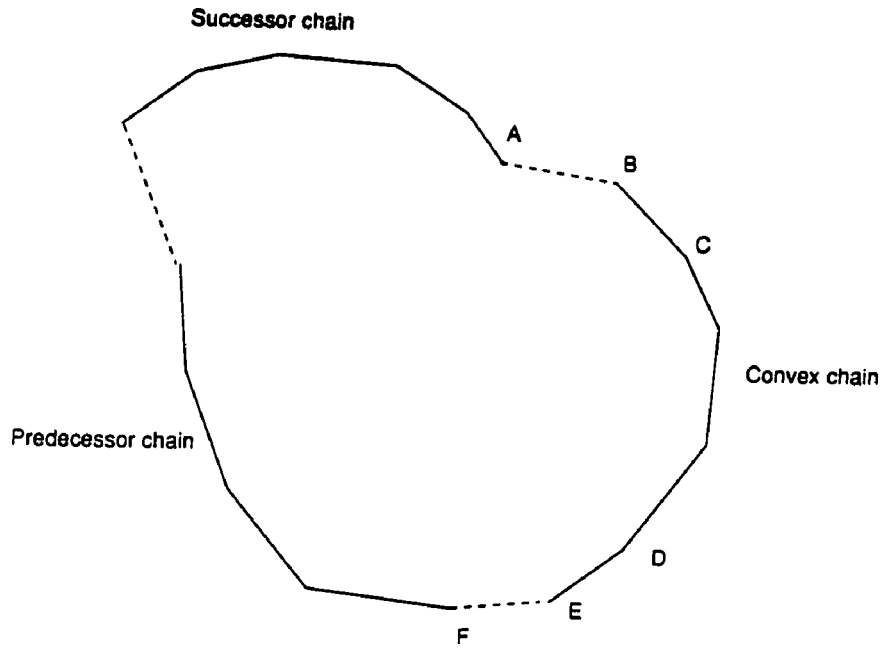


Figure 3.9: Ensuring Convexity at Joints between Convex Chains. To retain B and E for next iteration, $\angle ABC$ and $\angle DEF$ should be less than or equal to 180°.

it lies inside the triangle $p_a p_b p_c$. Even with one processor this step takes constant time in most practical cases. Note that it is possible for I_p, chosen in the above manner, to be coincident with one of the points surviving the preprocessing phase. Such cases are identified during sorting and since such points lie inside the convex hull they can be eliminated.

3.3.4 Parallel Sorting Around the Pivot

We use the parallel quicksort algorithm for this step. As already mentioned, this is not an optimal algorithm but is practical. Since almost 90-95% of the points are eliminated in the preprocessing phase in our test sets , the sorting step accounts for a much smaller fraction of the total time than would be the case without preprocessing. We expect this to be the case for most commonly

encountered distributions. The distributions for which our preprocessing phase would not be very effective are discussed in the conclusion.

When the sorting time constitutes a larger fraction of the total time, and when a larger number of processors are available, the overall speedup will be critically reduced if a poor parallel sorting algorithm is used. In such cases, the parallel bucket sort could be used. However, since it is not a comparison sort, we would have to use trigonometric functions to determine the angular displacements of the points and it could lead to numerical problems.

3.4 Proof of Correctness

Lemma 3.1 *The points eliminated in the point-rejection step lie inside the convex hull.*

Proof: Obvious.

Lemma 3.2 *The parallel Graham scan can terminate when the number of points in two consecutive iterations is the same. The points which remain at this stage form the convex hull of the set* $\{p_i\}$.

Proof: At the end of every iteration of the parallel Graham scan, all the chains assigned to the processors are convex. In addition to this, since a processor looks at the joints between its chain and those of its predecessor and successor, if the number of points in two consecutive iterations remains the same, all the joints between the chains belonging to consecutive processors are convex. Since the concatenation of convex chains, such that the joint between them is convex, yields a longer convex chain, the resulting polygon is the convex hull.

Using Lemma 3.1 and Lemma 3.2 it is clear that the algorithm correctly computes the convex hull of the given set of points.

Corollary 3.1 *The maximum number of iterations of the parallel Graham scan is* $(n_r - h + 1)$*, where* h *is the number of points on the convex hull, and* n_r *is the number of points in the first iteration of the parallel Graham scan.*

Proof: From Lemma 3.2 it follows that the number of points reduces by at least one for each iteration. Since $(n_r - h)$ points have to be removed, $(n_r - h)$ is the maximum number of iterations required to converge to the convex hull. One additional iteration is needed to detect the convergence.

3.5 Analysis of the Algorithm

Throughout this section we assume that there are n points in the original data set. m represents the number of processors and G refers to the grid resolution. Unless stated otherwise, $m << G^2$. Since m is usually small on a commercial shared-memory machine, $\log m$ can be ignored when compared to G^2 while evaluating complexities.

3.5.1 The Preprocessing Phase

- *Assigning points to cells*:

  The time taken for this step depends on the distribution of the data. If a cell is currently empty and if two or more processors simultaneously deal with points falling in the same cell, there would be a collision in writing into the cell. Once a cell is marked as *not-empty* there are no further write-collisions for that cell. When points are chosen randomly from a uniform distribution, the probability of the above collisions is very small, and diminishes as G increases. Hence, we expect this step to show close to optimal speedup, and the expected time for it is $T_1 = \theta(n/m)$.

- *Computing Booleans for $left(C_{ij})$, $right(C_{ij})$, $top(C_{ij})$ and $bot(C_{ij})$:*

  A total of G^2 computations have to be done to determine each of the above quantities for all the G^2 grid cells. Assuming that G is a multiple of m, each processor gets G/m rows (or columns). The computations for each row or column are performed sequentially. Hence it takes $T_2 = G^2/m$ time for this step. The work done in this phase, i.e., the $P \times T$ product, is $m \times G^2/m = G^2$ which is equal to the total number of grid cells. This method distributes the load fairly evenly among the m processors even when G is not a multiple of m.

- *Dominance computation:*

  A total of $G + 2\sum_{i=1}^{G-1} i = G^2$ computations (see Figure 3.5) need to be performed for each of the four dominance relations. As shown below, these can be done in $T_3 = G^2/m$ time when G is a multiple of m. For simplicity of exposition, consider the computation of dom^3 for the case where $G = k \times m$. For the diagonals in the upper left triangle (excluding the leading diagonal) of the grid cells, the ith processor ($i \neq (m-1)$) has to perform W_i^U computations,

$$\begin{aligned} W_i^U &= \sum_{j=l_1}[(i+1)] + [(i+1)+m] + [(i+1)+2m] + \cdots + [(i+1)+l_1 m] \\ &= \sum_{j=0}^{l_1}[(i+1)+jm] \\ &= (i+1)(l_1+1) + m(l_1^2+l_1)/2, \end{aligned}$$

  where l_1 is the largest integer such that $(i+1)+l_1 m < G$. This implies that $l_1 = \lfloor (km-(i+1))/m \rfloor = k-1$. Similarly, in the lower right triangle, the ith processor has to perform W_i^L computations,

$$\begin{aligned} W_i^L &= \sum_{j=l_2}[(G-1-i)] + [(G-1-i)-m] + \cdots + [(G-1-i)-l_2 m] \\ &= \sum_{j=0}^{l_2}[(G-1-i)-jm] \end{aligned}$$

$$= (G-1-i)(l_2+1) - m(l_2^2+l_2)/2,$$

where l_2 is the largest integer such that $(G-1-i) - l_2 m > 0$. This implies that $l_2 = \lfloor (km-(i+1))/m \rfloor = k-1$. The total work done by the ith processor is

$$\begin{aligned} W_i &= W_i^U + W_i^L \\ &= (i+1)k + m(k^2-k)/2 + (G-1-i)k - m(k^2-k) \\ &= kG = G^2/m \end{aligned}$$

Since all the processors except the $(m-1)th$ processor have been covered by the above proof, the $(m-1)th$ processor has to perform only G^2/m computations. Hence the load is distributed evenly among the processors when G is a multiple of m. If this is not the case there is a slight imbalance.

- *Gathering surviving points*:

  One parallel pass through the n points is sufficient to determine the surviving points. This takes $T_4 = n/m$ time when local buffers are used in each processor to minimize collisions while writing into global lists.

Therefore $T_{preproc} = T_1 + T_2 + T_3 + T_4 = \theta((n+G^2)/m)$ time. Choosing $G^2 <= n$, $T_{preproc} = \theta(n/m)$.

3.5.2 Radial Sorting Phase

For uniform distributions, the number of points remaining at this stage is proportional to the number of boundary cells. If the density of points in the boundary cells is N_{bc} and the expected number of boundary cells is $n_b = \alpha_1 G$, then the number of points is $n_r = N_{bc} \times n_b = N_{bc} \times \alpha_1 G$. For uniform distributions, $N_{bc} = \alpha_2 n/G^2$ and hence $n_r = \alpha_1 \alpha_2 n/G$. α_1 and α_2 are parameters that depend on the distribution. For exponential and multi-cluster distributions n_r is even smaller.

Even if sequential sorting is used, the time for this step is $O(n_r \log n_r)$, and since $n_r << n$, this is not a serious shortcoming. However, it is obvious that as soon as a better practical parallel sorting algorithm becomes available, the performance of our algorithm will improve automatically.

3.5.3 Parallel Graham Scan Phase

Each iteration of the parallel Graham scan takes n_i/m time when there are n_i points in the ith iteration. So if there are k iterations, this step takes $\sum_{i=1}^{k} n_i/m$ time.

Experimental results show that the algorithm usually takes less than 5 iterations for data sets containing up to 200,000 points. However it is not difficult to construct an input set which will take $\theta(n_r)$ iterations. Figure 3.10 shows such a case. In this case only one point is eliminated in every iteration and the convex hull of the set $\{p_i\}$ is the convex hull of the chain allocated to one processor in the last iteration of the algorithm. The work done by all the other processors in all the previous iterations is wasted. We did not encounter any pathological cases for the point distributions we considered. In all the cases we tried, the time taken by a multiprocessor algorithm was always less than the time taken by a uniprocessor algorithm. Immunity to such pathological cases can be increased by randomly varying the point in the star polygon from which chains of equal length are allocated to the processors in successive iterations. It will ensure that a processor receives considerably different chains in consecutive iterations and will make the algorithm more immune to pathological cases. One could also randomly vary the number of processors used in each iteration. It would also increase the probability of forming radically distinct convex chains in consecutive iterations and thereby make the algorithm less susceptible to pathological cases.

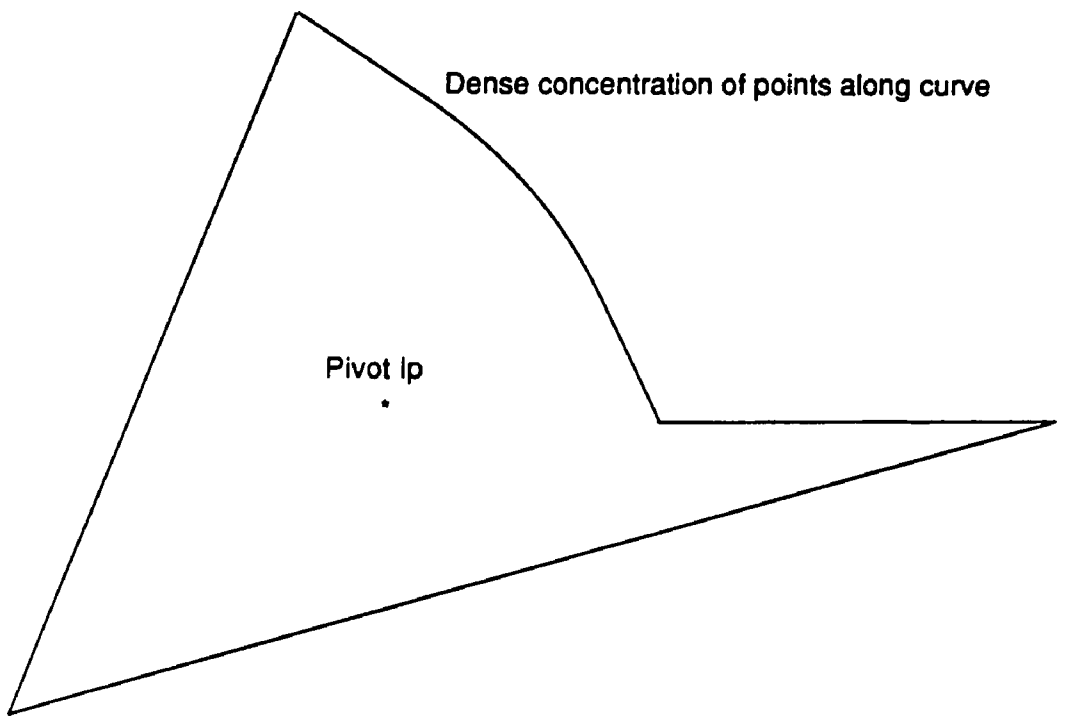


Figure 3.10: Pathological Case for the Parallel Graham Scan. Point density along curved section is high. Work done by processors handling points in this section is wasted.

3.5.4 Space Complexity

It is clear from the description of the algorithm that the space requirement is $\theta(n + G^2)$. Since G, the grid resolution, is chosen such that $G^2 < n$, the space requirement is $\theta(n)$.

3.6 Implementation and Results

The above algorithm was implemented on a 16-processor Sequent Balance 21000 machine. Shared data structures were used to store the points and cells.

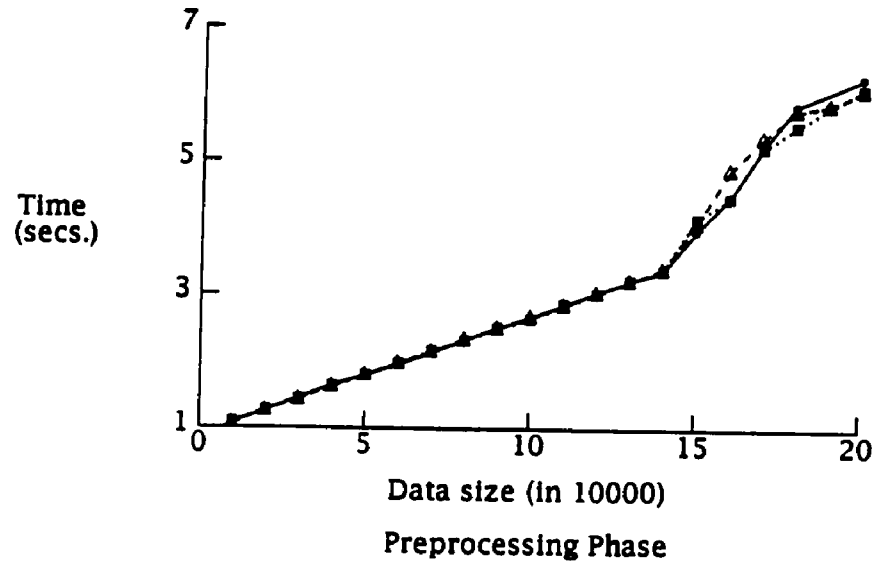
Since the performance of the algorithm is data dependent, it has been tested on uniform distributions of points within a unit square, a unit circle, and an annulus. These data sets have been chosen because they differ in the standard deviation of the number of points per cell and the expected number of points on the convex hull. Therefore, the algorithm is expected to spend different fractions of time in the preprocessing and postprocessing phases. Tables 3.1 and 3.2 verify this claim.

Data sets with up to 200,000 points have been used to evaluate the performance of our algorithm. For 200,000 points from a circle, square, and annulus, it took 8.7, 12.5, and 13.88 seconds, respectively. These results indicate that if the speed of the individual processors increases, interactive performance can be achieved. Tables 3.1 and 3.2 give more details of our experiments.

Figure 3.11 shows the variation of the time taken, as a function of the input size for the three distributions. For data sets containing up to 150,000 points, the growth rate is close to linear. This result agrees with the complexity analysis of the algorithm. For larger data sets, deviation from linear growth is observed. The causes for it are examined later.

Figures 3.12, 3.13, and 3.14 show the speedup achieved for the three types of data. Each type of data has been tested with sets containing 100,000 points and 200,000 points. For 100,000 points, the average speedup achieved for the preprocessing phase is about 12.5 with 15 processors. The corresponding figure for the 200,000 point case is 9. This drop in speedup is not predicted by the complexity analysis and demonstrates the effect of practical issues. An increase in the number of cache misses, and in collisions in accessing the cells, are probable

Dotted lines : Uniform distribution within an annulus (R1 = 0.3, R2 = 0.45)
Dashed lines : Uniform distribution within a square (L = 0.9)
Solid lines : Uniform distribution within a circle (R = 0.45)
Grid Size : 120 x 120



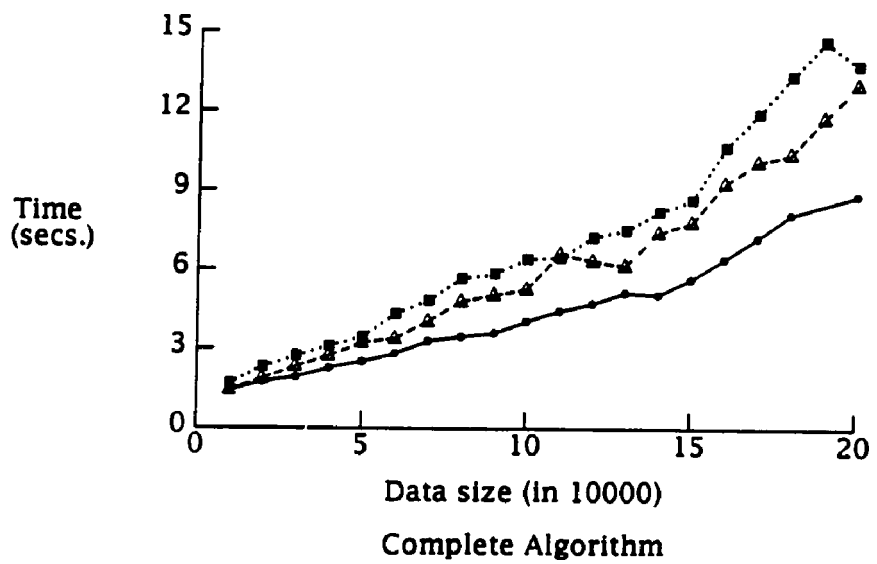


Figure 3.11: Timing Behavior of Algorithm with Data Size

Distribution : Points chosen randomly and uniformly from a circle (R = 0.45)
Grid Size : 120 x 120
Dashed lines : Preprocessing phase
Solid lines : Complete algorithm

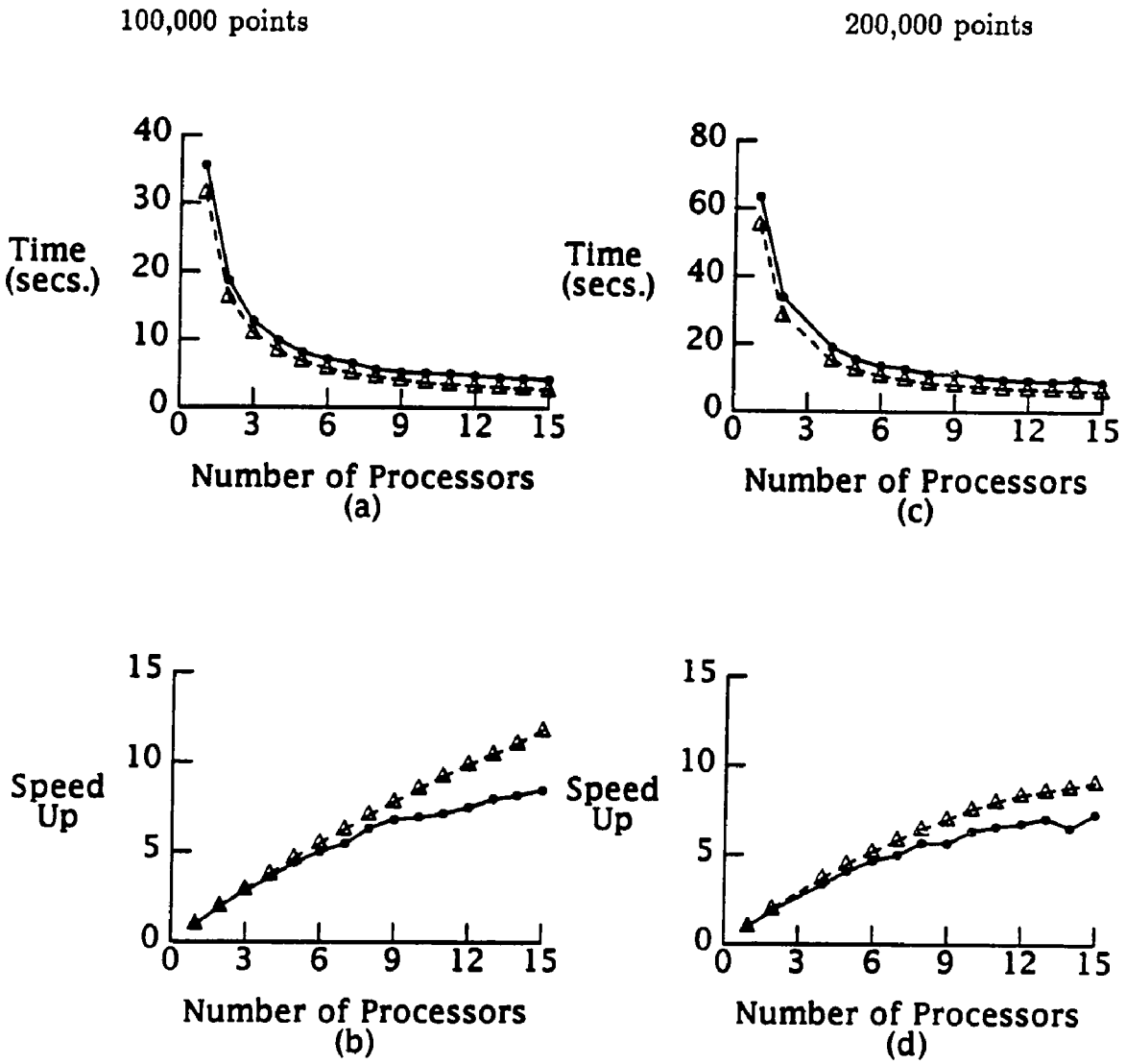


Figure 3.12: Timing and Speedup for Points in a Unit Circle

Distribution : Points chosen randomly and uniformly from a square (L = 0.9)
Grid Size : 120 x 120
Dashed lines : Preprocessing phase
Solid lines : Complete algorithm

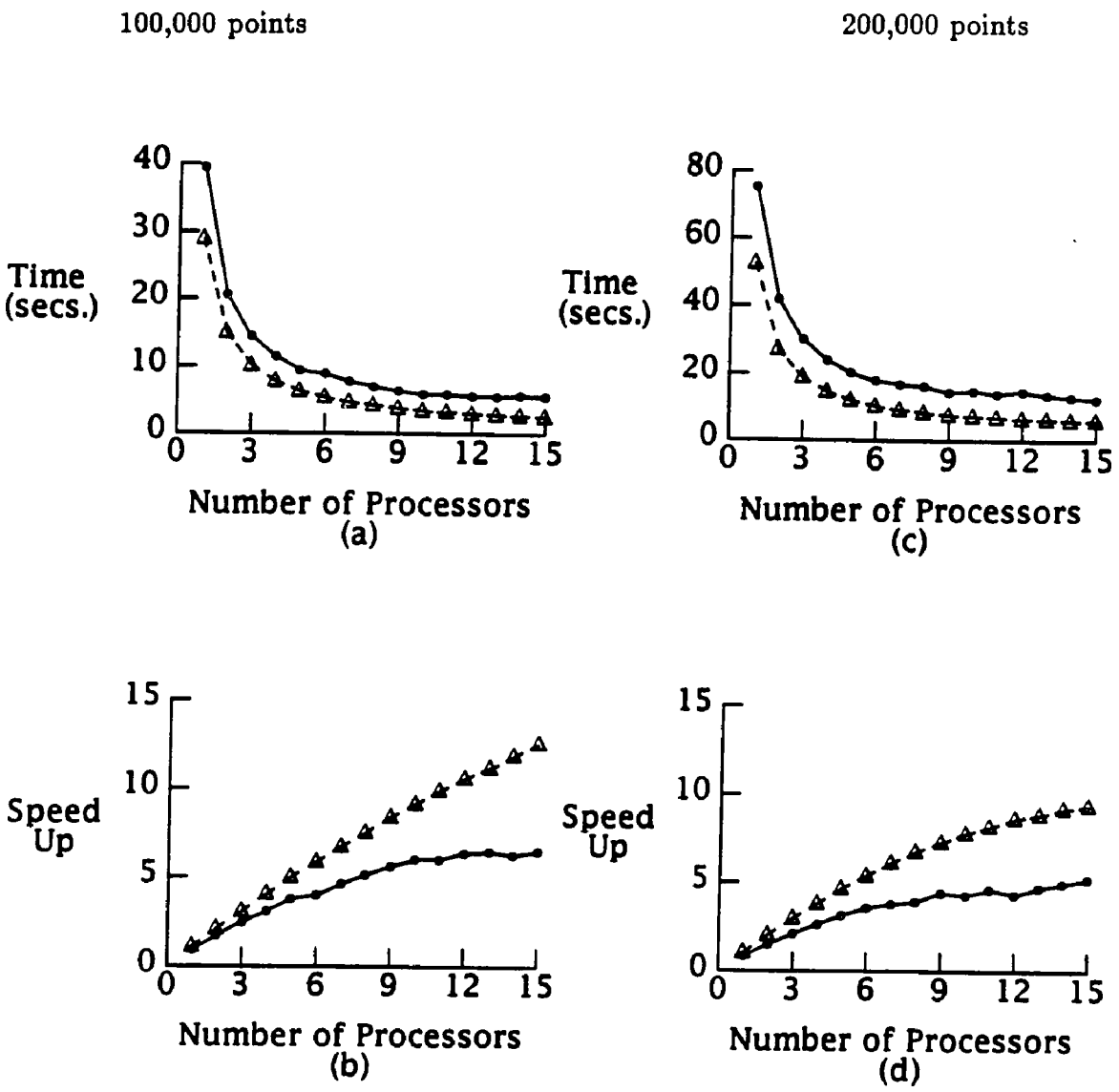


Figure 3.13: Timing and Speedup for Points in a Unit Square

Distribution : Points chosen from an annulus (R1 = 0.30, R2 = 0.45)
Grid Size : 120 x 120
Dashed lines : Preprocessing phase
Solid lines : Complete algorithm

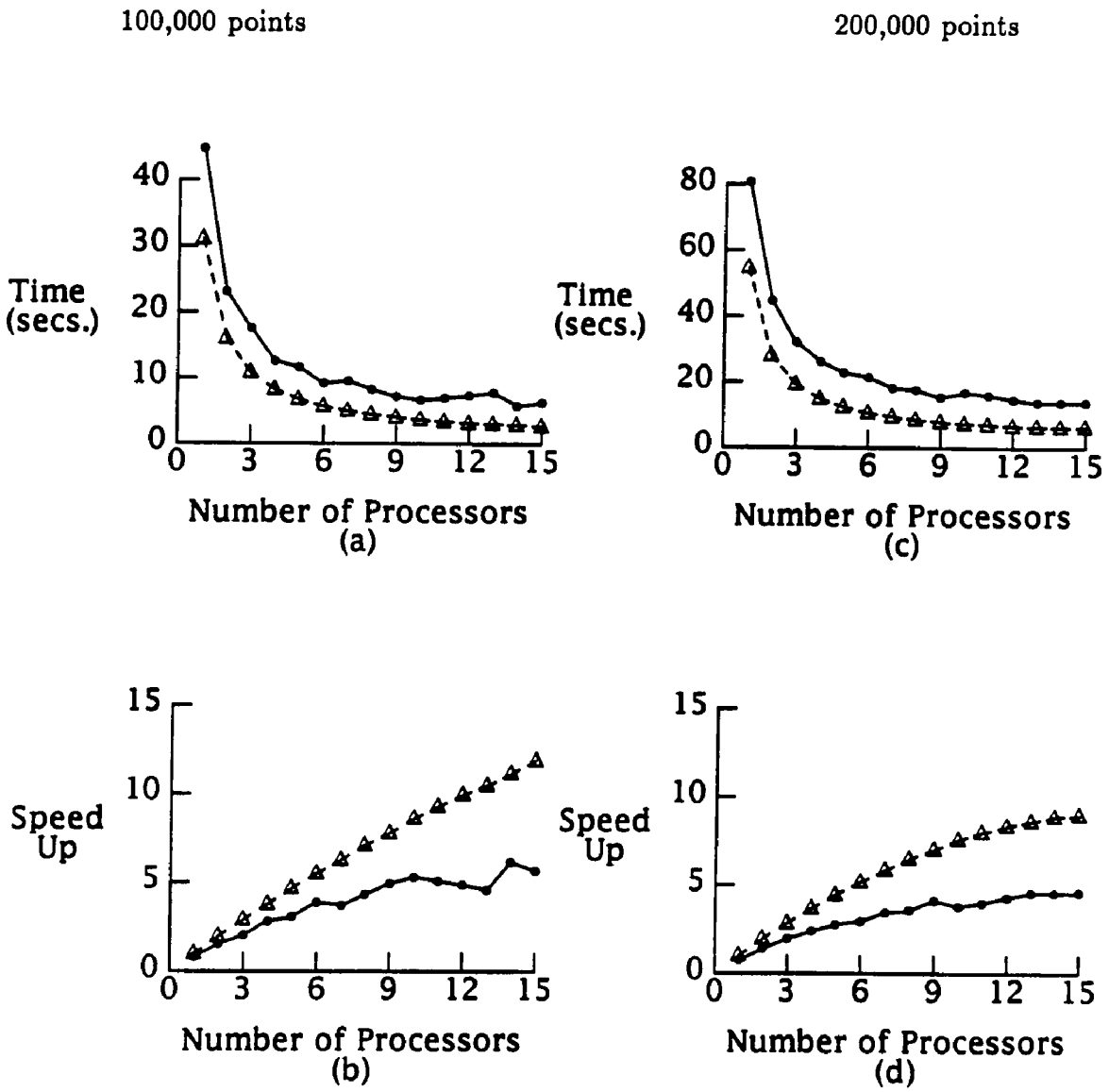


Figure 3.14: Timing and Speedup for Points in an Annulus

Table 3.1: Complexity of Individual Steps (100,000 Points)

Operation	Time 1 proc	% time	Time 15 procs	% time	Speedup
100000 points in a circle (R=0.45), Grid size = 120 451 cells and 1666 points after preprocessing					
Putting points in grid	14.37	40.33	1.50	35.80	9.58
Dominance computation	7.89	22.14	0.54	12.89	14.61
Pruning cells	1.67	4.69	0.12	2.86	13.92
Candidate determination	7.46	20.94	0.50	11.93	14.92
Vector determination	0.19	0.53	0.10	2.39	1.90
Radial Sorting	3.35	9.40	1.25	29.83	2.68
Eliminate duplicates	0.07	0.20	0.07	1.67	Sequential
Hull Determination	0.63	1.77	0.11	2.63	5.73
Complete algorithm	35.63	100.00	4.19	100.00	8.50
100000 Points in a square (L=0.9), Grid size = 105 416 cells and 3973 points after preprocessing					
Putting points in grid	14.47	36.70	1.50	27.32	9.65
Dominance computation	5.55	14.08	0.38	6.92	14.61
Pruning cells	1.38	3.50	0.10	1.82	13.80
Candidate determination	7.52	19.07	0.52	9.47	14.46
Vector determination	0.44	1.12	0.13	2.37	3.38
Radial Sorting	8.55	21.68	2.55	46.45	1.84
Eliminate duplicates	0.16	0.41	0.16	2.91	Sequential
Hull Determination	1.36	3.45	0.15	2.73	9.07
Complete algorithm	39.43	100.01	5.49	99.99	7.18
100000 Points in an annulus (R1=0.3, R2=0.45), Grid size = 120 428 cells and 4619 points after preprocessing					
Putting points in grid	13.77	30.83	1.45	23.09	9.50
Dominance computation	8.16	18.27	0.56	8.92	14.57
Pruning cells	1.65	3.69	0.11	1.75	15.00
Candidate determination	7.54	16.88	0.52	8.28	14.50
Vector determination	0.53	1.19	0.23	3.66	0.72
Radial Sorting	11.12	24.89	3.02	48.09	3.68
Eliminate duplicates	0.19	0.43	0.19	3.03	Sequential
Hull Determination	1.71	3.83	0.20	3.18	8.55
Complete algorithm	44.67	100.01	6.28	100.00	7.11

Table 3.2: Complexity of Individual Steps (200,000 Points)

Operation	Time 1 proc	% time	Time 15 procs	% time	Speedup
200000 points in a circle (R=0.45), Grid size = 120 451 cells and 2937 points after preprocessing					
Putting points in grid	29.00	45.83	4.31	49.54	6.73
Dominance computation	7.91	12.50	0.56	6.44	14.13
Pruning cells	1.67	2.64	0.11	1.26	15.11
Candidate determination	16.50	26.07	1.10	12.64	15.00
Vector determination	0.55	0.87	0.39	4.48	1.41
Radial Sorting	6.36	10.05	1.75	20.11	3.63
Eliminate duplicates	0.12	0.19	0.12	1.38	Sequential
Hull Determination	1.17	1.85	0.36	4.14	3.25
Complete algorithm	63.28	100.00	8.70	99.99	7.27
200000 Points in a square (L=0.9), Grid size = 105 416 cells and 7573 points after preprocessing					
Putting points in grid	29.14	38.73	4.28	35.08	6.81
Dominance computation	5.55	7.38	0.39	3.20	14.23
Pruning cells	1.38	1.83	0.10	0.82	13.80
Candidate determination	16.38	21.77	1.11	9.10	14.76
Vector determination	0.94	1.25	1.16	9.51	0.81
Radial Sorting	18.86	25.07	3.93	32.21	4.80
Eliminate duplicates	0.31	0.41	0.31	2.54	Sequential
Hull Determination	2.67	3.55	0.92	7.54	2.90
Complete algorithm	75.23	99.99	12.20	100.00	6.17
200000 Points in an annulus (R1=0.3, R2=0.45), Grid size = 120 428 cells and 8601 points after preprocessing					
Putting points in grid	28.42	35.16	4.30	30.98	6.61
Dominance computation	8.22	10.17	0.61	4.39	13.48
Pruning cells	1.64	2.03	0.12	0.86	13.67
Candidate determination	16.40	20.29	1.12	8.07	14.64
Vector determination	1.03	1.27	1.39	10.01	0.74
Radial Sorting	21.58	26.70	4.92	35.45	4.39
Eliminate duplicates	0.35	0.43	0.35	2.52	Sequential
Hull Determination	3.19	3.95	1.07	7.71	2.98
Complete algorithm	80.83	100.00	13.88	99.99	5.82

causes for the decrease in speedup (with constant G) with an increase in data size. The kinks in the graph for overall speedup are due to the unpredictable variation of time taken by the parallel quicksort.

Tables 3.1 and 3.2 show the times spent and the corresponding speedups achieved, by the various phases of the algorithm, for each of the distributions. Just as our complexity analysis predicts, all steps in the preprocessing phase show good speedup. Though the preprocessing phase eliminates 95% of the points for these examples, the sorting phase accounts for about 10-25% of the time in the sequential case. Moreover, it shows speedup of only 2.5 to 4.5. This gives an indication of the speedup that would be achieved by algorithms that sort all the points in the data set in a preprocessing phase. It also explains the deviation from linear speedup for the complete algorithm. The effects of non-linearity (as the number of processors increases) in speedup and time, due to sorting, can be reduced by using a finer grid. It would make the sequential algorithm spend more time in the preprocessing phase which shows good speedup, and less time in the sorting phase which shows a poor speedup. Tests also show that the sorting time can be cut to approximately a fourth of the present time by resorting to angle computations using inverse trigonometric functions. However, as stated earlier, robustness has been given priority over speed. The parallel Graham scan phase shows speedups ranging from 3 to 9, but this phase takes less than 5% of the total sequential time and hence it does not critically affect the overall performance.

Figure 3.11 shows that the times taken for the different distributions are within a factor of 1.6. From the description of the algorithm and the analysis of experimental results, it is clear that our algorithm would exhibit similar performance for many other distributions. Distributions in which points are distributed along a contour however, will be bad data sets for our algorithm.

3.7 Conclusion

The convex hull algorithm presented combined the use of the uniform grid, a parallel sort, and simple data partitioning for parallelization.

The uniform grid was used effectively (speedups more than 9 with 15 processors) to transform in parallel the domain of points into the domain of candidate points. This phase linearizes the complexity of the algorithm by eliminating most of the interior points for many commonly encountered distributions in linear time. The simplicity and regularity of this technique are the main reasons for its efficiency. This property also makes this phase well suited for hardware implementation.

The parallel quicksort was used to transform the domain of candidate points to that of chains. The parallel bucket sort was not used in order to maintain robustness by avoiding use of inverse trigonometric functions. Poor performance of the parallel quicksort algorithm limited the overall speedup of our algorithm. The availability of a faster and more easily implementable comparison-based parallel sort in future would immediately improve the overall performance of our algorithm. The parallel Graham scan phase of the algorithm is heuristic and not optimal; however, pathological cases for this phase are not expected to occur for most commonly encountered data sets.

Results from implementation reveal that simple data partitioning by distributing points, cells, rows, columns, diagonals, and chains to the processors lead to fairly even load-balancing for this problem.

The following chapters show further use of this technique for solving more complicated problems.

CHAPTER 4
PARALLEL POLYGON COMBINATION

The use of the framework presented in Chapter 2 in the development of a parallel algorithm for the *polygon combination* problem is demonstrated. The parallel polygon combination problem is intrinsically more complex than the convex hull problem. It also deals with more complex geometric entities, i.e., faces instead of points. Our algorithm concentrates only on the parallelization of the geometric issues of the problem. In this algorithm we also observe the effect of *special cases* due to incidence.

4.1 Introduction

In the context of geometric applications, a *Boolean* operation refers to a set operation on geometric objects. The objects are represented as r-sets [67], and the set operators such as the union, intersection and difference, are regularized [67] operators, which remove dangling edges and faces. Regularized set operators and r-sets form a Boolean algebra.

The problem of polygon combination in the context of this chapter is as follows: Given the *boundary* of two polygons, we need to determine the boundaries of the various regularized Boolean combinations of the two polygons such as the intersection, union and difference. The term *boundary* is elaborated in the following section. Figures 4.4 to 4.7 illustrate the problem.

Boolean combinations of polygons are used in the boundary evaluation of objects defined in the Constructive Solid Geometry (CSG) representation scheme. They are also used for clipping polygons and computing visible surfaces in computer graphics and for computing the areas of overlapping layers of circuit elements in VLSI. Currently there is no literature on parallel algorithms for this problem.

4.2 Assumptions

We are given the edges of the polygons, P_α and P_β, in a random fashion. The edges are oriented such that the interior of the polygon lies to their right. The polygons can have holes and can have more than one disjoint component. The format of the result of our algorithm is the same as that of the input. Section 4.4 discusses how the contours of the resulting polygon can be determined. The contours are not necessary for a variety of applications such as viewing, interference detection, and calculation of mass properties.

We assume that all vertices in the polygons are separated by a minimum distance of ϵ. During the intermediate stages of the algorithm, vertices closer than ϵ are treated as coincident. Collinear edges whose vertices can be treated as coincident are in turn treated as coincident. This treatment does not solve the fundamental problem of numerical errors, but is practical.

4.3 The Algorithm

The main steps in our CREW PRAM parallel algorithm are as follows:

1. Compute the points of intersection between the edges of the two polygons.

2. Use the intersections computed above to partition the edges of the polygons into sub-edges whose classification, i.e., whether they are inside, outside, or on, with respect to the other polygon, is constant. Whenever possible, classify the sub-edges by an analysis of the local neighborhood of the intersection. If the sub-edge lies on the other polygon, its relative orientation with respect to the other polygon is also determined. Table 4.1 shows all the possible classifications for a boundary element.

3. Classify the edges of each polygon that do not intersect the other polygon, and the unclassified sub-edges, with respect to the other polygon, by using a point-with-respect-to-polygon test.

4. Determine which sub-edges or edges need to be included to obtain the boundary of the desired Boolean combination. A salient feature of our algorithm is that the edges of all Boolean combinations can be determined by using a single table.

4.3.1 Parallel Computation of the Points of Intersection

The uniform grid technique is used for computing the points of intersection in parallel and is described below:

1. Partition the 2-D region of interest into $G \times G$ uniform square cells. The fineness of the grid G, is a function of the length of the edges in the input polygons. Usually $G = cL^{-1}$ is a good heuristic, where L is the average length of the edges and c is a tuning constant. The left and bottom boundaries of a cell belong to it, whereas the top and right boundaries do not. Appropriate adjustments are made while dealing with the peripheral (border) cells.

2. Insert the edges of the polygon into the cells of the grid, i.e., determine the *(cell, edge)* tuples. The cells which an edge passes through are determined by using a variant of the Bresenham line drawing algorithm [16]. If an edge passes through a grid corner, it is entered in all four cells adjacent to the corner (see following discussion).

   This step can be executed in parallel because the computation is mutually independent with respect to the edges. A *predetermined partitioning* scheme is used to distribute the edges among the processors. The only contention for resources that occurs in this step is when more than one processor wants

to write into the same cell's edge list. This problem is solved by using *atomic locks* to lock the cell data structure whenever it is updated.

3. For each cell C_i, use the *(cell, edge)* tuples to determine the relevant intersections among the edges passing through it. Note that a pair of edges have to be tested for intersection only if they belong to different polygons. In order to avoid multiple reporting of the same intersection between a pair of edges going through many adjacent cells, the intersection is reported only if the point of intersection belongs to the cell in which the pair is tested. This strategy does not work in the polyhedron combination algorithm in Chapter 5. Intersections that fall on the cell boundaries are handled correctly because no boundary belongs to more than one cell.

   If two edges intersect at a grid corner, they must be tested for intersection in all four grids cells incident at this grid corner. If they are tested in only one of the four cells, there is a possibility of missing the intersection because the computed point of intersection may lie in one of the other three cells, due to limited numerical accuracies on the computer. However, when the edges are checked for intersection in all four cells, only one of the cells will report the intersection because the point of intersection is unique. Moreover, a point-in-cell test can be devised such that a point belongs to only one cell. If there were no problems with limited numerical accuracies, an edge passing through a grid corner need be entered consistently in only one of the four cells. Therefore, we observe that the cardinality of the set of *(cell, edge)* tuples can increase due to limited numerical accuracies.

   In order to handle cases of collinearity and incidence (see Figure 4.1), overlapping collinear edges and edges which touch each other at a vertex, are not considered to intersect. However, if a vertex of one edge lies in the

interior of the other edge, the two edges are considered to intersect. This takes care of the special cases of overlapping collinear edges, and vertices lying on edges or other vertices. An intersection due to incidence is called an *improper* intersection and an intersection due to crossing edges is called a *proper* intersection.

For every pair of edges e_i of P_α, and e_j of P_β, that intersect at the point (x, y), two tuples, $(e_i, e_j, x, y, type\_of\_xsect)$ and $(e_j, e_i, x, y, type\_of\_xsect)$ are created. The type of intersection, i.e., whether it is due to a vertex incidence or a proper crossing of edges, is also recorded in the tuple. Note that the presence of special cases has increased the arity of the tuple-set.

This step is executed in parallel by distributing the grid cells among the processors because the computation is mutually independent with respect to the cells. To achieve better load-balancing in cases where certain regions contain a significant number number of edges, adjacent cells are given to different processors.

4. The set of $(e_i, e_j, x, y, type\_of\_xsect)$ tuples is now sorted by the first edge, e_i, of the tuple so that all the points of intersection along each edge are in consecutive locations. The parallel bucket sort or quicksort may be used for this purpose.

4.3.2 Formation and Classification of Sub-edges in Parallel

This step is parallelized by distributing the $(e_i, e_j, x, y, type\_of\_xsect)$ tuples among the processors so that each processor is responsible for a few edges. Each edge is divided into sub-edges with constant classification in the following manner:

1. The elements of the set of $(e_i, e_j, x, y, type\_of\_xsect)$ tuples corresponding to intersections along the edge, e_i, to be partitioned and classified, are converted

to a set of $(e_j, x, y, type\_of\_xsect)$ tuples. The vertices of the edge being partitioned are also entered into this set of tuples as they each form a vertex of a sub-edge. The set of $(e_j, x, y, type\_of\_xsect)$ tuples is sorted by either the x or the y coordinate of the point of intersection, depending on the slope of the edge, to increase numerical stability of the sort. Intersection points which are within a distance of ϵ are coalesced. If any of these intersection points is within a distance of ϵ from the vertices of the edge, they are coalesced with the vertex. Figure 4.1 shows how the previous steps take care to create sub-edges of non-zero length.

2. Consecutive elements from the sorted set of $(e_j, x, y, type\_of\_xsect)$ tuples that are separated by a distance greater than ϵ define the vertices of the sub-edges into which the edge is to be partitioned. The sub-edge is oriented in the same direction as the original edge of which it is a part. If at least one of the vertices of the sub-edge was the result of a *proper* intersection it is referred to as a *proper* sub-edge. The other sub-edges are referred to as *improper* sub-edges.

3. All proper sub-edges are classified at this stage by an analysis of the $(e_j, x, y, type\_of\_xsect)$ tuples responsible for causing them. Figure 4.2 shows how the orientation of the edges is used for this purpose. The information encoded in *type_of_xsect* is used to determine whether the sub-edge is a proper sub-edge or not. The classification of improper sub-edges is described in the following section.

4.3.3 Classification of Unpartitioned Edges and Improper Sub-Edges

Unpartitioned edges and improper sub-edges of partitioned edges are classified by using a point-with-respect-to-polygon test. Note that we cannot propagate the classifications from one unpartitioned edge to another along the boundaries of

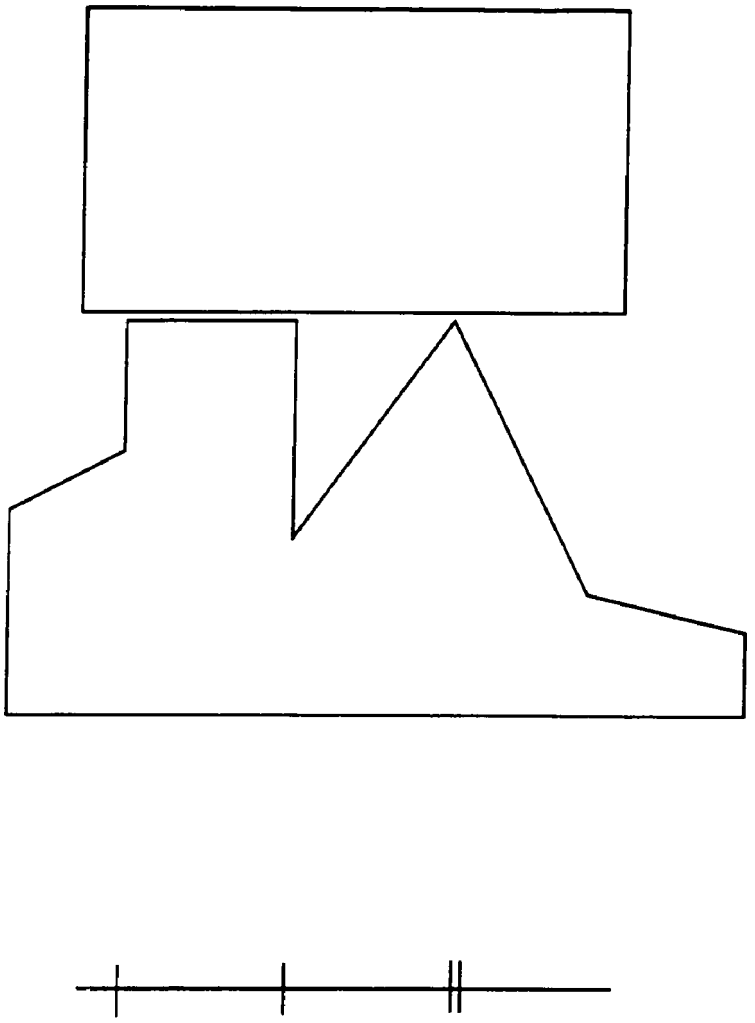


Figure 4.1: Special Cases Occurring in Polygon Combination. Edges incident on other edges create duplicate intersections.

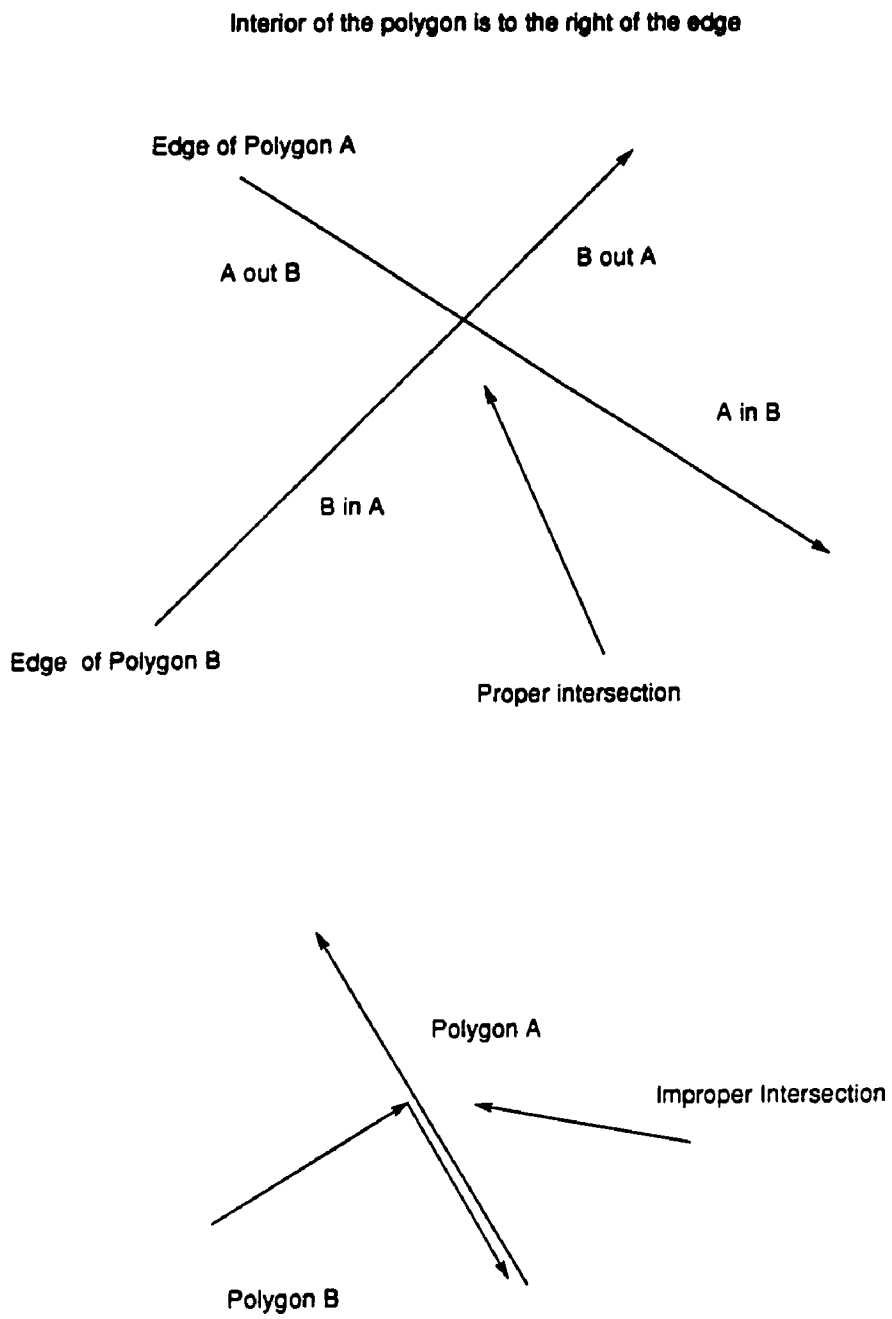


Figure 4.2: Classification of Sub-Edges at a Proper Intersection Vertex. Sub-edges that are to the left of the edges causing them are outside the other polygon.

the polygons as we do not know the contours of the polygons. The point-with-respect-to-polygon test is done by passing a horizontal ray from the midpoint of the sub-edge to be classified to the right extremity of the scene boundary, to determine the first edge of the other polygon it intersects. The advantage of using the uniform grid technique is that only those edges of the polygon in the cells which the ray passes through are involved in this computation. A horizontal ray is used as opposed to a vertical ray because the grid cells are stored such that contiguous cells of the grid in the horizontal direction are in contiguous memory locations of the machine. The ray-shooting procedure ends at the first cell in which the ray intersects an edge or edges of the other polygon. In this cell, determine the first edge of the other polygon to intersect the ray. Check whether the point to be classified lies to the right, left, or on this edge and classify the sub-edge appropriately. If the ray does not intersect or lie on any edge, the point to be classified lies outside the other polygon. Figure 4.3 shows an example.

This step is done in parallel by distributing among the processors, the edges and sub-edges that are yet to be classified.

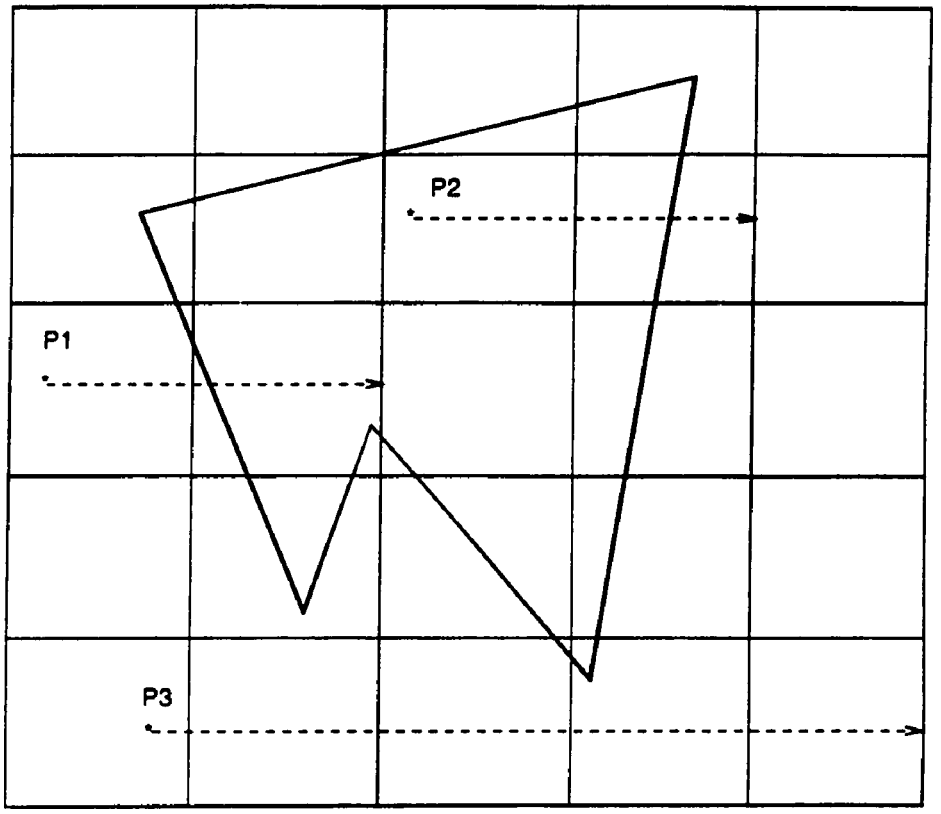


Figure 4.3: Point Classification w.r.t. Polygons using the Uniform Grid. Points P_1 and P_3 are outside the polygon. P_2 is inside. Note that the ray stops at the appropriate cell.

Table 4.1: Segments to be Included in Various Polygon Combinations

Elements to include	$P_\alpha \bigcup P_\beta$	$P_\alpha \bigcap P_\beta$	$P_\alpha - P_\beta$
Element of P_α inside P_β		+	
Element of P_α outside P_β	+		+
Element of P_β inside P_α		+	−
Element of P_β outside P_α	+		
Element of P_α on P_β, in same direction	+	+	
Element of P_β on P_α, in same direction			
Element of P_α on P_β, in opp. direction			+
Element of P_β on P_α, in opp. direction			

Table for other combinations are similar

4.3.4 Selecting the Appropriate Sub-Edges

Table 4.1 shows the elements to be included to obtain the boundary of each of the commonly encountered regularized Boolean operations. To use the table, select the operation desired, and then read down that column. For each row below with a "+" or "-", read to the left to find the type of element to include. A "-" means that the element is to be used with its direction reversed and a "+" means that the element is to be used in the same direction as reported. For example, if the elements for $P_\alpha \bigcup P_\beta$ are needed, all elements of P_α which are outside P_β, all elements of P_β which are outside P_α, and all elements of P_α which are on P_β and have the same orientation, i.e., the interiors of both polygons lie to the same side of this element, are included in the result.

A look at the elements chosen for $P_\alpha \bigcup P_\beta$ and $P_\beta \bigcup P_\alpha$ reveals that the table is asymmetric with respect to P_α and P_β. The asymmetry is introduced to avoid duplicate edges in the final object which would appear otherwise, due to overlapping collinear sub-edges.

4.4 Topological Aspects

The above algorithm and its implementation do not deal with determination of the contours in the resulting polygon. We now propose an approach for this aspect.

The edges in the result are sorted by the coordinates of their vertices. Consecutive elements in the sorted list can then be used to determine the edges incident at the vertices. This phase can be parallelized by using the parallel bucket sort and data partitioning.

The edges in the contours and the complete hierarchical organization of contours can be derived as shown in Section 5.4.5.1. This process appears to be sequential.

4.5 Implementation and Results

The algorithm was implemented on a 16-processor Sequent Balance 21000 parallel machine. Shared global arrays were used to store the edges, sub-edges, list of intersections, and the lists of *(cell, edge)* tuples. We did not use a parallel sorting algorithm to sort the $(e_i, e_j, x, y, type\_of\_xsect)$ tuples in our implementation, as sorting was not an intensive component of the algorithm.

The algorithm was tested on a variety of data sets. Figures 4.4 and 4.5 show two different types of data sets used. In the first example, the edges of the polygon were generated using a random number generator, with the restriction that a polygon should not intersect with itself. In the second example, each polygon consists of a set of squares. These polygons have been chosen because they differ in shapes and standard deviation of the edge lengths, and are thus expected to spend different fractions of time in the various phases of the algorithm. In the first example, the two polygons have 12,000 and 6,800 edges respectively, and in the second example, both polygons have 7,200 edges each. Due to the limited

resolution of the hard copy devices, the figures show data sets which are smaller than the ones actually used. Figures 4.6 and 4.7 show the results of our algorithm for the data sets shown in Figures 4.4 and 4.5, respectively.

Figures 4.8 and 4.9 show the times taken by the various phases and the speedup of the algorithm as a function of the number of processors for both data sets discussed. The figures show that close to linear speedup was achieved in both cases. This indicates that the total time taken by the algorithm would continue to decrease as more processors are added, till other factors such as bus capacity and inherently sequential sections begin to dominate. A speedup of 13.50 and 10.31 with 15 processors was achieved for the first and second examples, respectively. Using 15 processors, computing all the Boolean combinations for the first example took 177.19 seconds, using a 97×97 grid. For the second example, the corresponding time was 18.03 seconds, using a 100×100 grid.

Table 4.2 shows the fraction of time taken, and the corresponding speedups achieved, for each phase of our algorithm for both the examples. Table 4.3 shows the time taken for the various phases of the algorithm as a function of the number of processors and the grid resolution G, for the first example. This study is expected to be useful in estimating the performance of the algorithm when more processors are used, and on machines with different architectures. The following conclusions can be drawn by examining Tables 4.2 and 4.3.

1. The overall parallel efficiency ($100 \times T_1/(P \times T_P)$) of the algorithm is more than 80% for the first example and about 70% for the second example. For the first example, the parallel efficiency corresponding to the case which takes the minimum time for the algorithm is about 85%.

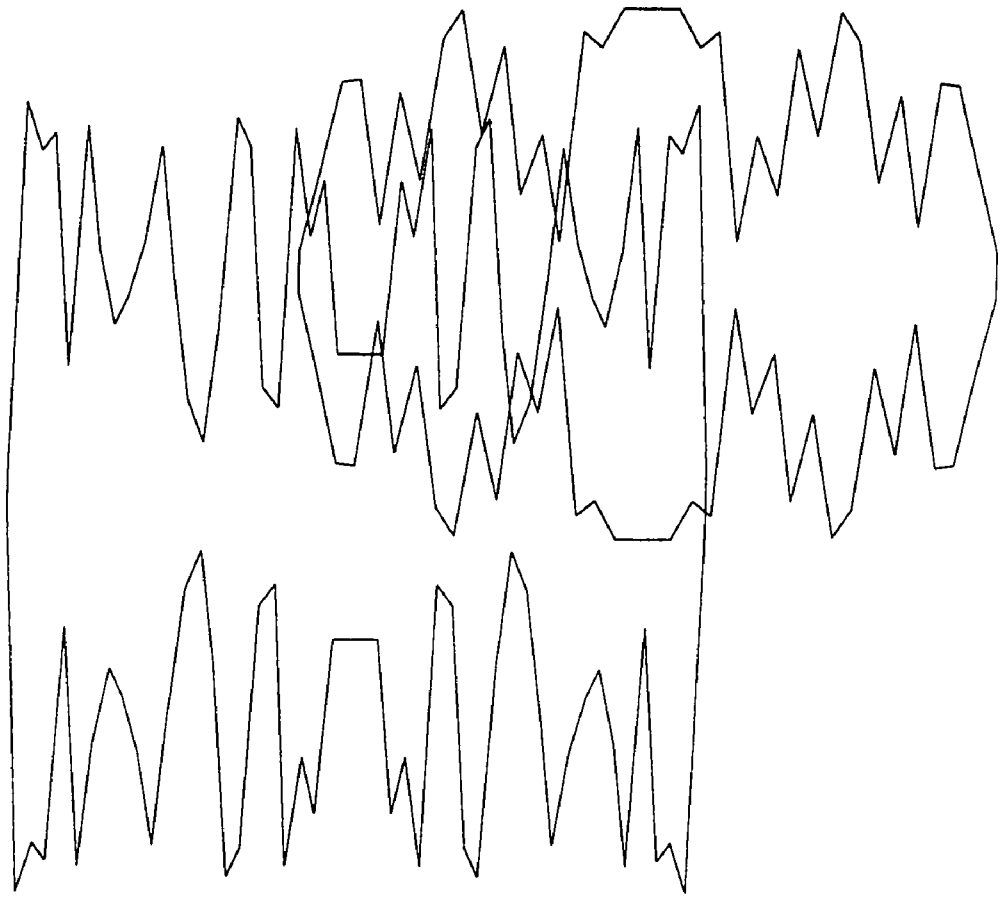


Figure 4.4: Data Set: Randomly Generated Edges

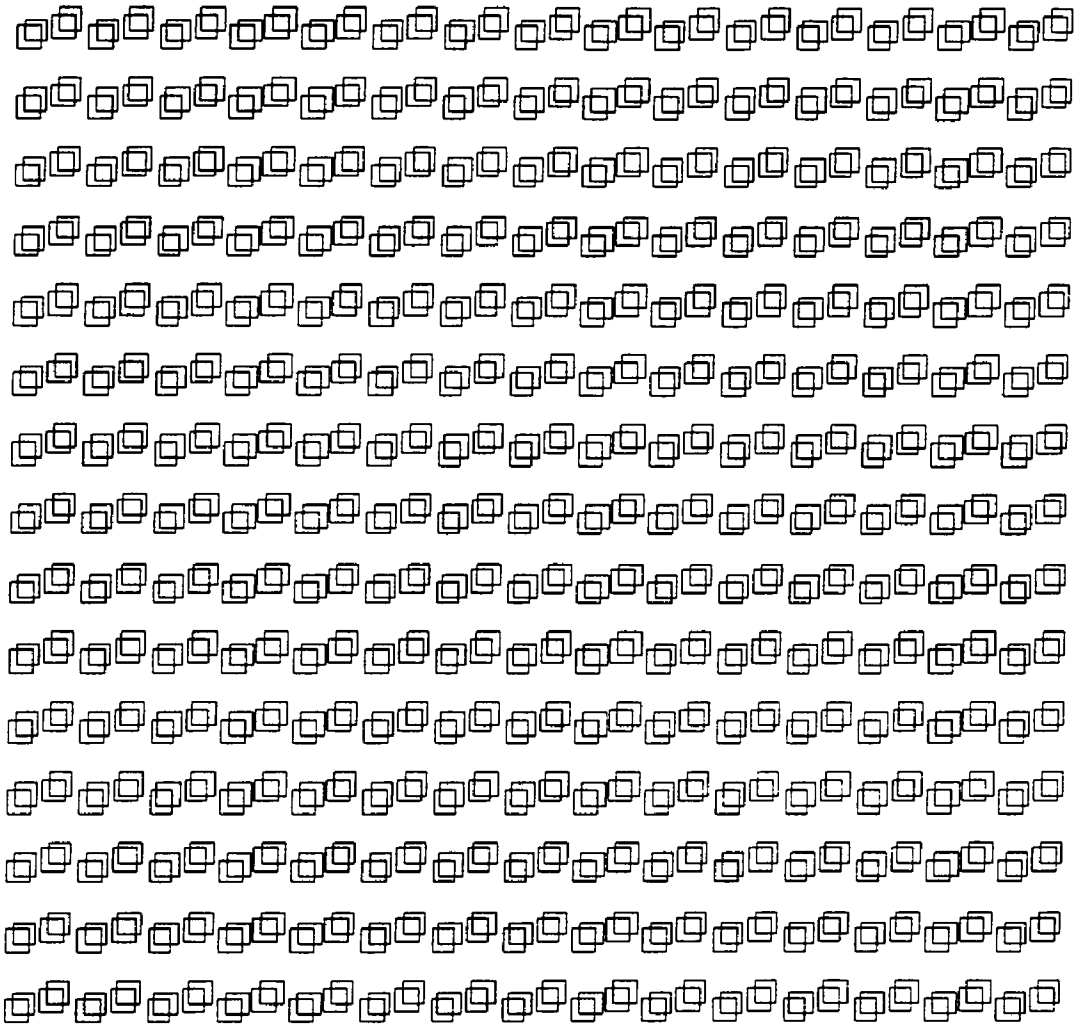


Figure 4.5: Data Set: Uniform Set of Squares

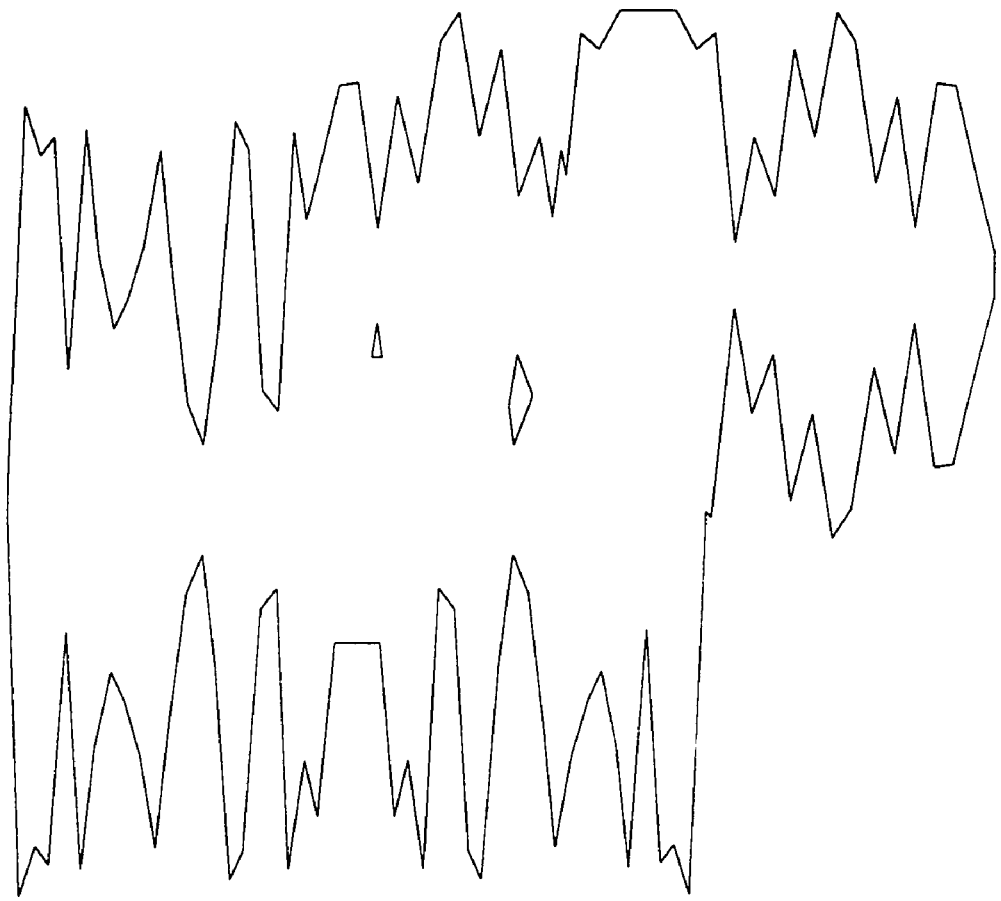


Figure 4.6: Results: Union of Randomly Generated Polygons

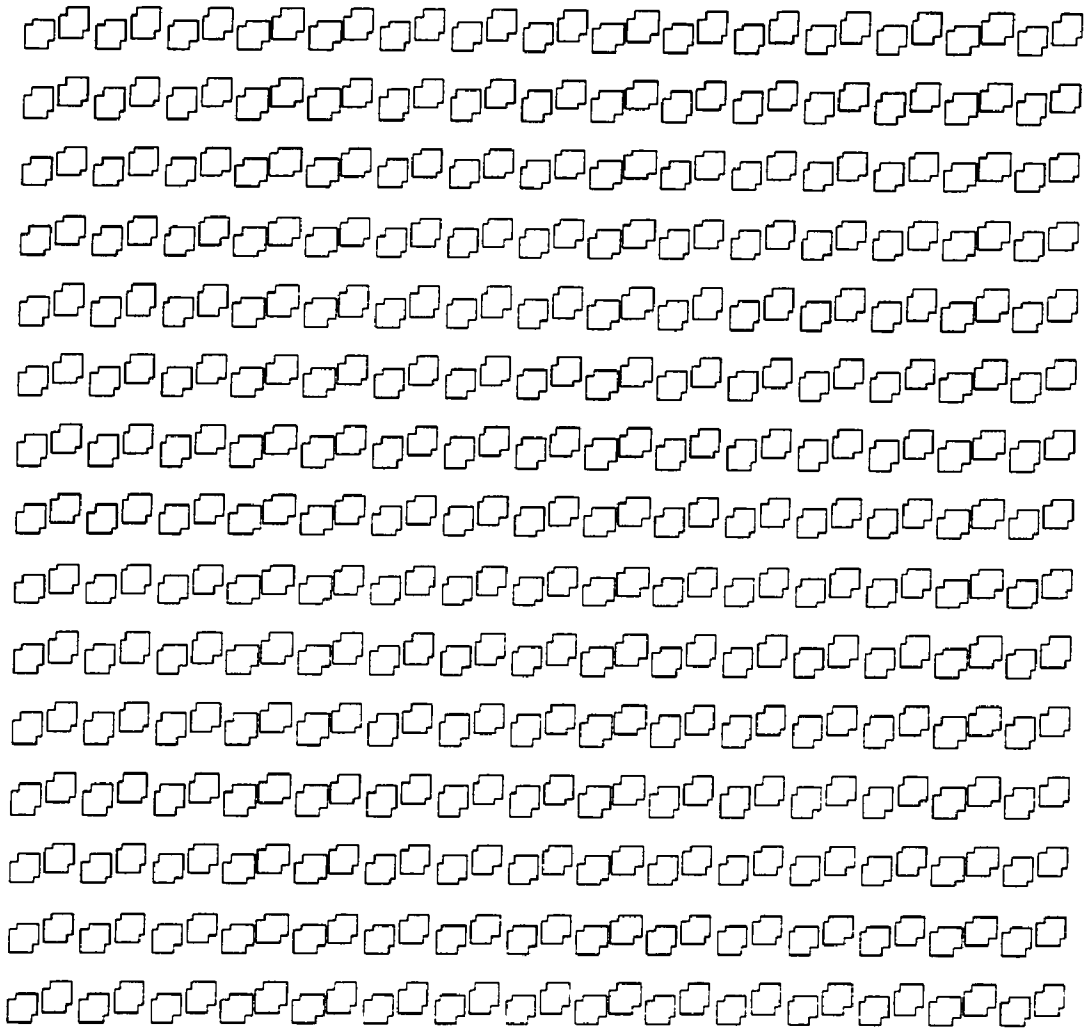
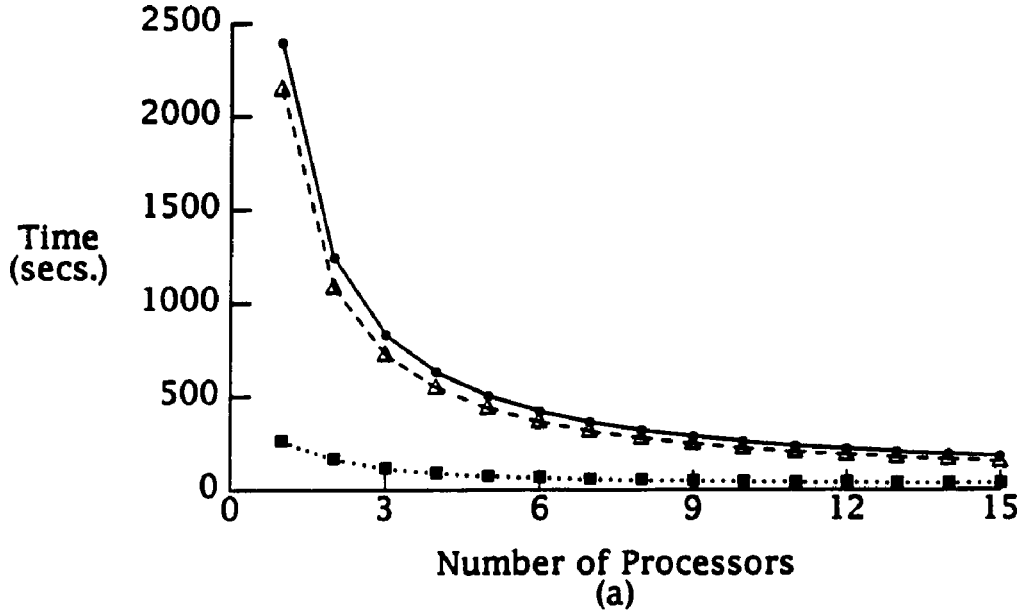


Figure 4.7: Results: Union of Sets of Uniform Squares



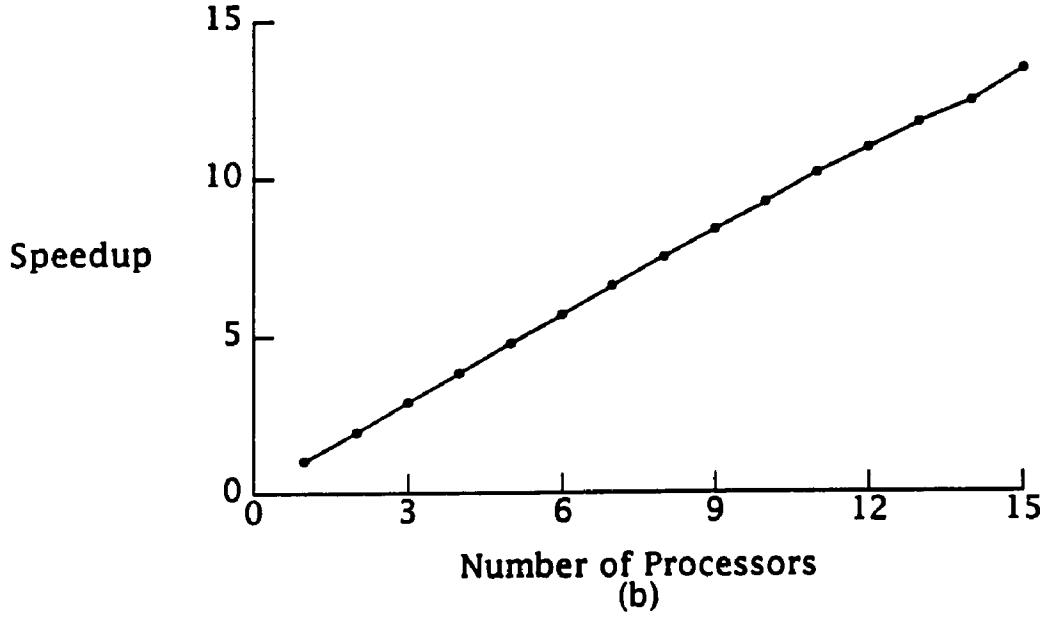
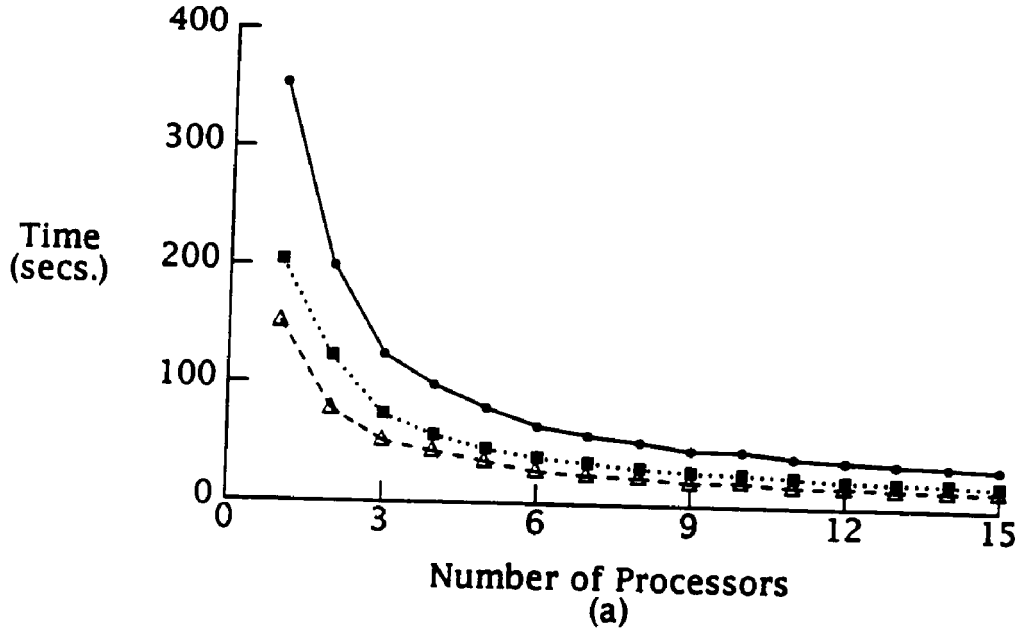


Figure 4.8: Timing and Speedup: Randomly Generated Polygons



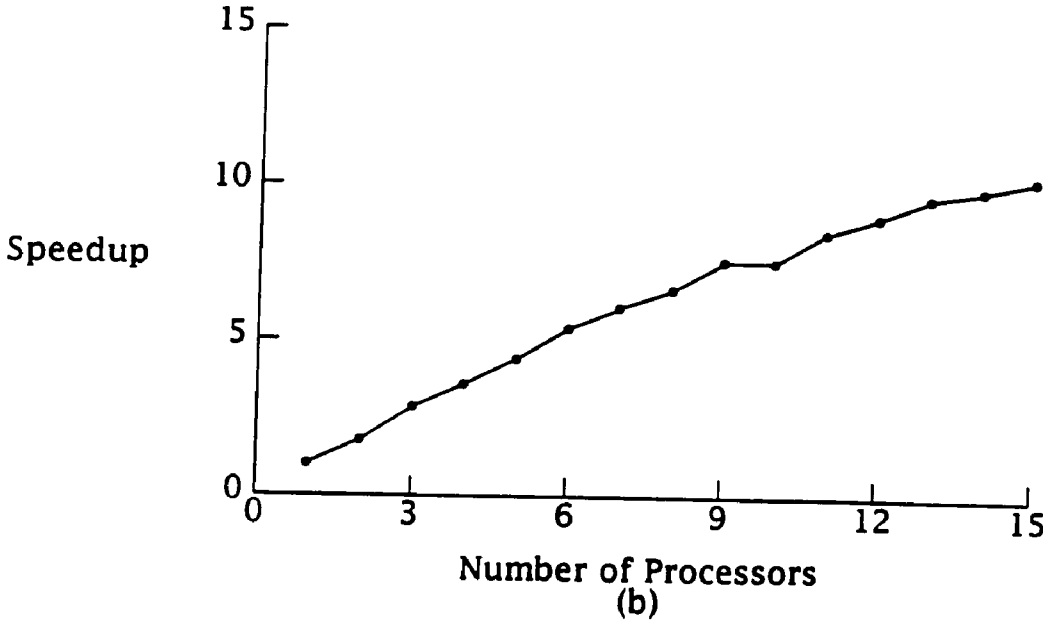


Figure 4.9: Timing and Speedup: Uniform Squares

Table 4.2: Complexity of Individual Phases of the Polygon Combination

Operation	Time 1 proc	% time	Time 15 procs	% time	Speedup
Random Polygon Example, Grid Size = 97					
Putting edges in the grid	39.82	1.66	8.48	4.79	4.70
Intersecting the edges	2101.27	87.83	141.34	79.77	14.87
Sorting the intersections	4.38	0.18	4.04	2.28	Sequential
Forming sub-edges	32.86	1.37	4.85	2.74	6.78
Classifying uncut edges	214.13	8.96	18.57	10.48	11.53
Complete polygon combination	2392.46	100.00	177.19	100.00	13.50
Square Example, Grid Size = 20					
Putting edges in the grid	7.96	2.25	4.64	13.52	1.72
Intersecting the edges	143.14	40.44	10.93	31.84	13.10
Sorting the intersections	2.48	0.70	2.48	7.22	Sequential
Forming sub-edges	23.06	6.52	3.28	9.55	7.03
Classifying uncut edges	177.30	50.09	13.00	37.87	13.64
Complete polygon combination	353.94	100.00	34.33	100.00	10.31

2. The intersection and the edge classification phases account for a large fraction of the total time of the algorithm. Both these phases show very good speedup (more than 10) and this is the reason for a respectable overall result. Thus, our framework is successful in parallelizing the main phases of algorithm and in obtaining reasonable performance.

3. The performance of the phase which determines the *(cell, edge)* tuples tends to saturate as the number of processors is increased. The benefit accrued by using more processors is negated by the overhead of the locking routines used to resolve collisions in accessing the shared cell data structure. As shown later in the polyhedron combination algorithm (Section 5.4.5), better speedup for this phase is achievable by using temporary local buffers, which are later copied into the shared cell-entity list, and then sorting the cell-entity list by the cell number with a parallel bucket sort.

Table 4.3: Variation of Time with Processors and Grid Resolution

Speedup = T_1(one proc.)/T_P(P procs.). Parallel Efficiency = $100 \times T_1/(P \times T_P)$

# Procs	Cast Grid	Intersect Edges	Sort Intersections	Form Sub-edges	Classification Time	Total Time	Speed up	Parallel Efficiency
				Grid Size = 60				
1	28.32	2171.41	4.45	33.25	340.94	2578.37	1.00	100.00
2	15.05	1093.17	4.39	18.31	222.79	1353.71	1.90	95.23
3	10.86	776.46	4.44	12.34	146.31	950.41	2.71	90.43
4	8.71	550.99	4.21	9.67	110.78	684.36	3.77	94.19
5	7.53	447.69	4.15	8.06	89.51	556.94	4.63	92.59
6	6.94	384.98	4.16	7.10	72.75	475.93	5.42	90.29
7	6.47	313.85	4.18	6.19	62.12	392.81	6.56	93.77
8	6.28	275.70	4.23	5.64	55.54	347.39	7.42	92.78
9	6.16	258.10	4.19	5.46	49.09	323.00	7.98	88.70
10	6.12	232.83	4.16	5.21	44.05	292.37	8.82	88.19
11	6.15	199.40	4.13	4.84	40.91	255.43	10.09	91.77
12	6.21	193.79	4.17	4.76	37.25	246.18	10.47	87.28
13	6.28	173.74	4.18	4.77	35.05	224.02	11.51	88.54
14	6.44	157.78	4.19	4.67	33.27	206.35	12.50	89.25
15	6.64	148.16	4.15	4.76	29.80	193.51	13.32	88.83
				Grid Size = 80				
1	34.60	2146.96	4.44	32.88	258.69	2477.57	1.00	100.00
2	18.47	1070.25	4.42	18.04	165.05	1276.23	1.94	97.07
3	13.28	717.01	4.26	12.26	109.78	856.59	2.89	96.41
4	10.60	566.61	4.13	9.57	82.55	673.46	3.68	91.97
5	9.14	478.04	4.08	7.94	67.42	566.62	4.37	87.45
6	8.36	360.17	4.06	7.10	55.73	435.42	5.69	94.83
7	7.70	307.78	4.08	6.29	47.26	373.11	6.64	94.86
8	7.41	294.87	4.08	5.80	41.00	353.16	7.02	87.69
9	7.21	239.71	4.07	5.54	37.42	293.95	8.43	93.65
10	7.10	247.11	4.07	5.24	33.00	296.52	8.36	83.55
11	7.08	196.57	4.09	5.19	30.36	243.29	10.18	92.58
12	7.10	188.60	4.10	5.20	27.57	232.57	10.65	88.78
13	7.14	166.94	4.09	5.19	25.45	208.81	11.87	91.27
14	7.24	154.97	4.06	4.97	24.11	195.35	12.68	90.59
15	7.42	159.21	4.11	5.00	22.27	198.01	12.51	83.42
				Grid Size = 100				
1	40.82	2121.41	4.35	32.84	214.60	2414.02	1.00	100.00
2	22.36	1065.00	4.35	18.05	133.46	1243.22	1.94	97.49
3	16.19	708.56	4.27	12.30	90.03	831.35	2.90	97.19
4	12.96	543.30	4.15	9.37	67.40	637.18	3.79	95.11
5	11.18	452.74	4.14	7.92	54.55	530.53	4.55	91.38
6	10.23	354.66	4.17	6.81	45.33	421.20	5.73	95.92
7	9.42	305.27	4.03	6.25	38.51	363.48	6.64	95.27
8	9.01	270.67	4.05	5.79	34.93	324.45	7.44	93.39
9	8.73	237.84	4.08	5.48	29.84	285.97	8.44	94.18
10	8.54	226.21	4.04	5.28	26.55	270.62	8.92	89.57
11	8.48	194.07	4.06	4.92	25.03	236.56	10.20	93.15
12	8.46	181.44	4.09	4.97	22.19	221.15	10.92	91.34
13	8.46	164.64	4.03	4.91	20.87	202.91	11.90	91.89
14	8.53	152.61	4.03	4.65	20.02	189.84	12.72	91.21
15	8.71	151.82	4.10	4.83	18.42	187.88	12.85	86.01

4. When just one processor is used, the sorting phase which transforms the algorithm from the domain of edges to the domain of sub-edges takes only a small fraction (less than 1%) of the total time. Hence the use of a sequential sorting algorithm for this phase does not degrade the performance significantly when 15 processors are used. However, as the number of processors increases, even small sequential sections (such as 1%) in the algorithm limit the maximum parallel efficiency achievable, and the use of a parallel sorting algorithm will be mandatory. To improve the performance, the polyhedron combination algorithm (Section 5.4.5) makes extensive use of parallel sorting schemes for the domain transformation steps.

5. The time taken by the grid, intersection, and classification phases varies with the grid size. The optimal grid size depends on the exact nature of the dependence of each of these phases on the grid resolution. In the sequential algorithm for segment intersection, the opposing nature of this dependence for the grid and intersection phases was the reason for the relative insensitivity of the total time as a function of the grid size [57]. A similar behavior is observed in the parallel polygon combination algorithm. In addition, since the various phases of the algorithm yield different speedup curves, the grid size which minimizes the total time can differ with the number of processors used. Table 4.3 shows that when either 5, 8, or 15 processors are used, the 60×60 grid is faster than the 80×80 grid. On the remaining occasions, the 80×80 grid is faster.

We expect the current implementation of our algorithm to show comparable results on other types of data sets if the sequential algorithm spends large fractions of time in the intersection and classification phases.

4.6 Conclusions

A parallel Boolean combination algorithm for polygons was presented. The algorithm combined the use of the uniform grid technique, a sort, and data partitioning for parallelization.

The uniform grid technique generated sub-tasks that could be done in parallel, reduced the number of comparisons for determining the intersections, and reduced the time for classifying points with respect to a polygon.

The sorting phase enabled the transformation from the domain of $(e_i, e_j, x, y, type\_of\_xsect)$ tuples to the domain of sub-edges. We did not use a parallel sorting algorithm for this purpose in the implementation because sorting did not form a significant part of the complete algorithm.

The implementation revealed that the determination of the intersections between the edges of the polygons and the classification of the edges in the resulting polygon were the most time-consuming phases of the geometric aspects of the Boolean operation algorithm for polygons. The good speedup (more than 10 with 15 processors) achieved indicates that the simple data partitioning scheme which distributed the edges, cells, and the sub-edges among the processors did not cause any noticeable load-balancing problems for the examples investigated.

As we show in the next chapter, the Boolean combination algorithm for polyhedra makes more extensive use of this paradigm.

CHAPTER 5
PARALLEL POLYHEDRON COMBINATION

The framework presented in Chapter 2 is now utilized for developing a parallel algorithm for *polyhedron combination.* This problem is more complex than the ones encountered so far. It involves several important geometric and topological issues.

5.1 Introduction

The problem of polyhedron combination in the context of this chapter is as follows: Given the *boundaries* of two polyhedra, we need to determine the boundaries of the various regularized Boolean combinations of the two polyhedra such as the intersection, union and difference. The term *boundary*, as used in this chapter, and the restrictions of our algorithm are elaborated in Sections 5.4.1 and 5.3.1, respectively.

Polyhedral Boolean operations are used to define objects in the Constructive Solid Geometry (CSG) representation scheme, detect spatial interferences and collisions, model manufacturing processes such as milling and drilling, model integrated-circuit fabrication processes, and for path planning in robotics.

Thus, it is clear that Boolean operations on polyhedra are useful in diverse areas. A parallel algorithm for this problem would be useful to all these areas.

The following sections provide a review of existing sequential Boolean operation algorithms, present a parallel algorithm for the problem, and discuss the implementation and results on a shared-memory parallel machine.

5.2 Literature Review

Though many independent sequential algorithms have been developed for this problem, most are minor variations of the major approaches. Requicha and Voelcker [67] present an excellent overview of the various techniques used for performing the Boolean operations. The following are some of the more recently developed algorithms which are representative of the different approaches to the problem.

Segal and Sequin [71] consider the problem of partitioning polyhedral objects into nonintersecting parts. *Face-face* intersection forms the core of their algorithm. Bounding boxes are used to reduce the number of comparisons. In a similar algorithm, Laidlaw and Trumbore [49] present a detailed case-by-case analysis of the special cases. Their method is applicable only to polyhedral objects with convex faces. One drawback is that this domain is not closed under Boolean operations, and hence faces have to be fragmented in order to make the domain of the result the same as that of the input.

Thibault and Naylor [74] use binary space partitioning (BSP) trees to perform set operations on polyhedra. They consider the problem of generating BSP trees for polyhedra and for polyhedral objects defined in CSG. One problem with this method is the possible excessive fragmentation of faces by the halfplanes. Another disadvantage is that their algorithm depends on sequential insertion of hyperplanes into the BSP tree, and is thus hard to parallelize.

Carlbom [18] presents an algorithm for set operations on planar polyhedral manifold objects. A cellular space subdivision scheme and the polytree data structure are used to efficiently search for intersections. The cells are subdivided till they become simple, i.e., contain only an edge or vertex or face of the object. Objects are clipped against the boundaries of the cells. This is not desirable because clipping

against the cell boundaries is expensive. Clipping could also lead to numerical precision problems when the solutions from the various cells have to be merged at the boundaries of the cells.

Yamaguchi and Tokieda's approach [79, 80] triangulates the potentially intersecting faces. The basis for their method is that Boolean operations on triangles are easy to perform. They propose a 4×4 determinant processor to handle the geometric queries encountered in this problem. A series of triangulation and detriangulation of faces is done in the course of the algorithm, and hence the mapping of data on to the triangle processors, such that the load is evenly balanced, is not trivial.

Putnam and Subrahmanyam [63] consider Boolean operations on n-dimensional objects. Their representation treats all entities as objects and makes no conceptual distinction between faces, edges, and vertices. Their algorithm is recursive, the recursion being on the dimension of the objects dealt with at every stage. Their algorithm seems extremely well-suited to object-oriented programming.

We are not aware of any implementations of the last three algorithms and hence it is difficult to comment on their performance.

Mäntylä [52] describes a *topological* approach to the set operation problem, which maintains all topological relations throughout the algorithm. This makes it possible to make consistent decisions while handling special cases when the arithmetic on the computer is not precise enough to make reliable and consistent decisions by the use of geometry alone. However, it also makes the algorithm very difficult to parallelize.

Hoffman, Hopcroft, and Karasick [44] describe an algorithm for performing regularized set operations on polyhedral solids. They use symbolic reasoning as a supplemental step that compensates for possible numerical uncertainty in order to make their algorithm robust. A more detailed description is given by Karasick

in [46].

All the above algorithms are sequential, although some of them could be parallelized. The authors are not aware of any parallel object-space algorithm for this problem. The parallel algorithm developed in this chapter is a parallelization of the sequential algorithm presented by Franklin [30]. Also, the algorithm presented here discusses the topological aspect of the Boolean operation algorithm in much greater detail than [30].

5.3 Important Algorithmic Issues

The Boolean operation algorithm has to deal with two issues, *geometry* and *topology*. The geometric aspect deals with obtaining all the intersections between the two objects. The topological aspect deals with maintaining the connectivity relations in the result. The maintenance of complete topology for the result of the Boolean combination is of prime importance in some applications. It is not necessary in many applications such as interference detection, and determination of visible surfaces and mass properties.

Many algorithms just deal with the geometric aspect. Some algorithms like the one presented here, treat these aspects in separate phases. The pros and cons of this approach will now be examined from the point of view of parallel processing.

Two fundamental and important problems arise in the polyhedron combination problem which can cause these algorithms to fail in some cases. The first one is due to the occurrence of special cases. The second one is due to the limited-precision arithmetic available on computers. In practical situations both problems occur in conjunction.

When consistent topological results are needed, the two aspects cannot be dealt with in isolation because special cases such as vertices lying on edges or faces, or edges lying on edges or faces, may produce duplicate intersections in the

geometric phase. These cases have to be identified and appropriate action has to be taken to have a valid topological result.

By the above reasoning, it can be concluded that the processing of a special case requires confirmation of independently made geometric decisions, and is not an inherently parallel task.

In some special cases, due to limited numerical accuracies on computers, two independent and contradictory geometric decisions may have to be coerced into one because of topological considerations. These problems are extremely complicated issues in themselves. It is a separate research topic and an investigation of these issues is beyond the scope of this thesis. However, the following observations are worth noting.

The first observation is that though numerical problems often cause geometric programs to fail, they manifest themselves only in special situations. The second observation is that in most practical situations, there are only a *few* instances of these special cases in any single problem of polyhedron combination. Therefore, the algorithm will spend most of its time in processing the normal cases. The third observation is that general solutions to these problems are slow and should be used only when necessary.

The above issues are important both from theoretical and practical points of view. However, even a sequential solution to these problems that is complete and convincing has eluded researchers thus far.

5.3.1 Objective and Scope of the Algorithm

Taking the above considerations into account, our objective is to provide a parallel algorithm which handles *normal* cases *rapidly*. The scope of our parallel algorithm does not include treatment of problems caused by numerical errors. Our algorithm can be used in conjunction with provably robust sequential algorithms to

develop a correct and fast solution to the complete problem. Thus even a parallel algorithm that handles normal cases and just *detects* the special cases as opposed to providing a parallel solution to the complete Boolean operation problem is useful. For the purpose of detection of special cases, more conservative tolerances can be used, so that the detection strategy is robust. Each of the cases detected above can then treated by a robust sequential algorithm. The doubtful cases may of course be distributed among the processors.

Even in the hypothetical situation where perfect arithmetic is available, special cases due to coincident geometric entities require separate treatment for generating a valid result. These issues are examined in Section 5.5. In order to create such an environment, our algorithm assumes that perfect arithmetic can be simulated by resorting to use of ϵ's. (features separated by distances less than ϵ are considered to be coincident). We exclude cases where this assumption will be violated.

The disadvantage of attempting to maintain complete topology in the intermediate phases of the algorithm by using *Euler operators* or their equivalents is that it will make the algorithm sequential because the *Euler* operators appear to be inherently sequential. This is our rationale for separating the geometric and topological aspects of the algorithm into distinct phases. Thus, our algorithm determines the topology of the resulting object in a *batch mode.* The ordering of the edges in the faces of the resulting object is determined only when *all* the intersections in the face of the original object are available. The faces that are incident at each edge of the resulting object are determined when all the faces in the resulting object are known. We do not maintain complete topology in non-manifold situations. It is not necessary in many applications such as interference detection, and determination of visible surfaces and mass properties.

To maintain clarity in the main algorithm, our treatment of special cases (assuming no arithmetic errors) arising due to coincident geometric entities, is presented in Section 5.5.

5.3.2 Assumptions about Polyhedra

At this point we would like to point out that by placing the complete object within a single grid cell, it is trivial to generate an input set on which the uniform grid technique will fail to reduce the number of face-face comparisons. Hence we assume that the distribution of the faces of the object is more or less uniform throughout the space. We also assume that the data sets we handle are reasonably complicated, so as to justify parallelization. Typically, each object will contain hundreds of faces. In such a scenario, most individual faces will be small compared to the spatial extent of the scene, and it will be beneficial to use the uniform grid.

5.4 The Algorithm

5.4.1 Polyhedron Representation Scheme

The polyhedral objects used in our algorithm are represented by a list of faces. The faces are represented as a list of contours. If a face has holes, the outermost contour is specified first. Each contour consists of an ordered list of edges. The orientation of the edges in the contour is such that the interior of the face lies to the left of the edges. An edge can be adjacent to only two faces. Every edge in the face knows the other face adjacent to it. The face also has a normal vector which points towards the exterior of the object. The normals to the faces incident at an edge of the polyhedron are different, i.e., there is discontinuity of the normal at an edge. The objects themselves may be composed of multiple shells.

5.4.2 Notation

Some of the terms used in the algorithm are defined here.

1. The two polyhedra are referred to as P_a and P_b respectively.

2. F refers to the faces of the polyhedra. F_a represents a face of polyhedron P_a.

3. CS refers to a coordinate system. A 2-D coordinate system on a face F_a, of P_a, is represented by $CS(F_a)$.

4. WCS refers to the world coordinate system used to define the objects. A transformation from coordinate system $CS(1)$ to $CS(2)$ is represented as $\mathbf{T}(CS(1) \rightarrow CS(2))$.

5.4.3 Overview of the Algorithm

The parallel Boolean combination algorithm is divided into a set of data-parallel phases. These phases work with a series of tuple-sets. It first uses the uniform grid technique to reduce the number of pairs of faces which have to be tested for intersections. It then finds the intersections between these pairs of faces. The intersections are then organized and the faces of the original polyhedra are split into sub-faces. Depending on the Boolean operation being performed, appropriate sub-faces are selected to create the desired object. The parallelization is explained in the following sections.

5.4.4 Tuple-Sets used by the Algorithm

The following are some of the tuple-sets which the algorithm generates. The generation of these tuples is discussed in the following section:

1. The set of *(cell, face)* tuples. *Cell* refers to the label of the grid cell, and *face* refers to the label of the face.

2. The set of *(face, face)* tuples. It is a list of labels of potentially intersecting pairs of faces of the two polyhedra.

3. The set of *(face, cut-line, other-face)* tuples. A *cut-line* (see Figure 5.2) is a line in the plane of the *face* across part (or all) of it. The cut-lines are used to subdivide the *face* into *sub-faces.* The *other-face* is the face of the other object that intersected this face to create the cut-line.

4. The set of *(face, edge, cut-point)* tuples. *Edge* is an edge of the *face* which the cut-line intersects. A *cut-point* (see Figure 5.2) is the point on the edge where a cut-line meets it. The cut-points are used to split the edges of the faces.

5. The set of *(segment, segment-type, origin)* tuples. *Segment* could be a cut-line in the face, a complete edge of the face or a split edge of the face. *Segment-type* refers to the type of the segment. *Origin* has details about the exact origin of the segment, i.e., which edge, sub-edge or cut-line it came from.

6. The planar graph G_{psg} domain. It is used to manipulate a planar straight-line graph to determine the *sub-faces* of the faces. As Figure 5.3 shows, a sub-face is a part of a face.

7. The set of *(sub-face)* tuples. The set of faces of the polyhedra resulting from the Boolean operations are subsets of the union of this set of sub-faces and the set of original faces of the object. Every sub-face knows the polyhedron it came from, and the origin and other details of its edges.

8. The set of *(edge, sub-face)* tuples. *Edge* stores the identifier of the edge, its type, i.e., whether it is an uncut edge of a face, cut-line on a face, or a split

edge of a face, and has details about its origin and the actual coordinates. *Sub-face* is the identifier of the sub-face to which the edge belongs.

5.4.5 Detailed Algorithm

1. Establish a 2-D coordinate system, $CS(F_i)$, for each face-plane F_i. This will be used later for referring to the points on the plane and for routines such as point-in-polygon testing. For efficient computation in later stages, transformation matrices $\mathbf{T}(CS(\text{WCS}) \rightarrow CS(F_i))$ and their inverses are determined for converting between 3-D and 2-D and vice-versa. In subsequent sections, this step is referred to as the *domain transformation step.*

   The face-normals are also calculated at this point, if they are not already given.

2. Calculate an efficient grid resolution G for the grid to be cast over the polyhedra. The surface area of the cross-section of a grid cell should be comparable to the average area of the faces in the polyhedra, so that the average face goes through only a few cells.

3. For each face in either polyhedron, determine the cells it passes through, and add the elements to the appropriate set of *(cell, face)* tuples. A face is entered in every cell cut by the portion of the plane of the face contained within the bounding box of the face. We do not use an extension of the Bresenham line algorithm [16] for this step because, for complex faces it appears to be more complicated than our method.

   The above three steps are done in parallel by distributing the faces equally among the processors. The fairness of the distribution of load among the processors depends on the complexity of the faces.

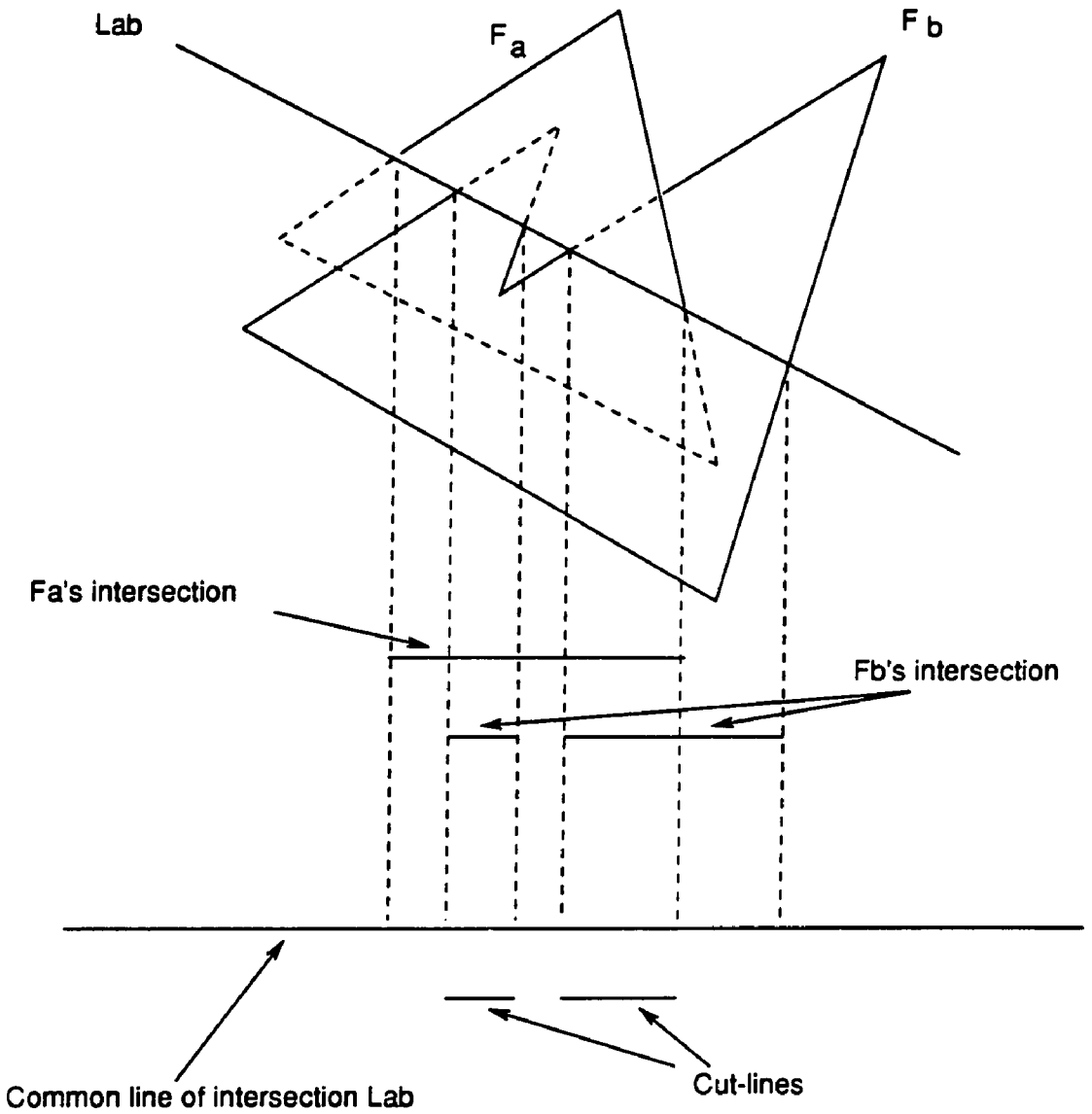


Figure 5.1: The Intersection of Two Faces F_a and F_b

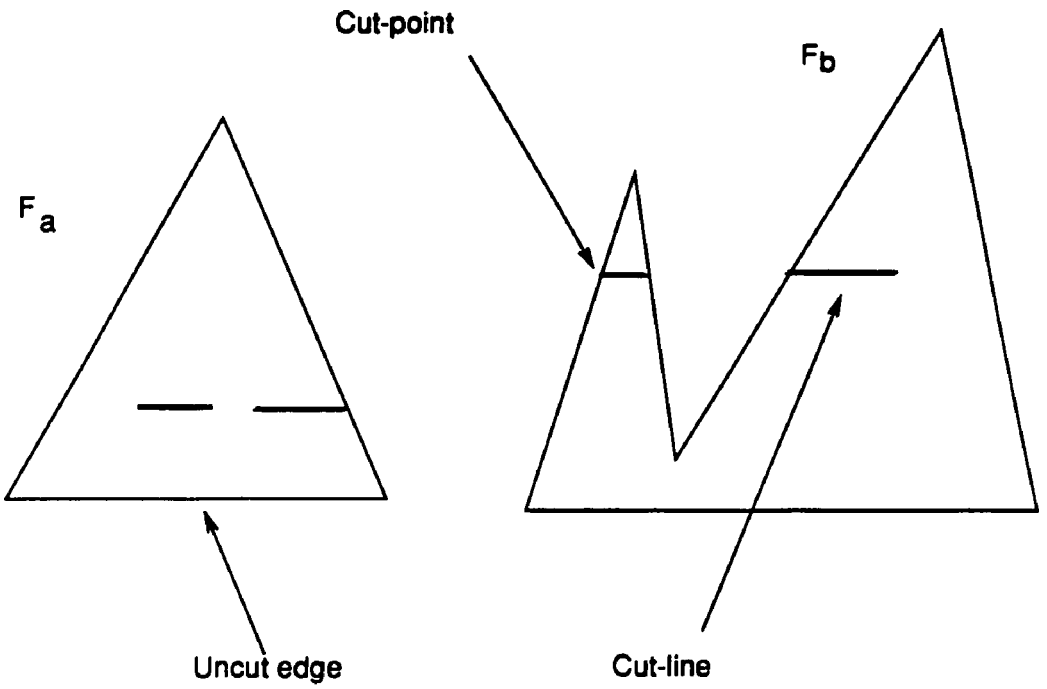


Figure 5.2: Cut-Lines and Cut-Points of Faces F_a and F_b. Each face has two cut-lines due to mutual intersection. F_a has one cut-point and F_b has three cut-points.

4. Use the parallel bucket sort discussed in Section 2.3.1 to sort the two sets of *(cell, face)* tuples by cell number, so that all faces passing through the same cell are in consecutive locations of the set of *(cell, face)* tuples.

5. Using the two sorted lists of *(cell, face)* tuples form the list of *(face, face)* tuples of potentially intersecting faces. This step is done in parallel by distributing the *(cell, face)* tuples among the processors, such that each processor deals with a few cells. This list of *(face, face)* tuples can have duplicates, because the same pair of faces may go through more than one cell. Since it is expensive to intersect pairs of faces and to detect multiple intersections between the same pair of faces, computed in different cells potentially by

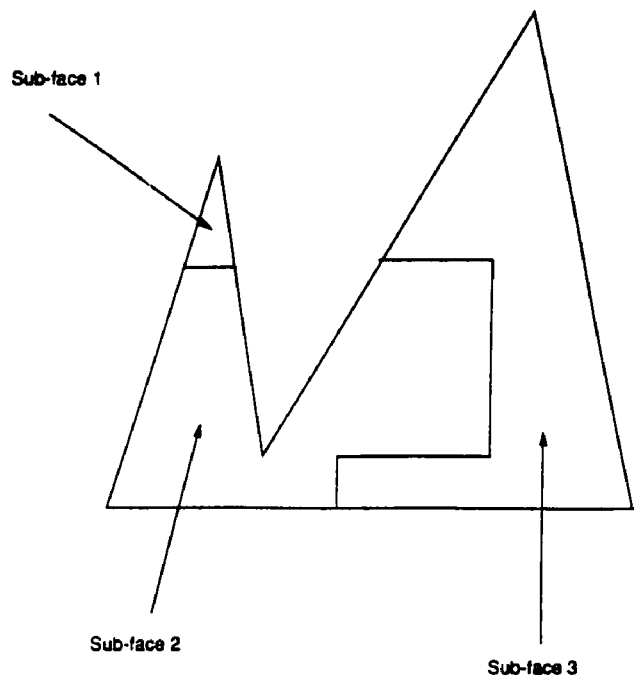


Figure 5.3: Sub-Faces of Face F_b

different processors, at a later stage, this list is pruned at this point to eliminate duplicates. The list is pruned in parallel by converting the linear list into a hash table. Duplicate entries are removed by checking for a previous entry at the time of insertion into the hash table. This step is parallelized by distributing the *(face, face)* tuples among the processors. *Write* conflicts are resolved by the use of locks. The hash table is then reinverted in parallel, by distributing the buckets among the processors, into a linear list which contains no duplicates.

Alternatively, the set of *(face, face)* tuples can first be sorted by the *face* in parallel, and the duplicates can then be removed by scanning the sorted list. The implementation shows that there is very little difference between the time taken by these two methods.

6. For each tuple (F_a, F_b) in the set of *(face, face)* tuples, determine whether F_a and F_b intersect. If they do, determine the cut-lines and the cut-points. Figure 5.1 illustrated the determination of cut-lines and cut-points. Figure 5.2 shows the cut-lines for F_a and F_b. Since this is the most important step in the whole algorithm, where the final geometry is determined, it is given in considerable detail.

   (i) Calculate the line of intersection L_{ab} between the planes of the faces F_a and F_b. We assume that the faces are not coplanar. See Section 5.5.3 for handling of coplanar faces.

   (ii) Establish a 1-D coordinate system $CS(L_{ab})$ on L_{ab}, and calculate the matrices $\mathbf{T}(CS(L_{ab}) \rightarrow CS(F_a))$ and $\mathbf{T}(CS(L_{ab}) \rightarrow CS(F_b))$ to switch between it and the 2-D coordinate systems on the planes of F_a and F_b.

   (iii) For both faces, determine the parts of the line L_{ab} that fall within them. This step involves intersecting L_{ab} with all the edges of the face.

   (iv) Convert the two included ranges of L_{ab} back to its 1-D coordinate system, $CS(L_{ab})$.

   (v) Intersect the ranges of the two faces to get the common range. If the common range is empty, the two faces do not intersect. In this case the algorithm proceeds with the next *(face, face)* tuple. If the faces do intersect, mark them as *cut*. Note that we have reduced the dimension of the problem from 3-D to 2-D, and it makes this step faster.

   (vi) Convert the common range back to the coordinate system of each face. Each interval is a *cut-line* of the face. Each cut-line, and the pair of faces, are added to the set of *(face, cut-line, other-face)* tuples. Each cut-line is

oriented in the plane of the *face* such that the outward normal of the *other-face* causing it is to its right, so that sub-face classification can be done by a simple inspection of the cut-line.

(vii) Each endpoint of an interval of a common range is on an edge of either F_a or F_b (or both if the edges intersect). The face, edge, and the endpoint are added to the set of *(face, edge, cut-point)* tuples.

(viii) Process the next *(face, face)* tuple.

Step 6 is done in parallel, by distributing the elements of the set of (face, face) tuples equally among the processors. Write conflicts into the sets of (face, cut-line, other-face) and (face, edge, cut-point) tuples are resolved by the use of locks. In order to minimize the probability of many processors queuing on the same lock, processors maintain short local lists and update the global list when their local lists get filled up.

7. Sort in parallel, using a bucket sort, the set of *(face, cut-line, other-face)* tuples by face number, so that all the cut-lines for a face are in consecutive locations of the set of *(face, cut-line, other-face)* tuples.

8. Sort in parallel, using a bucket sort, the set of *(face, edge, cut-point)* tuples by face (major key) and edge (minor key) so that all the cut-points of every edge in each face are in consecutive locations of the set of *(face, edge, cut-point)* tuples.

9. For each face F, in the list of faces intersected by the other object, do the following to determine the sub-faces into which it should be partitioned. Figure 5.3 shows the sub-faces for the face F_b shown in Figure 5.1.

   (i) For each edge E of F, determine the cut-points on it from the set of *(face, edge, cut-point)* tuples, sort them along E, and use them to

partition E into segments. Add information about these segments to the set of *(segment, segment-type, origin)* tuples for the current processor.

(ii) From the set of *(face, cut-line, other-face)* tuples, determine the cut-lines on F and add them to the set of *(segment, segment-type, origin)* tuples for the current processor.

(iii) Add all edges of F which were not split by the intersection process to the set of *(segment, segment-type, origin)* tuples for the current processor.

(iv) Order the set of *(segment, segment-type, origin)* tuples to define a planar graph in the plane of F. Its regions are the sub-faces of F. Determine the vertices and edges in order around the perimeter of each sub-face. This step is done by a generalized sub-face reconstruction algorithm discussed in Section 5.4.5.1.

(v) Determine whether each sub-face is inside or outside the other polyhedron, by inspecting a cut-line in the sub-face. Though holes in faces are treated as sub-faces of the original face, no cut-line will be found in any contour whose interior is a hole, and hence in these cases the sub-faces will rightly not be classified at all.

(vi) For each sub-face, add an appropriate element to the set of *(sub-face)* tuples.

Step 9 is done concurrently by distributing the faces among the processors. At the start of this step, each processor also determines the (face, edge, cut-point) and (face, cut-line, other-face) tuples for the faces it is responsible for, by a binary search followed by a sequential pass through the respective lists. Each processor has its own data structures to manipulate the planar graph, and hence this step has no write conflicts.

10. Use a depth-first traversal to propagate the sub-face classification across the edges to the uncut faces of the solid.

    The above step is done in parallel by distributing the sub-faces equally among the processors. Each processor starts a depth-first traversal.

11. The classification of faces in shells which do not intersect at all is determined by performing a point-in-polyhedron test. An efficient point-in-polyhedron procedure is discussed in a following section.

    The above step is done in parallel by distributing the unclassified faces equally among the processors.

12. With the union of the set of sub-faces determined by all the processors, the faces to be included in any of the Boolean combinations of the polyhedra P_a and P_b can be determined using Table 5.1.

13. For each edge from each of the contours in the resulting object, enter its sub-face label into the set of *(edge, sub-face)* tuples. *Edge* contains identifier information and details about the origin of the edge, i.e., whether it was due to a cut-line, a split edge of a face, or a whole edge of a face in the original object.

    The edges of the contours in the resulting object are equally distributed among the processors. Write conflicts are resolved by the use of locks.

14. Sort the *(edge, sub-face)* tuples by the identifier of the *edge*, using a parallel bucket sort. Identify groups of split-edges whose *origin* information is the same. Sort them by coordinates of the vertices of the edges. This process ensures that consecutive pairs of *(edge, sub-face)* tuples refer to the same edge element in space, though they may have resulted from different sub-faces and may have been determined by different processors.

15. Consecutive pairs of tuples of the set of *(edge, sub-face)* tuples now have the face-around-edge adjacency information.

    The edges of the contours in the resulting object are equally distributed among the processors.

5.4.5.1 Reconstructing the Topology of Sub-Faces

The algorithm presented here is an extension of the one given by Franklin and Akman [32]. The main enhancement here is that graphs with multiple components are also handled.

An adjacency list representation of the graph, for the segments in the plane of the face whose sub-faces have to be determined, is created. This step is done by sorting the segments by the coordinates of their vertices and grouping them by combining vertices which are closer than a predetermined ϵ. The adjacency list of each vertex is arranged such that the edges incident on it are in a clockwise order. A depth-first search is then performed to identify the different connected components in the graph.

The contours in each connected component are determined by traversing the graph in the following manner. A *starting vertex* is chosen for the first contour in each component. The *starting vertex* is made the *current vertex*. The first unused edge in the adjacency list of the *current vertex* is made the *current edge*. The *other vertex* of the *current edge* is made the next vertex in the contour. The *current edge* is now marked as used, and the *other vertex* is made the *current vertex*. The first unused edge in the adjacency list of the *current vertex* which follows the *current edge* (in a circularly wrapped fashion) is made the new *current edge*. This process is repeated till the contour closes itself by returning to the *starting vertex*. The remaining contours are determined in a similar fashion. A simple inspection shows that since the edges around a vertex were arranged in a clockwise fashion, all the

contours will be oriented in an counter-clockwise fashion except the outermost contour of each component which will be clockwise. The outermost contour of every connected component is marked with a special tag, in order to efficiently find the hierarchical nesting of the components which is needed to determine the contours for multiply connected sub-faces. In order to form a nested hierarchy of the connected components, the connected components are sorted by the area of the outermost contour in the component. Then for each component, starting with the smallest component, the algorithm finds the next smallest component which contains it. In each pass of this phase, at least one component will be position in its final place in the hierarchy. Bounding boxes for the components are used as a preliminary check to avoid rigorous component containment checks when they are not necessary. A component containment is detected conclusively by determining the location of a point on one component with respect to the outermost contour of the other component. If the point is contained in the outermost contour of the component, the remaining contours in the component are tested in order to place the contained component in the appropriate contour of the containing component. A point-in-contour test is similar to the point-in-polyhedron test discussed in the next section. Note that since we are dealing with disjoint connected components, no point to be classified lies on the boundary of the contour to be classified against. This feature makes the point-in-contour procedure simpler.

Every contour in each connected component, which does not enclose another connected component, is a simple sub-face of the original face. If a contour encloses components, then the outermost contour of each of the contained components becomes an inner contour for the sub-face. The sub-faces are then classified by a simple inspection of the cut-lines in their contours.

5.4.5.2 An Efficient Point w.r.t Polyhedron Procedure

The uniform grid is used to perform the point-with-respect-to-polyhedron test efficiently. In order to classify a point, shoot a semi-infinite ray in the positive z direction. The z direction is chosen as opposed to other directions because grid cells that are neighbors in this direction are stored in contiguous memory locations of the parallel machine. Determine the faces of the polyhedron which pass through the cells traversed by the ray. For each of these faces, the point of intersection between the ray and the plane of the face is checked for inclusion in the face. If the point of intersection lies in the face, the intersection is counted. If the total number of such intersections is odd, the point is inside the polyhedron. When the point of intersection between the face and the ray lies on the boundary of a face (edge or vertex), we consistently shift the ray by an infinitesimal amount, symbolically, in the negative x and y directions, so as to classify the point of intersection as either inside or outside the face. Note that the point-with-respect-to-polyhedron test does not have to handle the point-on-polyhedron case because it does not arise in our algorithm.

5.5 Treatment of Special Cases

The following section discusses the implication of some of the special cases encountered due to coincident geometric entities, to our algorithm. Our approach to handle the special cases determines the right geometry and part of the topology, but does not attempt to represent the complete topology of non-manifold situations. In this discussion we do not allow cases where the use of ϵ's leads to inconsistent decisions.

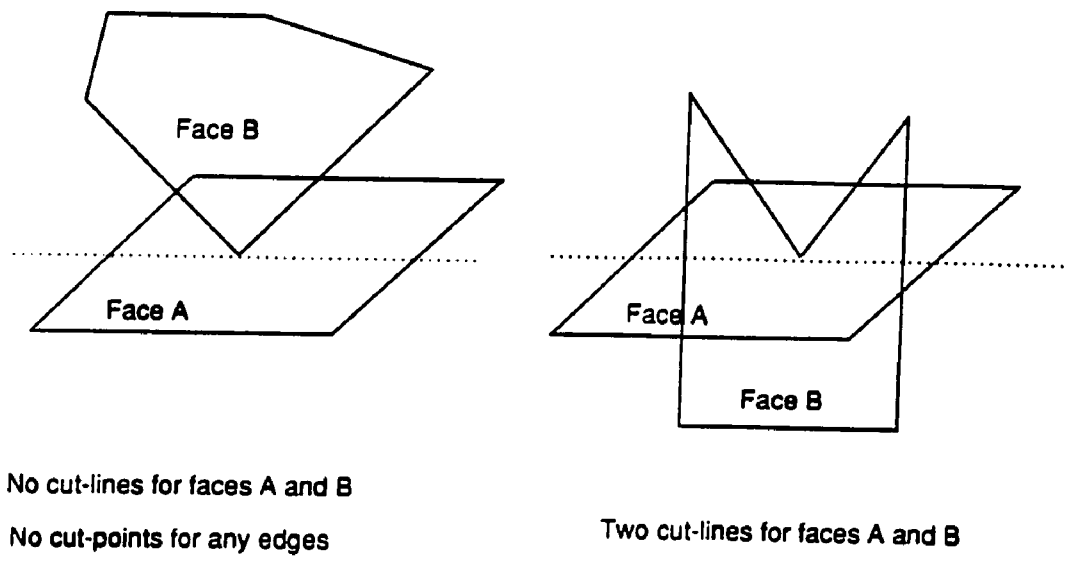


Figure 5.4: Special Case: Vertex Lying on an Edge or a Face

5.5.1 Vertices Lying on Faces or Edges

Suppose that a vertex of a face of one polyhedron lies either on the edges or a face of the other polyhedron. Consider the step where a face containing this vertex is compared with the face on which it lies. The two edges of the face that are incident on this vertex will introduce duplicate points of intersection.

In order to handle this case, the step which determines cut-lines due to the intersection of two faces needs modification. Multiple vertices on L_{ab} which are coincident are either coalesced or eliminated. Since the complete topology (ordering of edges) and the geometry (coordinates of edges) of the face are available at the time of intersection in the processor handling it, these decisions can be made by examining the edges around the vertex. Figure 5.4 shows two examples. In one case the two vertices are eliminated and in the other case they are coalesced. Isolated vertices in the interior of the faces are eliminated.

5.5.2 Edges Lying on Faces or Edges

We now describe the manner in which the algorithm handles the case where an edge of one polyhedron lies on an edge or face of the other polyhedron. The important issues that need to be re-examined with this special case are the following:

1. Topology in non-manifold situations.
2. Merging or elimination of duplicate cut-lines.
3. Propagation of sub-face classifications.

Recall that our polyhedron representation scheme does not allow more than two faces to be incident at an edge. Therefore in this case, the algorithm does not attempt recovery of the face-around-edge information in the non-manifold situations. In order to be able to maintain this information, a more complex polyhedron representation scheme such as the *radial-edge* [76] or *star-edge* data structure [46] has to be used. Our strategy still obtains the topology of the faces (the edges in the face are reported in order) in the resulting polyhedron and the faces-around-edge adjacency in all the other manifold regions of the polyhedron. This result is sufficient for determination of mass properties and for generating pictures.

Our algorithm breaks the topology of the polyhedra at the face-level and uses face-face intersection as the basic intersection operation. Therefore, when an edge that is common to two faces of one polyhedron lies on a face or edge of the other polyhedron, *each* face-face intersection test among the above faces will produce one cut-line. These cut-lines occupy the same physical location, but are duplicated because of the independent face-face intersection tests. The issue of either merging or eliminating duplicate cut-lines is resolved in the sub-face formation algorithm (Section 5.4.5.1).

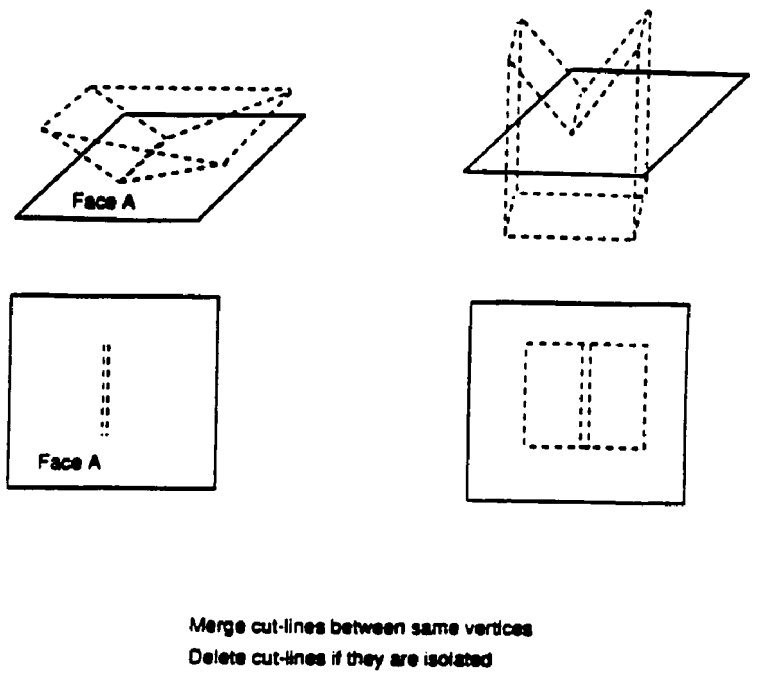


Figure 5.5: Special Case: Edge Lying on a Face

When an edge of one object lies on the face of the other object, the two cut-lines produced by the two faces adjacent to the edge are first combined into one. Then the sub-face formation algorithm removes isolated cut-lines. Figure 5.5 shows two examples. The net result of the above steps is that the two duplicate cut-lines are either coalesced or completely removed.

Similarly, when an edge of a face lies on an edge of the other face, duplicate elements between the same two vertices in the planar graph for the sub-face are combined. Figure 5.6 shows two examples.

The second modification is that the sub-face classification algorithm cannot derive valid sub-face classification from cut-lines that are created by edges lying on faces or edges. Recall from Section 4.3.2 that a similar strategy was used for the polygon combination problem to avoid erroneous classification of sub-edges when vertices were incident on edges. This constraint is enforced by the following strategy. The *(face, cut-line, other-face)* tuples generated by the face-face intersection

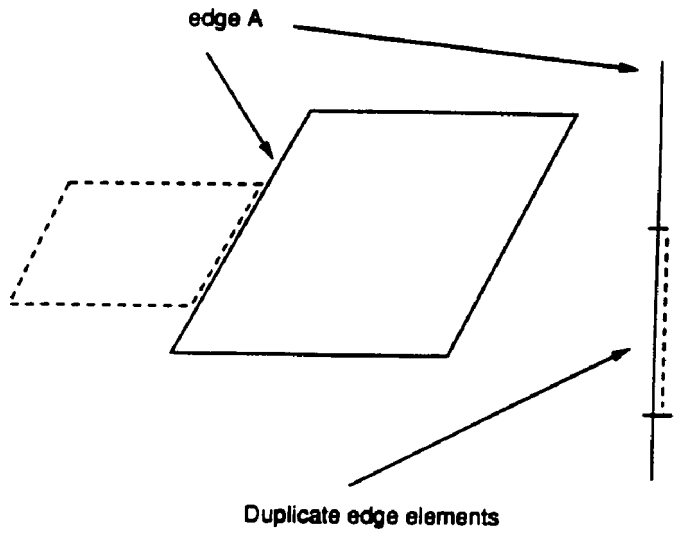


Figure 5.6: Special Case: Edges Lying on Edges

process are replaced by *(face, cut-line, other-face, type)* tuples. *Type* refers to the type of cut-line, i.e., whether it was caused by a penetration of the face or by an edge of one face lying on the other. The classification algorithm does not allow classifications to be inferred from cut-lines caused by edges of one polyhedron lying on faces or edges of the other polyhedron.

5.5.3 Faces Lying on Faces

The intersection process does not change due to the presence of coplanar faces; however, when two faces are coplanar, the fact is recorded. The faces that are transverse to the coplanar faces generate all the necessary cut-lines to partition the coplanar faces into sub-faces. This is because the normal to the polyhedron changes at an edge. The following issues have to be resolved in order to handle this special case:

1. The classification of sub-faces of faces that have been determined to be coplanar with other faces cannot be done by an inspection of a cut-line in its contour.

2. Duplicate contours which represent the same area in the plane of the coplanar faces have to be avoided in the resulting polyhedron.

Sub-faces of faces that are coplanar to other faces are classified by either just a point-with-respect-to-polygon test, or a point-with-respect-to-polygon test followed by a point with-respect-to-polyhedron test, depending on whether the coplanar contours overlap or not. The point-with-respect-to-polygon test is sufficient to detect whether the coplanar contours overlap. If the coplanar contours do not overlap, a point-with-respect-to polyhedron test is sufficient to classify the contour. It should be noted that the contours that are coplanar either overlap completely or are completely disjoint. Also, the point-with-respect-to-polyhedron test does not have to handle the point-on-polyhedron case.

When no face-around-edge topology is required, duplicate contours are avoided by simply making Table 5.1 asymmetric with respect to the two polyhedra. When complete face-around-edge topology is necessary, a contour-similarity test is needed to match the duplicate contours. This problem is different from the graph isomorphism problem because the geometry of the contours is also available. The information from these duplicate contours is then used in conjunction to derive the face-around-edge topology. An edge in a contour and its mate in the duplicate contour have the property that if one of them is a cut-line, the other is due to an edge or a split-edge. Edges in the contour that are due to cut-lines have the information about the *other-face* that created them in the intersection process. Edges that are due to *split-edges* of the faces do not have this information. Figure 5.7 illustrates the above discussion.

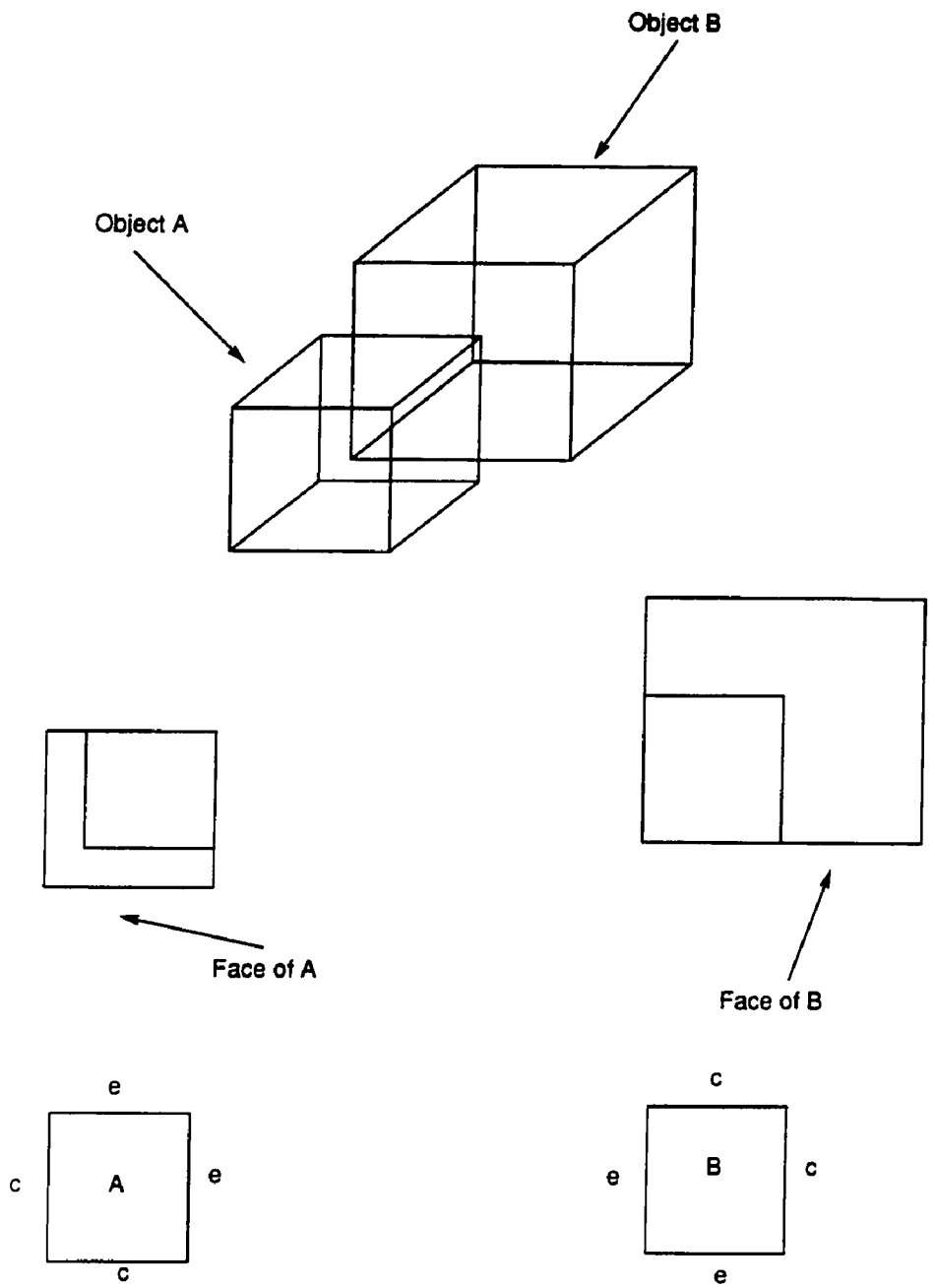


Figure 5.7: Handling Duplicate Coplanar Contours. Two cubes touch each other. This causes two overlapping contours, one in a face of A and one in a face of B. Information from these contours is used in conjunction to determine face-around-edge connectivity. *e* indicates a split-edge and *c* indicates a cut-line.

Table 5.1: Sub-Faces to be Included in Various Polyhedron Combinations

Elements to include	$P_\alpha \bigcup P_\beta$	$P_\alpha \bigcap P_\beta$	$P_\alpha - P_\beta$
Element of P_α inside P_β		+	
Element of P_α outside P_β	+		+
Element of P_β inside P_α		+	−
Element of P_β outside P_α	+		
Element of P_α on P_β, in same direction	+	+	
Element of P_β on P_α, in same direction			
Element of P_α on P_β, in opp. direction			+
Element of P_β on P_α, in opp. direction			

Table for other combinations are similar

5.5.4 Summary of the Impact of Special Cases

To summarize, special cases due coincident geometric entities have the following effects on the algorithm:

1. The arity of the tuple-sets increases. This implies that the space requirement increases.

2. Duplicates must be detected and handled appropriately.

3. The algorithm has a greater reliance on explicit point classification tests to guarantee correctness. The ability to to infer classifications implicitly and to propagate classifications diminishes.

4. The maintenance of *flags* to indicate special case situations, as shown in the previous discussion, ensures that the speed of the algorithm is not affected drastically for the non-special cases.

5.6 Analysis of the Algorithm

The complexity of geometric algorithms such as the Boolean combination algorithm is not easily analyzable. The worst-case analysis often gives pessimistic results, and the average-case analysis is hard to do, which is confirmed by the dearth of any such analysis in the literature. In fact there is little consensus on what an average case is. We define an average case as an input set which satisfies the assumptions outlined in the Section 5.3.2. An attempt is made in this section to provide an approximate expected-case analysis for the shared-memory version of the algorithm. Output-sensitive variables are defined to derive the complexity of the algorithm in terms of the size of the output.

In this section p refers to the number of processors used and G refers to the grid size. Note that p is small compared to the number of faces in the objects. So long as a word of memory is cheaper than a processor, p will be much less than the problem size in most cases. n_f is the total number of faces in both the objects. E refers to the edges of the polyhedra. E_a represents an edge of polyhedron P_a. n_e is the total number of edges in the two objects. Every edge is counted once for each face it belongs to. $\overline{n}_e$ is the average number of edges per face of the object.

5.6.1 Domain Transformation of Faces

In order to transform the coordinates of the edges from the WCS to the local coordinate system of the face, it takes $\theta(e)$ time where e is the number of edges in the face. In the average case, each processor handles n_f/p faces. Hence, this step takes $\theta(n_f\overline{n}_e/p)$ time, assuming that in the average-case the load is evenly distributed among the processors.

5.6.2 Putting Faces into the Grid

The determination of the bounding box for the face takes $\theta(e)$ time, where e is the number of edges in the face. The number of cells through which the face goes depends on the grid size G and the area of intersection of the plane of the face and the bounding box. If the average area of the intersection is A_b, then each face goes into $A_b \times G^2$ cells. If the volume of the bounding box of the face is V_b, then this step requires $V_b \times G^3$ *point against face* tests. It is usually a small constant since the bounding boxes for the faces are comparable to the size of the grid cell. Thus this step takes $O(\overline{n}_e)$ time per face on the average. Since each processor deals with n_f/p faces, this step takes $O(n_f\overline{n}_e/p) = O(n_e/p)$ time with our assumption about the average size of the faces. We assume that in the average-case the load is evenly distributed among the processors.

5.6.3 Computing Face-Face Pairs

The number of tentative face-face pairs generated depends on the interaction between the two objects and the grid size G. Let n_{cf1} and n_{cf2} denote the number of cell-face pairs for the first and second object respectively. It takes $\theta(n_{cf1} + n_{cf2})/p)$ time to sort these pairs by the cell number by using an efficient parallel radix sort as given in [48]. The computation of the face-face pairs takes $\theta((n_{cf1} + n_{cf2} + n_{ffb})/p)$ time in the average-case as the two cell-face lists have to be scanned and the face-face pairs have to be stored. If n_{ffb} is the number of such pairs before duplicates are removed, and n_{ff} the number of pairs after pruning, the pruning step takes $\theta(n_{ffb} + n_{ff})/p)$ time, ignoring collisions in accessing the hash table while making entries.

5.6.4 Computing the Face-Face Intersections

The preliminary test for checking whether two faces intersect involves the testing of the edges of each face with the plane of the other and takes $\theta(e)$ time, where e is the number of edges in the face. The conclusive check involves a 1-D intersection and merging step of the common ranges of intersection which can take $O(e \log e)$ time (to sort the coordinates of the common ranges of intersection) as there will be at most $O(e)$ such ranges. However, in most cases it takes only $O(1)$ time as there are only a small constant number of cut-lines between any two faces.

Assuming $\overline{n}_e$ edges for each face, the above step can take $O((n_{ff}/p)\overline{n}_e \log \overline{n}_e)$ time and takes $O(\overline{n}_e n_{ff}/p)$ time in most practically encountered cases.

5.6.5 Sub-Face Topology Reconstruction

Let e be the number of edges and n be the number of vertices in the planar graph under consideration. Note that $n < e$ for the planar graphs considered here as we do not encounter trees. Let n_c and n_{cc} be the number of contours and connected components, respectively, in the planar graph of the face to be partitioned into sub-faces. In the following analysis we assume that only two faces are adjacent to an edge. Let v_i refer to the ith vertex and let $deg(v_i)$ be its degree.

Lemma 5.1 *The maximum degree of a vertex in the above planar graphs is 4.*

Proof: The degree of every vertex in the original face of the object is 2. Any new vertices formed by the intersection of the other object with this face have a degree of 2, 3 or 4. New vertices formed by the cut-lines on this face which do not intersect any of its edges and which are not caused by edges lying on it have a degree of 2, because only 2 faces are adjacent to any edge in the objects we consider. The degree is 3 if a face of the other object intersects an edge of this face. The degree is 4 if an edge of the other object (and hence two faces) intersect an edge of this face. If an edge of a face lies on another face, the degree can be 2 or 4.

When non-manifold conditions are allowed in the input polyhedra, the maximum degree in the planar graphs depends on the number of faces that can be incident on a single edge.

The following is the time taken to create a planar graph, given the edges and the cut-lines for the face:

1. Sorting the vertices of the e edges in the graph takes $O(e \log e)$ time.

2. Identifying the vertices takes $\theta(e)$ time as it needs one pass down the sorted list of the vertices of the edges.

3. In order to form the adjacency list representation of the graph, it is necessary to identify the labels of the other vertex of the edges incident at each of its vertices. This process takes $\log n \sum_{i=0}^{n} deg(v_i) = O(e \log n)$ time, as it takes $\log n$ time to search for the label of a vertex, given its coordinates.

4. It takes $\sum_{i=0}^{n} deg(v_i) \log(deg(v_i))$ to manipulate the adjacency lists so that the vertices around a vertex are ordered in a clockwise manner. If it is assumed that the degree of each of the vertices is equal to the the average degree of a vertex, which is $2e/n$, this step takes $O(e \log(e/n))$ time. In an average graph, e is comparable to n, and this step takes $O(e)$ time. Even otherwise, the upper bound on the degree of the vertices of these graphs suggests the same order of complexity.

5. Finding the connected components of the graph takes $O(n + e)$ time as it is equivalent to a depth-first search.

The complexity of the traversal of the graph created above, to determine all its sub-faces, is given below.

1. During the traversal of the graph to identify its contours, each vertex in the graph is visited $deg(v_i)$ times. It takes $deg(v_i)$ time at each visit to a vertex

to search for the next vertex in the contour from the adjacency list for v_i. Hence this step takes $\sum_{i=0}^{n} deg(v_i) \times deg(v_i)$ time. Since the average degree of a vertex is $2e/n$, this step takes $O(e^2/n)$ time. In an average graph e is comparable to n and hence the above step takes only $O(e)$ time, which is optimal because all the edges have to be visited to identify all the contours.

2. The maximum nesting depth for the connected components for the graph is n_{cc}. The bounding boxes for all the connected components can be determined in $O(n)$ time, as each vertex knows the component it belongs to. Sorting the bounding boxes by area takes $O(n_{cc} \log n_{cc})$ time. A total of $n_{cc}^2/2$ bounding box checks have to be done in the worst case. However, n_{cc} such checks is a more reasonable estimate for practical cases. If the bounding boxes of two components overlap, then determining whether a component is contained inside a second component is equivalent to determining whether a point from the first component is in the second component. This takes $O(e_c)$ time, where e_c is the number of edges in the component, as all the edges of the component are visited. Thus, deriving the complete hierarchy information can take $O(n_{cc}^2 e_c)$ time in the worst case, where all the components are disjoint and *all the bounding box tests fail* and the point in component procedure is invoked for each failure of the bounding box test. A synthetic example where this case can occur is shown in Figure 5.8. *We have not observed such cases in our test examples.* It is more likely that there are n_{cc} point in polygon tests in the average-case. Thus, this step takes $O(n_{cc} e_c)$ time.

Hence, in the average-case, the complete sub-face topology reconstruction algorithm takes $O(e \log e + n + n_{cc}(e_c + \log n_{cc}))$ time. In most cases $n_{cc} = 1$ or 2 and this step takes $O(e \log e)$ time. Each processor has to handle n_{ffx}/p such graphs, where n_{ffx} is the number of faces that actually intersect.

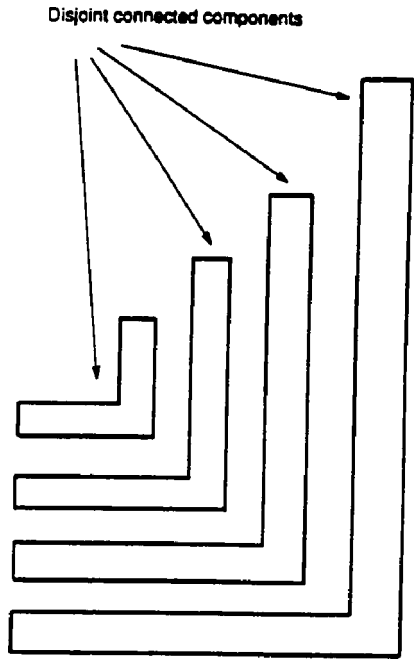


Figure 5.8: A Bad Case for the Sub-Face Algorithm. Though the connected components are disjoint, their bounding boxes overlap.

Therefore, this step takes $O(n_{ffx} \times e_{avg} \log e_{avg}/p)$ time, where e_{avg} is the average number of edges in a graph.

5.6.6 Determining Face Adjacency around Edges

If the resulting object has n_{oe} edges, $\theta(n_{oe})$ *(edge, sub-face)* tuples are created. In the average-case each processor handles $O(n_{oe}/p)$ of these tuples. Hence, this step takes $O(n_{oe}/p)$ time.

5.6.7 Parallel Sorting

The algorithm requires parallel sorting for sorting the *(cell, face)*, *(face, cut-line, other-face)*, *(face, edge, cut-point)*, and *(edge, sub-face)* tuples. If n_{cl}, n_{cp}, and n_{oe}, denote the total numbers of the above tuples respectively, this takes

$O((n_{cf1} + n_{cf2} + n_{cl} + n_{cp} + n_{oe}) \times (1/p + \alpha p))$ time where αp accounts for collisions in accessing the buckets. Our experiments indicate that in most cases α is quite small.

5.6.8 Total Complexity

Thus the complexity of the complete Boolean operation algorithm is $O((n_e + n_{cf1} + n_{cf2} + n_{ffb} + n_{cl} + n_{cp} + n_{oe} + \overline{n}_e n_{ff} + n_{ffx} e_{avg} \log e_{avg})/p)$. Note that some of the parameters depend on the size of the input, some on the size of the output, and some on the grid resolution G.

5.7 Implementation and Results

The algorithm was implemented on the Sequent B21000 parallel machine with 16 processors. Our implementation differs in the following aspects from the algorithm presented:

1. The grid size to use is not determined by the program.
2. Face adjacencies across the edges of the resulting object are not determined.
3. Overlapping coplanar faces are not handled.

Figures 5.9, 5.10 and 5.11 show some of the data sets used to analyze the performance of the algorithm. In addition to this, two sets of 216 uniform cubes each were used, as the Boolean operation on cubes is useful in domain decomposition and VLSI problems. Figures 5.12, 5.13 and 5.14 show some of the results of the algorithm.

Figures 5.15 to 5.19 show the time taken and the speedup achieved for each phase of the algorithm for the set of cubes. A $6 \times 6 \times 6$ uniform grid was used for this example. Each of the objects had 1296 faces and 5184 edges. Computing

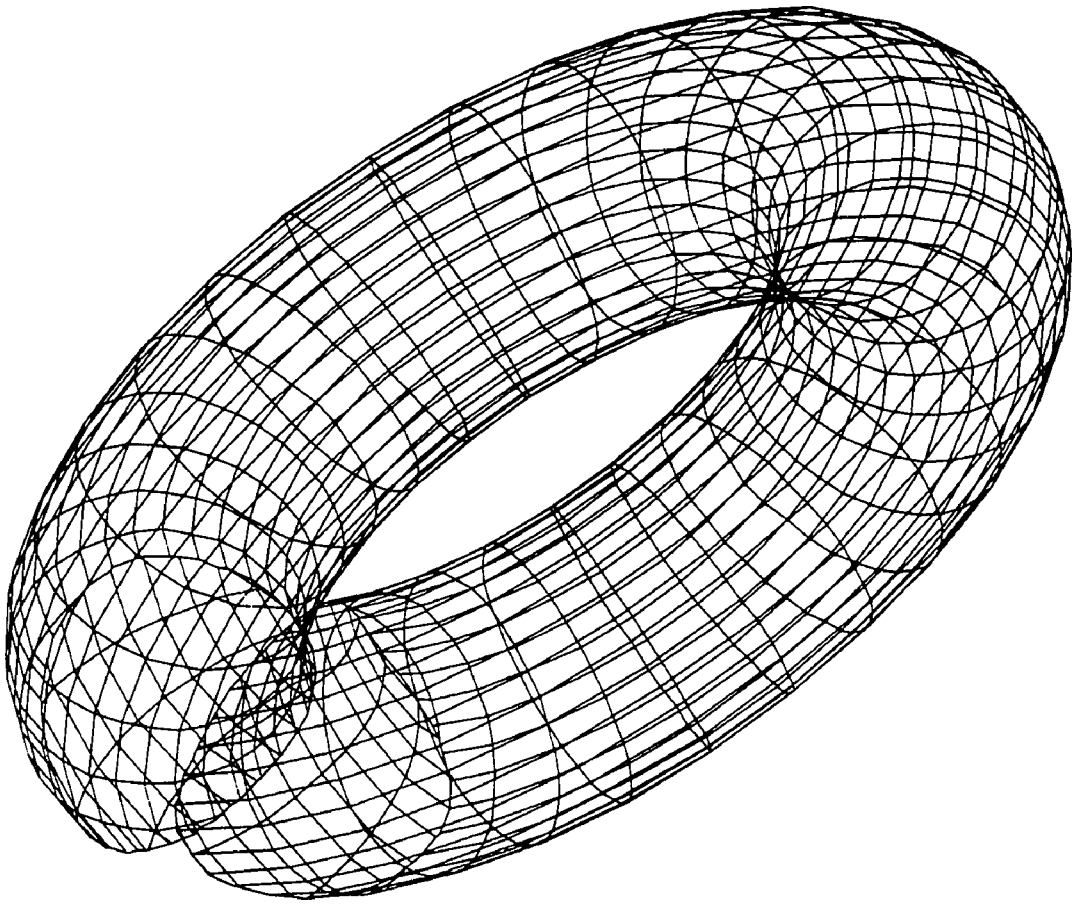


Figure 5.9: Data Set: Polyhedral Approximation of a Split Torus

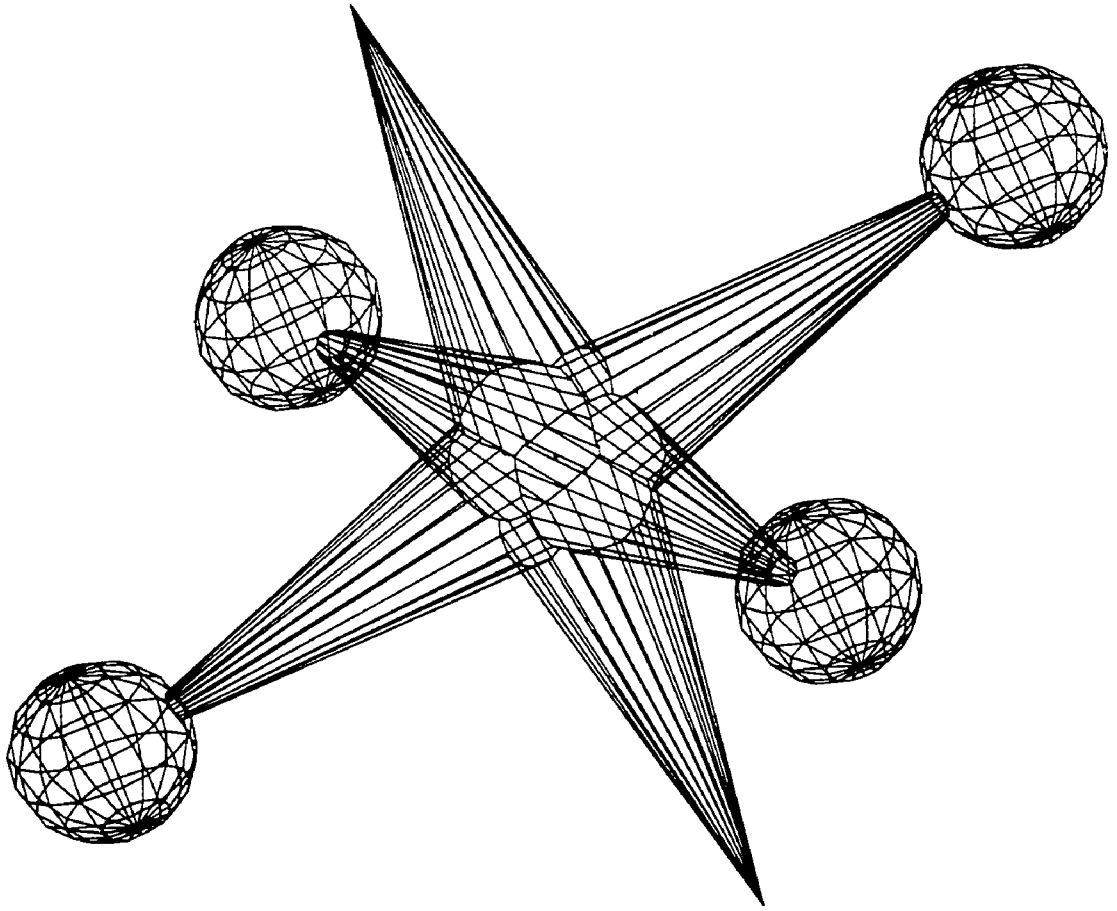


Figure 5.10: Data Set: Polyhedral Approximation of a Compass

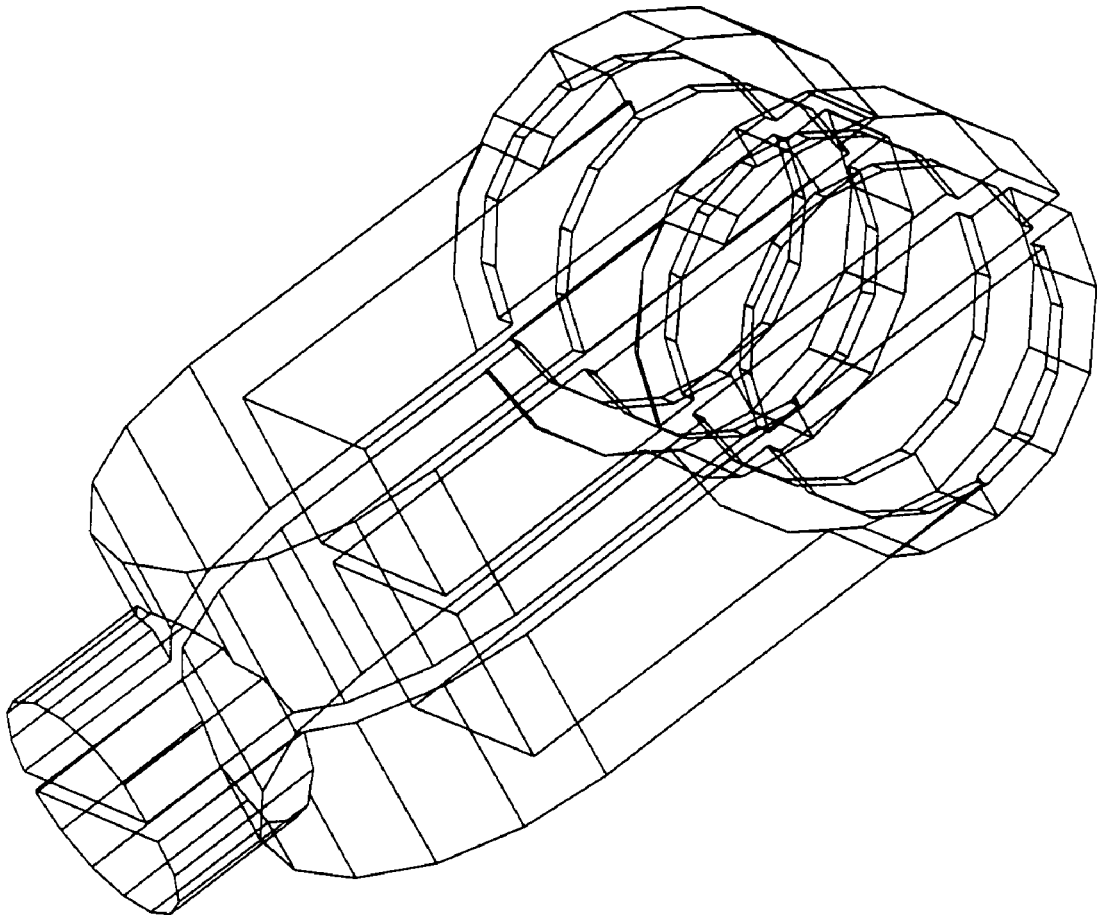


Figure 5.11: Data Set: Polyhedral Approximation of a Mechanical Part

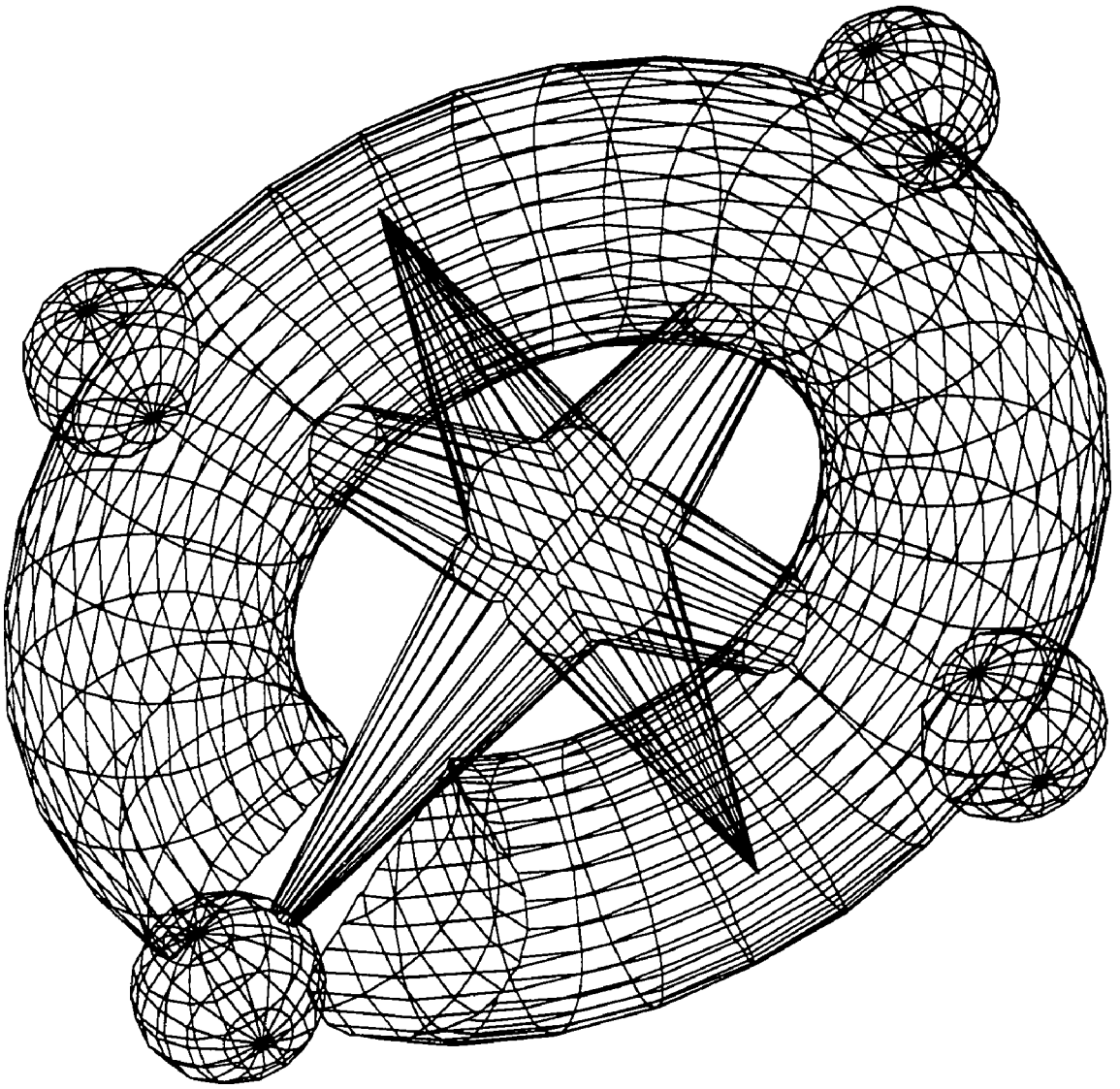


Figure 5.12: Results: Union of Compass and Split Torus

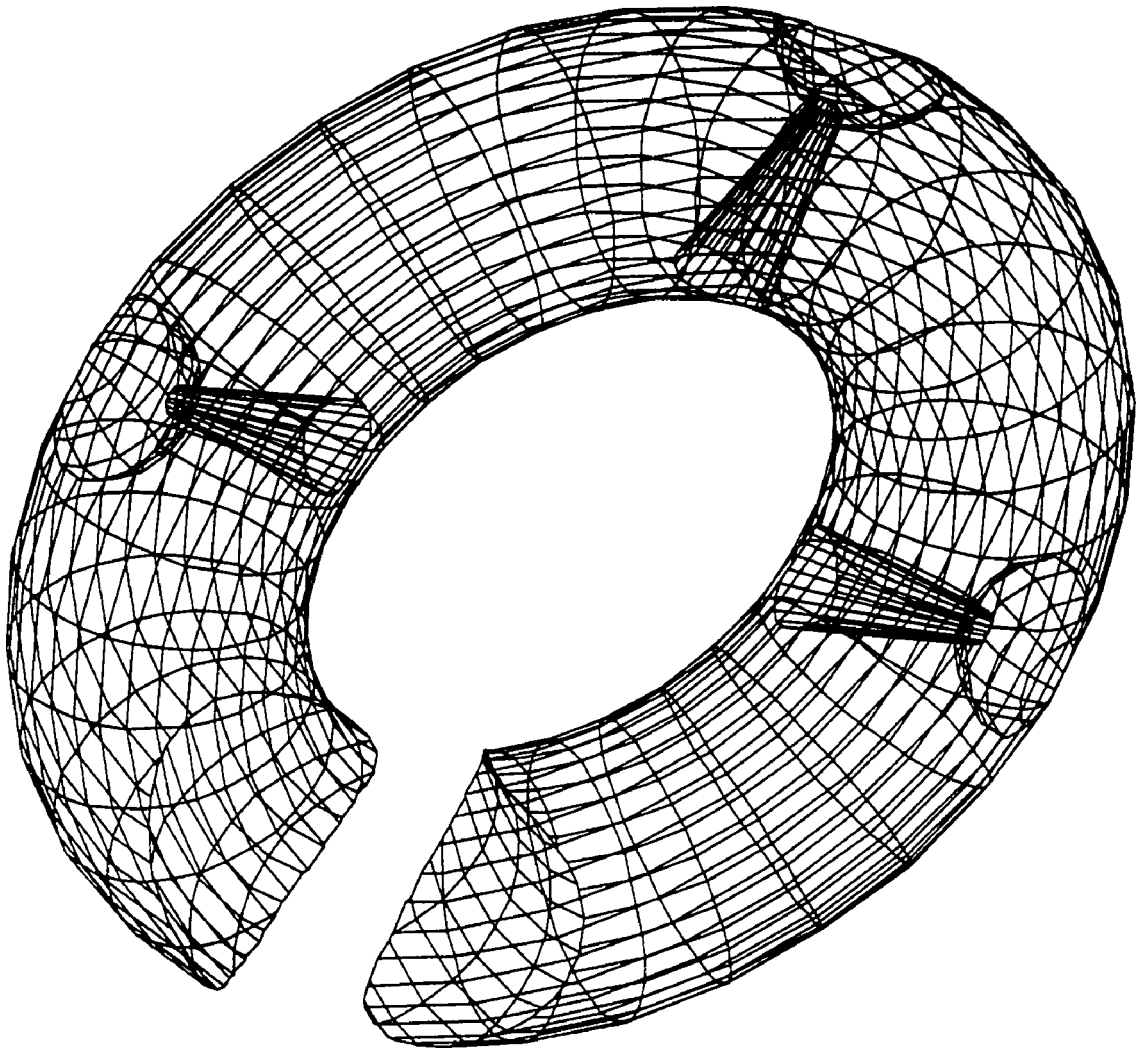


Figure 5.13: Results: Compass Subtracted from the Torus

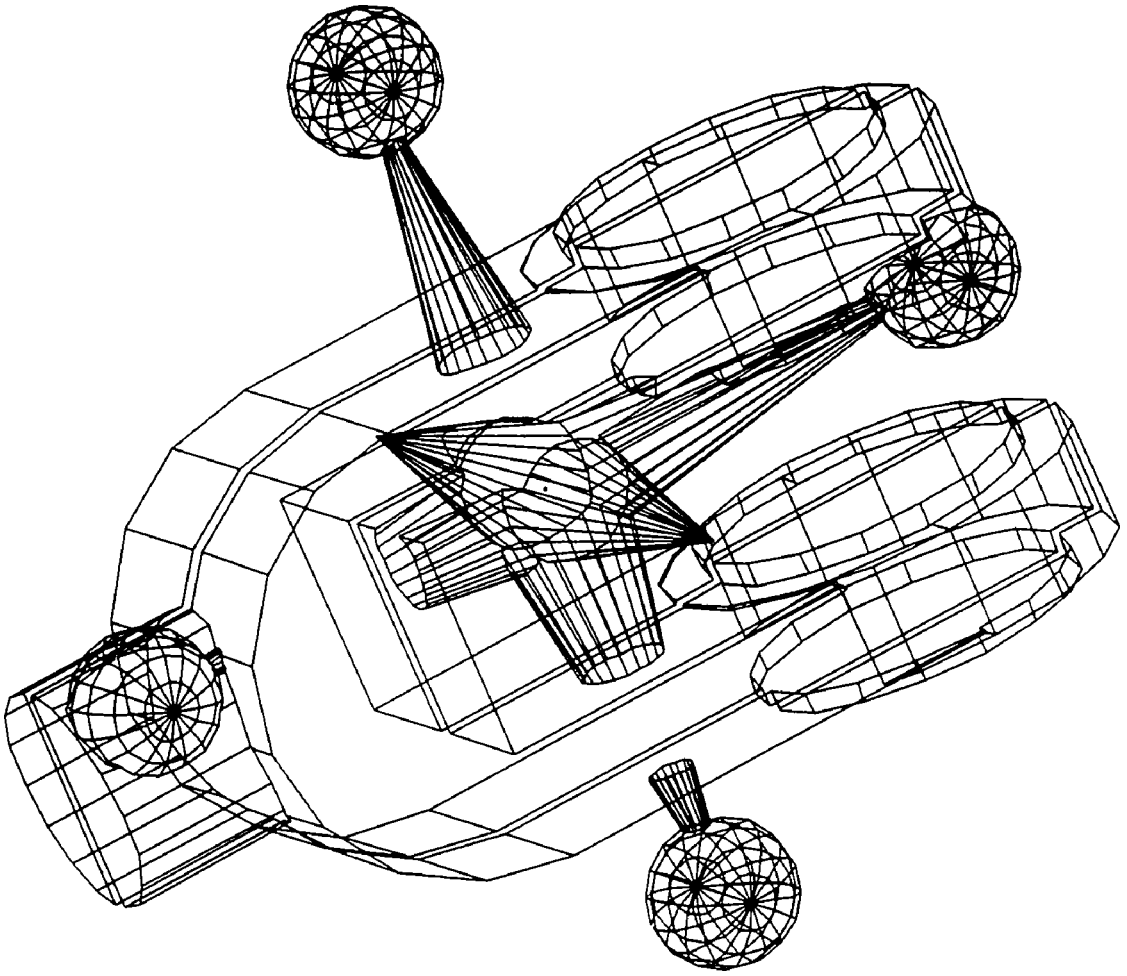


Figure 5.14: Results: Union of a Mechanical Part and Compass

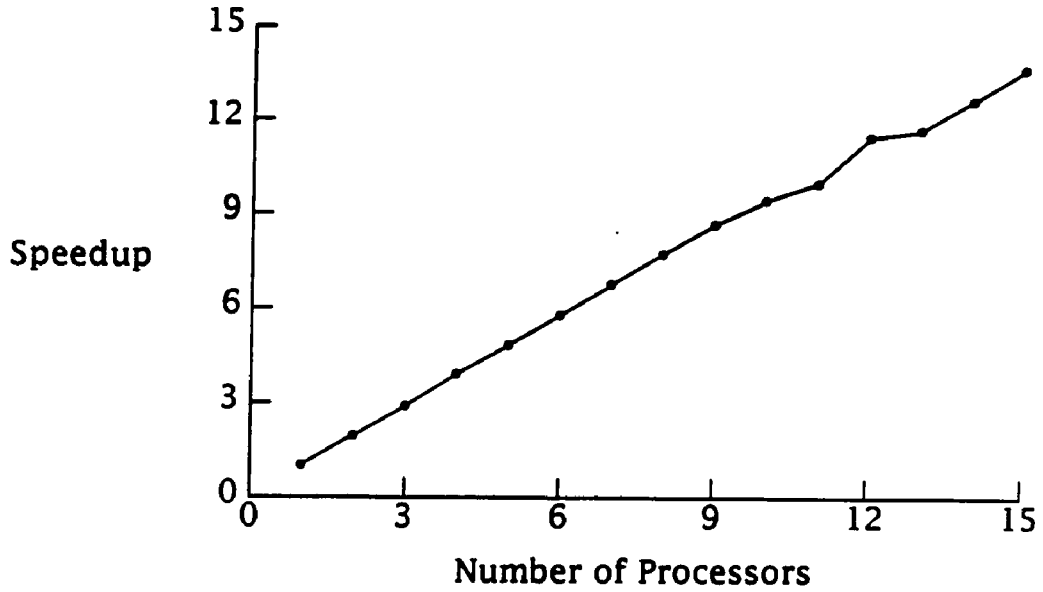


Figure 5.15: Cubes Example: Speedup for Domain Transformation of Faces

all the Boolean combinations for this example took only 17.55 seconds. A total of 3600 *(cell, face)* tuples were generated and 13824 *(face, face)* tuples were isolated for determining potential intersections, of which 1296 actually intersect. Thus the uniform grid reduced the number of face-face intersection comparisons to about 0.8% of the 1296 × 1296 comparisons for a brute-force algorithm. A study of Figures 5.15 to 5.19 and Table 5.2 shows that most phases of the algorithm show almost linear speedup. The overall parallel efficiency is about 77%, corresponding to a speedup of 11.54 while using 15 processors. The graphs show that speedups in excess of 13 while using 15 processors have been obtained.

Lest objections be raised against the nature of the objects used in the previous example, we tested our algorithm with two commonly encountered objects (see Figures 5.9 and 5.10) from computer-aided design. Figures 5.20 to 5.24 and Table 5.3 show a behavior that is similar to that for the previous example. Here

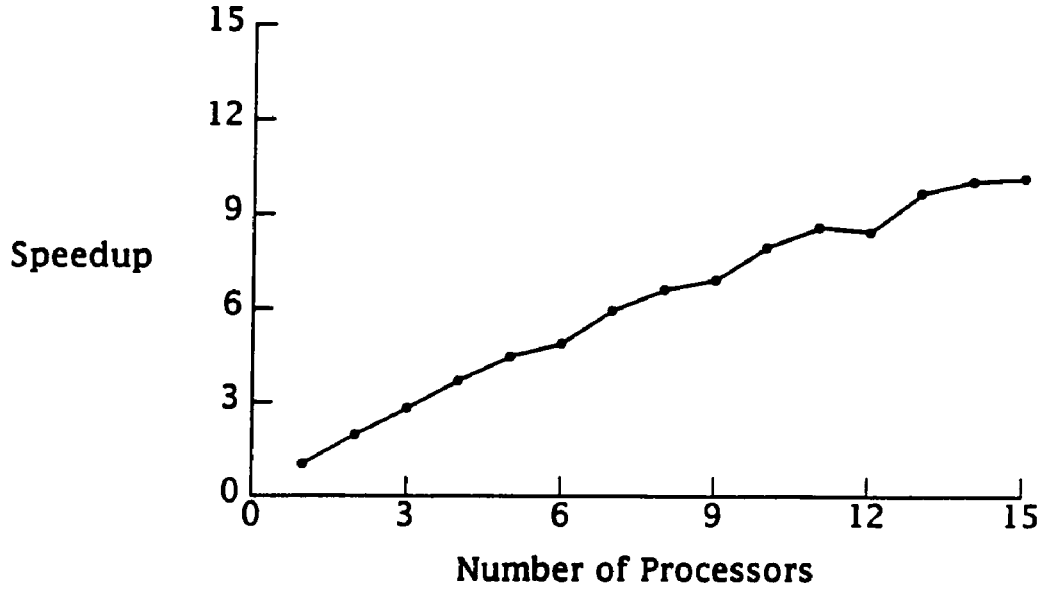


Figure 5.16: Cubes Example: Speedup for putting Faces into Grid Cells

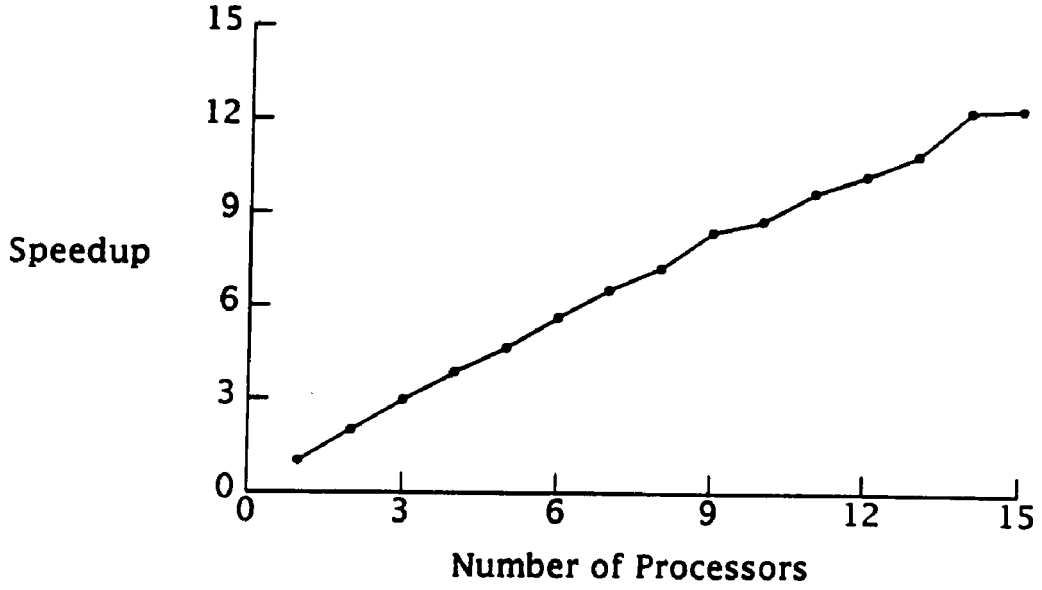


Figure 5.17: Cubes example: Speedup for intersecting Faces

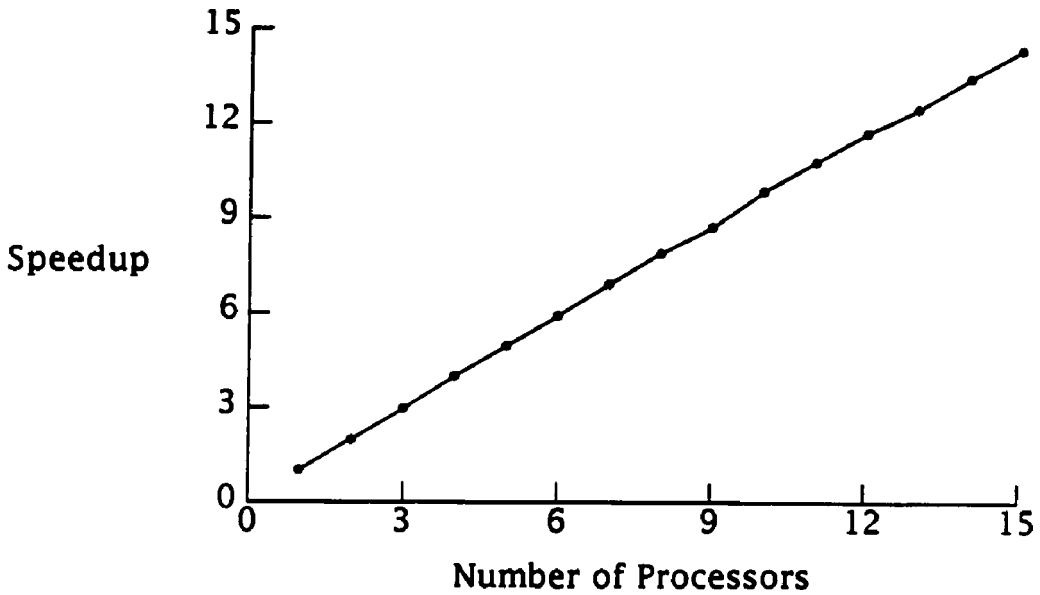


Figure 5.18: Cubes Example: Speedup for forming Sub-Faces

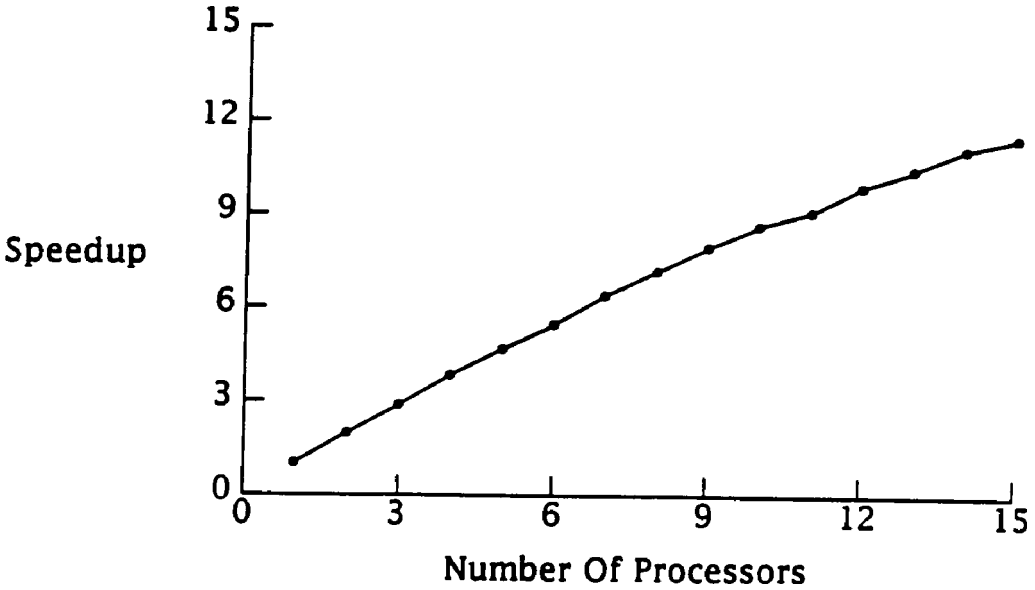


Figure 5.19: Cubes Example: Speedup for Complete Polyhedral Intersection

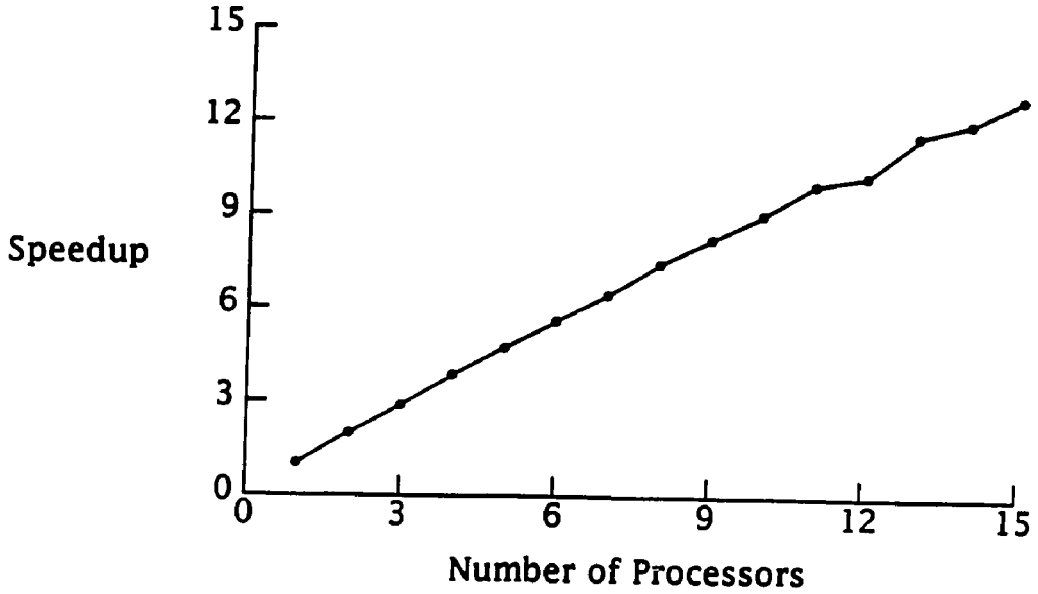


Figure 5.20: Compass and Torus: Speedup for Domain Transformation of Faces

one object has 1030 faces and the other object has 696 faces. Computing all the Boolean combinations for this example took only 8.88 seconds. A $12 \times 12 \times 12$ uniform grid was used in this case. A total of 7239 *(face, face)* tuples (1.01% of what a brute force algorithm would test) were tested for intersections, of which only 214 actually intersect. The sorting of cut-lines and cut-points in this example did not show good speedup because of their small number (less than 500). As in the previous example, the pruning of *(face, face)* tuples to eliminate duplicates shows a poor speedup. As a consequence, the overall speedup for this example was 9.18, though the other steps showed significantly higher speedups.

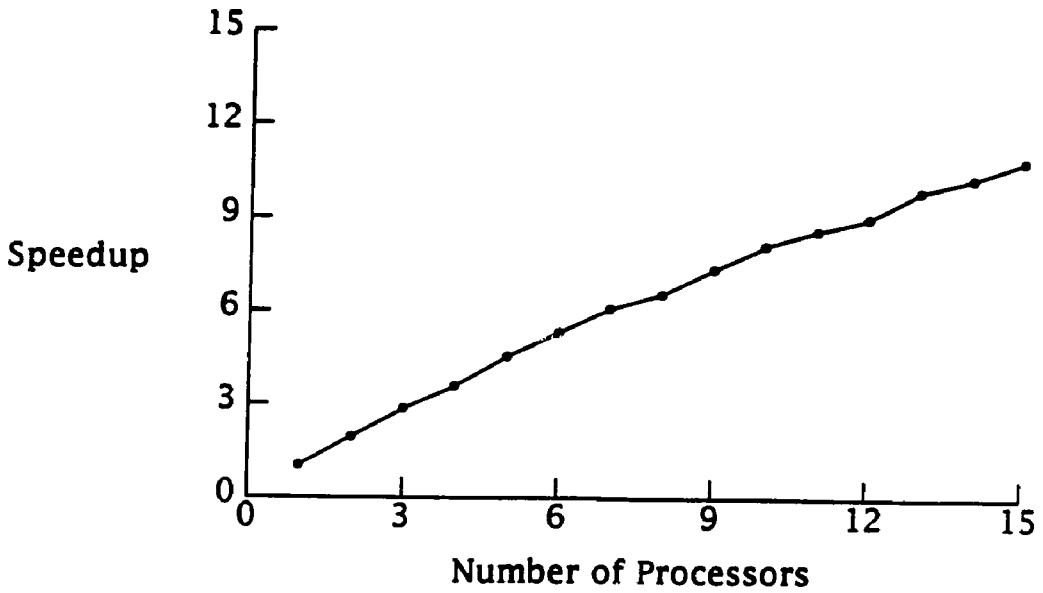


Figure 5.21: Compass and Torus: Speedup for putting Faces into Grid Cells

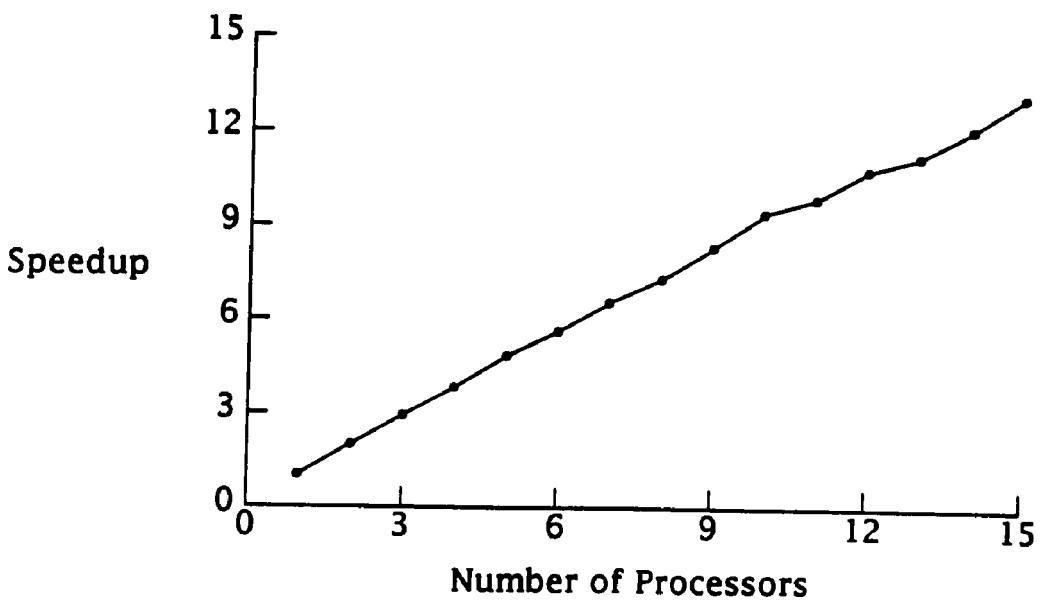


Figure 5.22: Compass and Torus: Speedup for Intersecting Faces

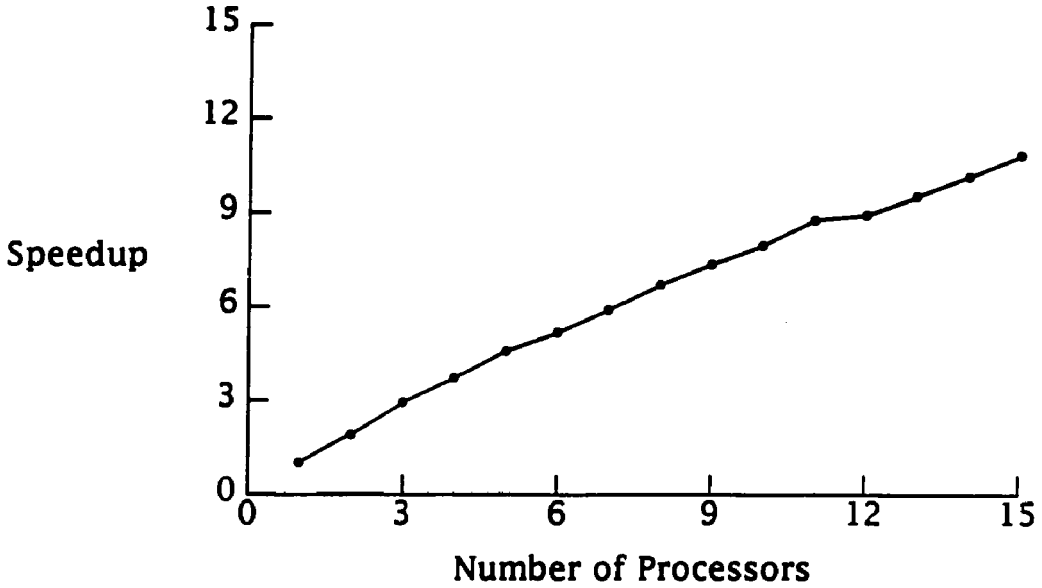


Figure 5.23: Compass and Torus: Speedup for forming Sub-Faces

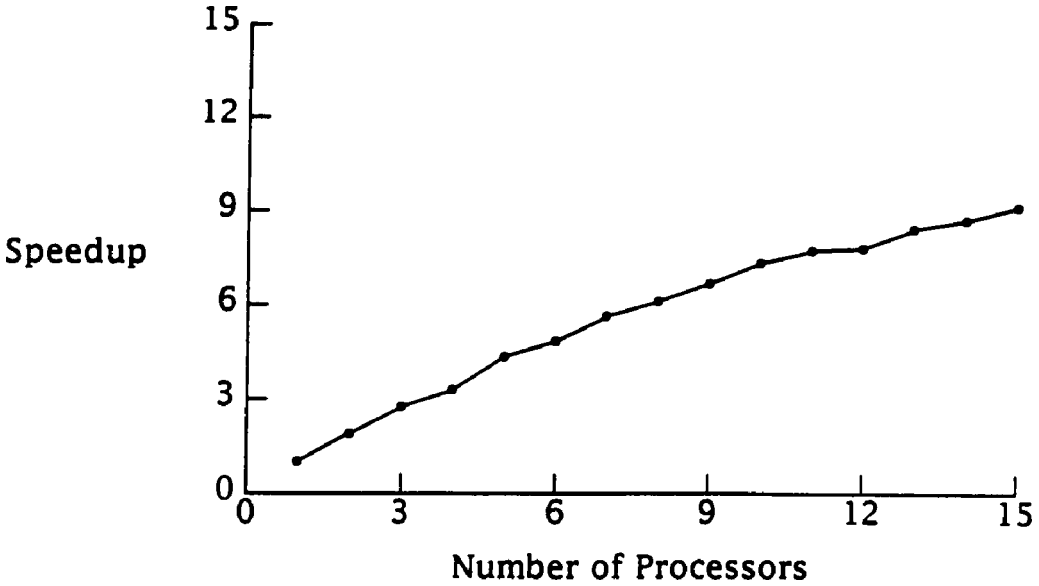


Figure 5.24: Compass and Torus: Speedup for Complete Polyhedral Intersection

Tables 5.2 and 5.3 show the speedup for the various phases of the algorithm and gives the time taken for each step as a fraction of the total time for the algorithm for the two examples discussed above. It shows that the domain transformation, face-face intersection, and the sub-face topology reconstruction phases account for more than 85% (75%) of the total sequential time and 72% (56%) of the total parallel time for the first (second) example. These phases show very good speedup, i.e., more than 12 (9.5) with 15 processors for the first (second) example. Figures 5.15 to 5.19 and Figures 5.20 to 5.24 show that the speedup for these phases is almost linear. This means that the time taken by these important and time-consuming phases will further reduce as the number of processors increase. Hence it is worthwhile to use a bigger parallel machine for this problem. Other phases such as the determination of *(cell, face)* tuples which account for about 10% of the total time, show speedups of 10 or more with 15 processors. The parallel sorting phases yield sub-linear speedup. The actual speedups range from 7 to 9 with 15 processors when there are more than 500 elements to sort. The step which prunes the *(face, face)* tuples shows an extremely poor speedup of about 3. This is probably due to frequent collisions in accessing the hash table. Fortunately, the phases which show poor speedup account for only a small fraction of the total time for the algorithm. The overall speedup is less than ideal due to a small fraction of inherent sequential computation in the algorithm. Slight imbalances in the distribution of work to the processors, due to the variation in the complexity of the individual faces of the objects, is another cause for deviation from the ideal speedup.

Counters and timers were introduced in the implementation to monitor the sizes of data structures that were not exactly predictable in advance. Examples include the numbers of *(face, face)* tuples and *(cell, face)* tuples. Table 5.4 shows the statistics gathered for the two examples. This information allowed us to gain

an intuitive understanding of the internal behavior of the algorithm and to verify our theoretical analysis given in Section 5.6. Tables 5.2, 5.3, and Table 5.4 together show the linear variation of time with the cardinality of the tuple-sets. This result and the linear speedups conform closely with our theoretical prediction. A more detailed comparison between experiment and approximate theory is given in Table 5.5.

We define an *agreement factor* which is an indication of the correlation between the theoretical analysis and the experimental observation.

$$Agreement factor = \frac{\text{cardinality of tuple-set example 1}}{\text{cardinality of tuple-set example 2}} \times \frac{\text{time for example 2}}{\text{time for example 1}}$$

An agreement factor close to unity indicates that the theoretical prediction and experimental results agree. Table 5.5 shows that when one processor is used, the agreement factor is very close to unity. Thus, the complexity of the polyhedron combination algorithm is indeed linear in the sizes of the tuple-sets on which it operates. When multiple processors are used, an agreement factor value close to unity, also indicates that the workload is evenly balanced. With 15 processors, the agreement factor for the sorting phases is not good when there are less than 500 elements to sort, because the parallel bucket sort shows poor speedups in such cases. Moreover, the agreement factor can be used to roughly estimate the time that would be taken for other data sets without executing the program, provided the appropriate parameters can be estimated.

The experiments show that there is very little variation in the speedup of the algorithm, as the number of grid cells is varied by a factor of 8 on either side of the optimal grid size. The total time taken by the algorithm varies only by 10 to 15% for the above variation in the grid size. The time taken for the determination of the *(cell, face)* tuples and the number of *(face, face)* tuples are two important quantities that depend on the number of grid cells. Most other

Table 5.2: Timing Statistics: Cubes Example. Shows the time taken by the important phases in the polyhedron combination algorithm. Times taken by one processor and by 15 processors are shown, both as actual values and percentages.

Operation step of algorithm in ( )	Time p=1	% age time	Time p=15	% age time	Speedup
Domain transformation of faces (1)	18.53	9.15	1.35	7.69	13.73
Putting faces in the grid (3)	10.00	4.94	0.98	5.58	10.20
Sorting the *(cell, face)* tuples (4)	3.36	1.66	0.34	1.94	9.88
Pruning the *(face, face)* tuples (5)	4.67	2.31	1.51	8.60	3.09
Forming *(face, face)* tuples (5)	0.71	0.35	0.72	4.10	seq
Intersecting the *(face, face)* tuples(6)	54.54	26.92	4.39	25.01	12.42
Sorting cut-lines (7)	4.86	2.40	0.65	3.70	7.48
Sorting cut-points (8)	3.58	1.77	0.52	2.96	6.89
Forming sub-faces for object 1 (9)	51.90	25.62	3.61	20.57	14.38
Forming sub-faces for object 2 (9)	49.38	24.37	3.41	19.43	14.48
Propagating face classif. for object 1 (10)	0.52	0.26	0.04	0.23	13.00
Propagating face classif. for object 2 (10)	0.46	0.23	0.03	0.17	15.33
Classif. uncut shells for object 1 (11)	0.04	0.02	0.00	0.00	N. A.
Classif. uncut shells for object 2 (11)	0.03	0.01	0.00	0.00	N. A.
Complete Boolean combination	202.58	100.1	17.55	99.98	11.54

phases of the algorithm, such as the sub-face formation step, have no dependence on the uniform grid. The relative insensitivity of the total time with the grid size can be explained by the complementary nature of the dependence of the phases that depend on the number of grid cells. On the one hand, the cost of determining the *(cell, face)* tuples increases with the number of grid cells, as a face is likely to pass through more cells. On the other hand, the number of *(face, face)* tuples tested for intersections reduces as the number of grid cells increase. Since both these phases show comparable speedup, the overall speedup does not vary significantly with the grid size.

Table 5.3: Timing Statistics: Compass and Torus. Shows the time taken by the important phases in the polyhedron combination algorithm. Times taken by one processor and by 15 processors are shown, both as actual values and percentages.

Operation step of algorithm in ( )	Time p=1	% age time	Time p=15	% age time	Speedup
Domain transformation of faces (1)	13.11	16.08	1.02	11.49	12.85
Putting faces in the grid (3)	9.59	11.76	0.88	9.91	10.90
Sorting the *(cell, face)* tuples (4)	4.48	5.49	0.63	7.11	7.11
Pruning the *(face, face)* tuples (5)	3.23	3.96	1.29	14.53	2.50
Forming *(face, face)* tuples (5)	0.51	0.63	0.50	5.63	seq
Intersecting the *(face, face)* tuples(6)	30.12	36.94	2.28	25.68	13.21
Sorting cut-lines (7)	0.88	1.08	0.22	2.48	4.00
Sorting cut-points (8)	0.62	0.76	0.19	2.14	3.26
Forming sub-faces for object 1 (9)	10.99	13.48	0.92	10.36	11.95
Forming sub-faces for object 2 (9)	7.37	9.04	0.77	8.67	9.57
Propagating face classif. for object 1 (10)	0.21	0.26	0.04	0.45	5.25
Propagating face classif. for object 2 (10)	0.34	0.42	0.03	0.34	11.33
Classif. uncut shells for object 1 (11)	0.13	0.16	0.10	1.13	1.3
Classif. uncut shells for object 2 (11)	0.03	0.04	0.01	0.11	3.0
Complete Boolean combination	81.53	100.1	8.88	100.01	9.18

Table 5.4: Size of Data Structures: Both Examples

Entities	Cubes and Cubes	Compass and Torus
Processors	15	15
Grid size	6	12
Faces in object 1	1296	696
Edges in object 1	5184	2920
Faces in object 2	1296	1030
Edges in object 2	5184	4240
Face-face pairs before pruning	13824	9210
Face-face pairs after pruning	13824	7239
Cut lines for object 1	1296	214
Cut lines for object 2	1296	214
Cut points for object 1	1296	268
Cut points for object 2	1296	160
Cell-face pairs for object 1	2304	2082
Cell-face pairs for object 2	1296	2654
Cell-face pairs	3600	4736
Face-face pairs which actually intersect	1296	214
New segments for object 1	10368	2196
New segments for object 2	10368	1380
New sub-faces for object 1	1296	268
New sub-faces for object 2	1296	159
New contours for object 2	1944	402
New contours for object 2	1944	238

Table 5.5: Polyhedron Combination Algorithm: Analysis vs Experiments. Shows the agreement between the approximate theoretical analysis and the experimental results. Agreement factors close to unity indicate good agreement. The tuple-set used for estimation correspond to the ones indicated in the analysis. They are also indicated in parenthesis beside the operation for some of them.

$$AgreementFactor = A.F. = \frac{\text{cardinality of tuple-set example 1}}{\text{cardinality of tuple-set example 2}} \times \frac{\text{time for example 2}}{\text{time for example 1}}$$

For example 1 G = 12, for example 2 G = 12

Operation	A. F. p=1	A. F. p=15
Domain transformation of faces (edges)	1.02	1.10
Putting faces in the grid (edges)	1.39	1.30
Sorting the *(cell, face)* tuples	1.01	1.41
Pruning the *(face, face)* tuples	1.16	1.44
Forming *(face, face)* tuples	0.90	0.90
Intersecting the *(face, face)* tuples	1.02	0.96
Sorting cut-lines	1.10	2.05
Sorting cut-points	1.05	2.21
Forming sub-faces for object 1	1.18	1.43
Forming sub-faces for object 2	0.83	1.26

Table 5.6: Timing results for Karasick's Algorithm. Timing data for sphere approximation. Each row describes the time and representation output size at each iteration. Note that these times are elapsed time as opposed to computation time.

Vertices	Edges	Faces	Time
16	24	10	5
20	36	18	10
52	84	34	21
88	144	58	55
130	226	98	71
246	406	162	128
412	676	266	242
644	1076	434	434
1070	1778	710	771
1840	2988	1150	1536
3016	4880	1866	3306
5204	8236	3034	6751

For completeness, Table 5.6, which shows the performance of Karasick's intersection algorithm, is included from [46]. Readers are however advised to be cautious in comparing raw computation times since the algorithms and machines on which they were implemented are not exactly identical.

5.8 Conclusion

A practical parallel algorithm for performing Boolean operations on two polyhedral objects was presented. The algorithm was implemented on an actual parallel machine and the results from implementation were discussed.

Like the earlier algorithms, this too combined the use of the uniform grid, a parallel sort, and the technique of data partitioning for parallelization.

The uniform grid technique generated sub-tasks which could be solved in parallel, reduced the number of comparisons for determining the intersections between the faces, and reduced the query time for classifying points with respect to

a polyhedron.

The parallel bucket sort was used extensively to transform from the domain of one tuple-set to another. The sub-linear speedup achieved by phases which depended on sorting limited the overall speedup of the algorithm.

Most phases of the algorithm relied on data-parallelism. Speedups in excess of 10 with 15 processors indicate that the simple data partitioning scheme which distributed the faces, edges, cells, and the sub-edges, etc., among the processors, did not cause any noticeable load-balancing problems for the examples investigated.

The experimental results for the 15-processor case agree within a factor of 1.5 with our approximate theoretical analysis. This shows that the expected complexity of our parallel algorithm is linear in the sizes of the tuple-sets generated in the course of the algorithm.

CHAPTER 6
PARALLEL SEGMENT INTERSECTION ON A HYPERCUBE COMPUTER

6.1 Introduction

A parallel algorithm, based on the framework presented in Chapter 2, for determining all the intersections between line segments lying in the plane is described. This problem occurs in many geometric applications such as interference detection, visible surface determination, and Boolean operations of polygons. The sequential algorithm [6] is given first and its parallelization is presented next. This complements both the optimal sequential algorithm of Chazelle and Edelsbrunner [20], which seems hard to parallelize, and the parallel algorithm of Goodrich [39], which seems difficult to implement. In contrast to the polyhedron combination problem, this problem has no topological aspects.

The algorithms discussed in Chapters 3-5 were implemented on a tightly coupled shared-memory parallel machine. In order to test our framework for a different class of parallel machines, this algorithm has been implemented on the Intel iPSC1 hypercube distributed-memory machine. The main purpose of our implementation is to gain an insight into the communication requirements and the relative complexities of the various phases of the algorithm.

6.2 The Sequential Algorithm

This algorithm is the same as the one used in the preliminary phases of the polygon combination algorithm (Section 4.3). It is recalled here for convenience.

1. For each edge, determine which cells it passes through and create the appropriate *(cell, edge)* tuples.

2. Sort the list of *(cell, edge)* tuples by the cell number so that all the edges passing through the same cell are in consecutive locations.

3. For each cell, compare all the edges in it, pair by pair, to test for intersections.

6.3 Parallelization of the Sequential Algorithm

The key idea in parallelizing the sequential algorithm is that, computation of the cells through which the edges pass and of the intersections in the cells, can each be done concurrently.

The task of determining the *(cell, edge)* tuples is parallelized by letting the host-processor broadcast the edges to the node-processors which then compute, in parallel, the cells through which its edges pass. Note that in a distributed-memory machine the complete description of the edge is duplicated before broadcasting, unlike in a shared-memory machine where indices into the shared array of edges were sufficient. At the end of the above step, the *(cell, edge)* tuples of the edges handled by a processor are stored in its local memory.

Before the computation of intersections within the cells can proceed, all the *(cell, edge)* tuples that belong to a cell have to be gathered in the processor responsible for it. Potentially, every processor has to communicate with every other processor and this requires an *all-to-all personalized* (global) communication scheme. A simple (though sub-optimal) way to perform such an information exchange is to reduce the operation to an *all-to-all* (not personalized) communication operation. This type of communication can be done by embedding a ring in the hypercube network (using a Gray code labeling for the processors) and by circulating the *(cell, edge)* tuples around it. In the first iteration every processor sends the *(cell, edge)* tuples it computed, to the next processor in the ring. In the next $n-2$ iterations, the processors send the message they received in the last iteration, to the next processor in the ring. Communication and computation are overlapped.

While the message is on it way to the next processor, every processor processes the buffer it just sent out, to check whether any relevant *(cell, edge)* tuples are present in the buffer. If so, it stores them in its list of tuples. In order to speed up this retrieval step, the *(cell, edge)* tuples computed by each processor are first sorted by the cell number before the communication process begins. This ensures that the tuples for each processor are stored contiguously in the message buffers. After $n-1$ such iterations, every processor would have communicated with all the other processors and will therefore have all the *(cell, edge)* tuples it needs for computing the intersections.

Next, the cells are evenly distributed among the node-processors, and the computation of intersections within the cells is done concurrently. For reasons of implementational simplicity in the communication mechanism, every node-processor gets a set of cells with consecutive labels. This implies that a processor gets cells that occupy contiguous regions in the scene. Finally, each processor sends its results to the host-processor.

Note that the task of avoiding duplicate intersections between the same pair of edges involves no inter-processor communication. This is because the unique point of intersection belongs to the cell of only one processor. A similar version of this argument did not hold true for avoiding duplicate intersections between the same pair of faces in the polyhedron algorithm. In that case, explicit inter-processor communication was necessary to avoid duplicate intersections.

6.4 Implementation and Results

The above algorithm was implemented and tested on three different data sets. The first data set was randomly generated. The second data set was a set of edges representing the state boundaries of the U.S.A. To generated the third data set, the map of the U.S.A. was shifted by 10% and was overlaid on it. Figure 6.1

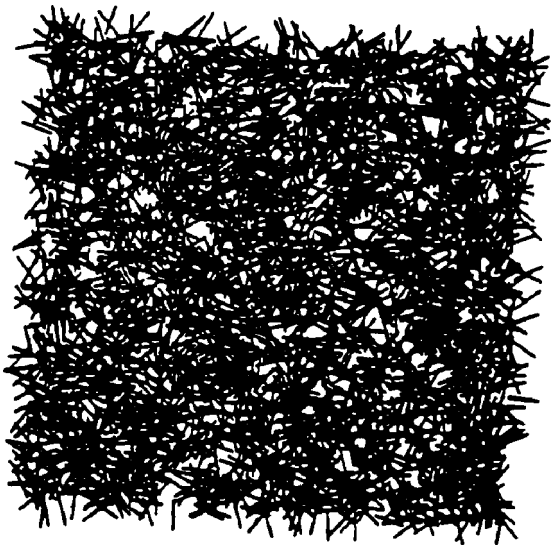
shows a plot of two of the data sets.

Figures 6.2 and 6.3 show the times taken by the main phases of the algorithm for various grid resolutions for the data sets shown in 6.1. The dotted lines represent the timings for the case where 32 processors were used. The solid lines represent the timings when 16 processors were used. The x-axis is stretched by a factor of two for the 16 processor case, so that comparisons between the 16-processor and the 32-processor cases can be made more easily. Thus a point (x, y) on a dotted line in the graph indicates that the processor whose *id* was x took y seconds for that stage of the algorithm. A similar interpretation holds for the solid lines. A closer study of the graphs in Figures 6.2 and 6.3 reveals the following:

1. The time taken to determine the *(cell, edge)* tuples varies inversely with the number of processors used. The time spent in this step with 32 processors is about half the time spent when 16 processors are used. This behavior holds true for both random and real edges.

2. The work of determining the *(cell, edge)* tuples is distributed evenly among the processors. This is shown by the fairly horizontal shape of the graphs for the grid time. This is the case even though the lengths of the edges vary considerably. The reason for this is that though the individual edge lengths vary considerably, the sum of the lengths of the edges processed by the processors is quite uniform. If this is not the case, more sophisticated schemes for distribution of edges among the processors have to be used for this step. However, it does not appear to be necessary while processing real scenes.

3. The intersection time decreases as the number of cells increases. This is true for both random edges and real edges. In the case of real edges, a few contiguous cells have a large number of edges at all grid resolutions. The

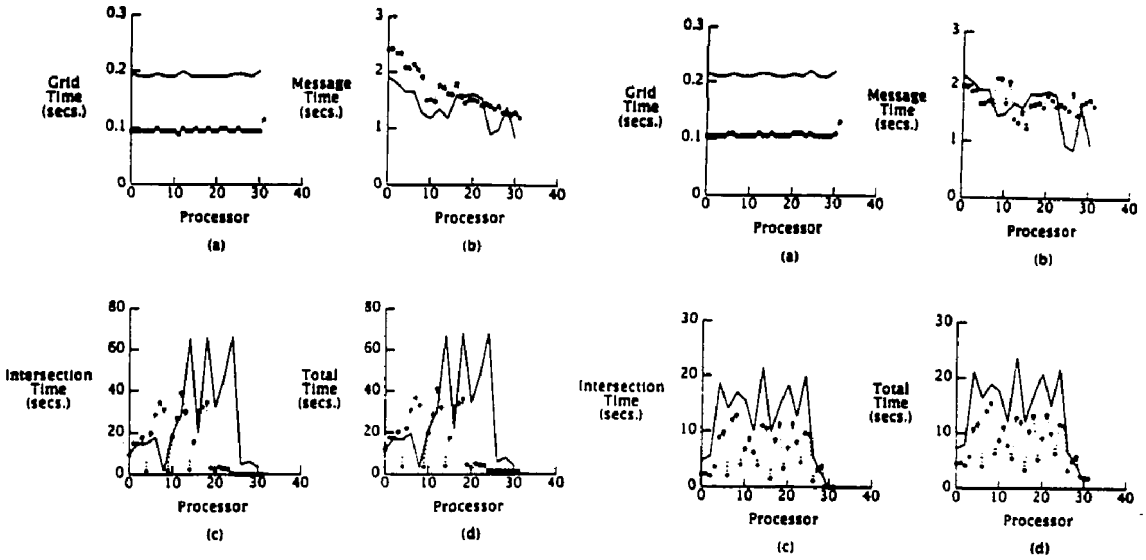


(a) Random edges

(b) Overlay of the maps of the U.S.A.

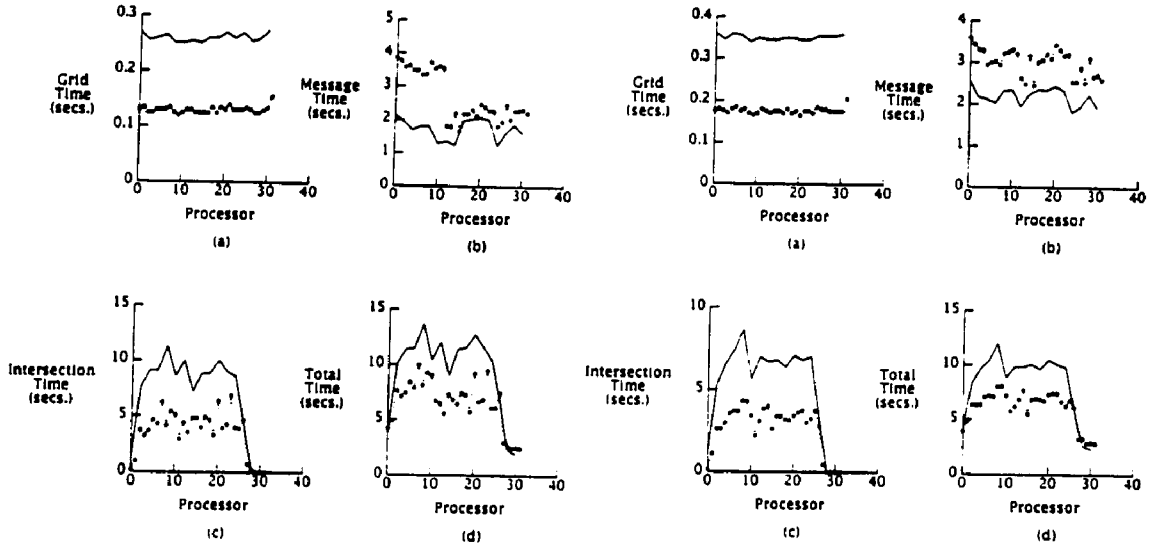
Figure 6.1: Data Sets Used for Testing the Algorithm

Number of edges: 3800, Number of intersections: 16823
Dotted Lines: 32 Processors, Solid Lines: 16 Processors



Grid Size: 5

Grid Size: 10

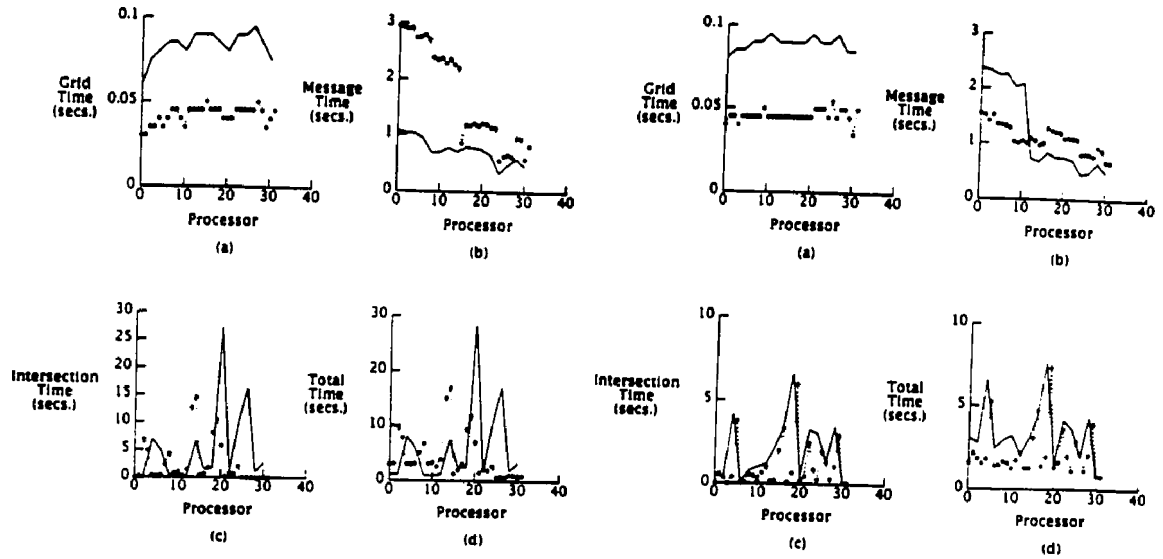


Grid Size: 20

Grid Size: 40

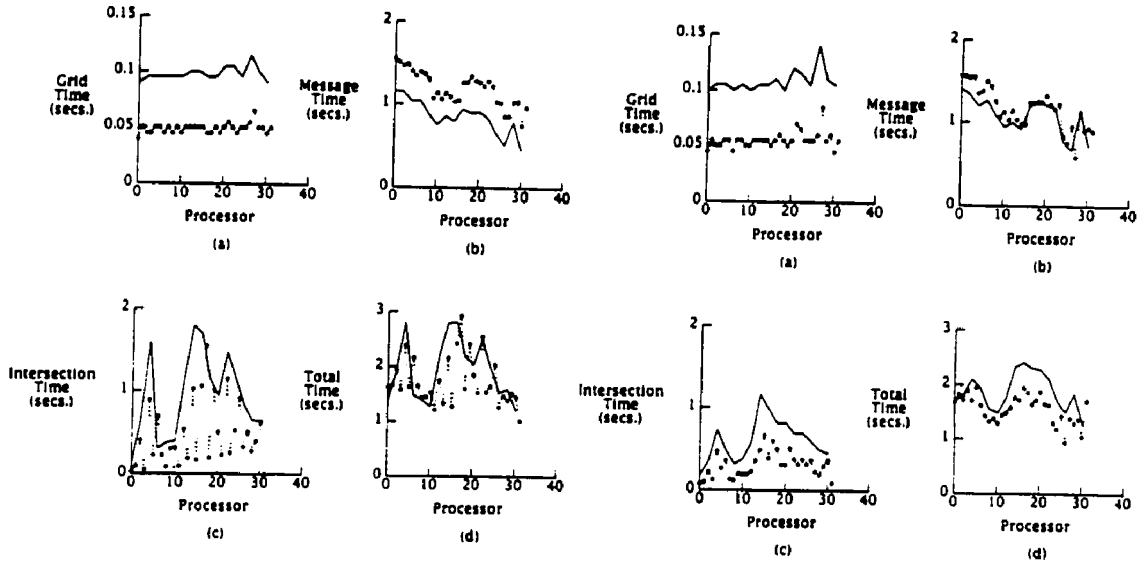
Figure 6.2: Timings: Example with Random Edges

Number of edges: 1830, Number of intersections: 2344
Dotted Lines: 32 Processors, Solid Lines: 16 Processors



Grid Size: 5

Grid Size: 10



Grid Size: 20

Grid Size: 40

Figure 6.3: Timings: USA Map Shifted and Overlaid on Itself

naive work subdivision scheme allocates contiguous dense cells to the same processor. Thus, the time taken by the processor responsible for this dense region dominates the computation. For random edges, the naive allocation of cells to the processors is not a problem.

4. The intersection time scales inversely with the number of processors when the data is random. The time taken by 16 processors is approximately twice the time taken by 32 processors. For certain grid sizes, when real data is used, the intersection time does not change much when 32 processors are used instead of 16. This behavior is caused by the naive allocation of cells to the processors. A solution to this problem is suggested in the following section.

5. The communication (message) time becomes significant (compared with the total time) when the number of cells increases and also when the number of edges is small. In the former case, it is due to the rapid rise in the number of *(cell, edge)* tuples, whereas in the latter case, the actual intersection computation takes very little time ($\approx 5\%$) compared to the communication time. In the first case a coarser grid resolution will give better results. In the second case it is not worthwhile to use many processors to solve the problem.

   The main observation is that while in the sequential algorithm the time taken is insensitive to the actual grid size over a wide range, in the parallel algorithm the number of cells has to be reasonably small so that the computation to communication ratio is high.

6. The communication time goes up as the number of processors increases. This is because of the larger size of the network. However, even with static, non-randomized, allocation of work to the processors, and a simple communication scheme, in all but one of the cases, communication time is less than

half the total time.

7. As the grid became finer, the slowest processor took less than twice the average processor time. This shows that the load is reasonably well distributed among the processors.

6.5 Suggestions for Improving Performance

The main purpose of our study was to determine the relative complexities of the various phases of the algorithm. In order to simplify the implementation of the algorithm, certain compromises were made which degraded the performance of the algorithm. The following are some simple ways to improve the performance of the algorithm:

1. The distribution of cells to the processors has to ensure that a processor is not assigned contiguous cells from a very dense region.

2. The global communication scheme used in the implementation is not optimal. In that scheme, all *(cell, edge)* tuples travel a distance of $(n-1)$ nodes, even though the diameter of the hypercube network is only $\log n$. More sophisticated communication schemes will alleviate this bottleneck.

3. *A priori* knowledge about the spatial extent of the edges can be used to distribute the edges to the processors which determine the *(cell, edge)* tuples. This scheme assumes a static assignment of cells to the processors. With this scheme, the need for global communication for reconfiguring the *(cell, edge)* tuples can be minimized. However, long edges which pass through many cells could require still global distribution.

4. Experiments indicate that the determination of the *(cell, edge)* tuples takes only a small fraction of the total time. Hence, it might be more efficient

to assign the task of determination and distribution of the *(cell, edge)* tuples, to the host processor. This obviates the need for all-to-all personalized communication and a host to node broadcasting scheme is sufficient.

6.6 Conclusion

This chapter demonstrated the mapping of the parallel segment intersection algorithm, based on the framework presented in Chapter 2, onto a distributed-memory machine.

The implementation revealed that the intersection and communication phases were relatively more important than the grid phase of the algorithm. The *speedup* of the algorithm was not measured because the amount of memory available on a single node-processor was not sufficient to test the algorithm on the complete database. Instead, the profile of the time taken by the processors and the load distribution among the processors were studied. Our results show that the slowest processor takes less than twice the average time taken by the algorithm. While this is a reasonable result, there is scope for substantial improvements, as was discussed in a separate section.

The communication cost for sorting of the *(cell, edge)* tuples was studied. The results from implementation confirm that these costs could become significant at finer grid resolutions. This factor could dictate the optimal grid resolution to use. The high startup cost of inter-processor communication makes the algorithm attractive only when the size of the edge database is sufficiently large.

CHAPTER 7
CONCLUSIONS AND RECOMMENDATIONS

7.1 Conclusions

This research has addressed the issue of using parallel processing to obtain faster performance for a set of geometric applications. A framework which combined the use of data partitioning, the uniform grid technique, and parallel sorting was presented. It was used to develop new parallel algorithms for the problems of determining the following:

1. the convex hull of a set of points in the plane,
2. the intersections between a set of segments in the plane,
3. the Boolean combinations of polygons, and
4. the Boolean combinations of polyhedra.

This thesis provided the first parallel algorithms for the last two of the above four problems. These problems differed in the dimensionality of the geometric entities dealt with, intrinsic complexity, and the balance between topological and geometric aspects. This demonstrated the generality of the framework. The implementation and results showed its practicality.

The simplicity and effectiveness of data partitioning for geometric algorithms was demonstrated. The tuple-sets necessary to solve the problems were identified. The attributes and arity of the tuple-sets depended on the problem. It was shown that special cases and problems due to numerical errors could increase the arity of a tuple-set. Problems with robustness could also increase the cardinality of tuple-sets.

The practical advantages of the uniform grid technique in the context of parallel processing were identified. Its use in the generation of sub-tasks, which could be solved in parallel, reducing the number of comparisons between geometric entities and the cardinalities of tuple-sets, and for efficient point-with-respect-to-polygon (polyhedron) tests was presented. Its potential to linearize the cardinalities of the intermediate tuple-sets was also shown.

This thesis provided algorithms that were sufficiently simple to be implemented on real parallel machines. This is in contrast to existing parallel geometric algorithms which are more complicated and appear difficult to implement. Table 7.1 summarizes the results from implementation. Speedups of 9 or more have been obtained with 15 processors. Close to linear speedups were achieved for most algorithms. The notable exceptions to this were the sorting phases.

The complexity, the relative importance in terms of fraction of total time, and the speedup of each of the phases of our algorithms were studied. Our experiments show that phases that depend on comparison-based parallel sorting could become bottlenecks as the number of processors increases. By identifying the bottlenecks in the algorithm, we have determined the aspects of the algorithm whose performance needs to be improved by the use of standard performance optimization techniques. We also expect these results to be of value in estimating performance when the number of processors increases, and on other parallel machines. When processors with different cost and capabilities have to be used, such studies can be used to better optimize the use of the processors.

It was shown that, with our framework, the geometric aspects were more easily parallelizable than topological aspects. Topological aspects could be parallelized efficiently only when data-parallelism was present. In the convex hull algorithm, the topological aspect depended on parallel sorting and showed a significantly poorer performance than the preprocessing phase. In the *restricted* version of

the polyhedron combination algorithm, it was demonstrated that the computation could be restructured so that the topological aspect could exploit data-parallelism. When special cases and numerical problems exist, this approach can still be used as long as the special cases are *detected in parallel* by independent processors but are *handled sequentially* by an algorithm which uses topological information to make consistent geometric decisions.

Though the issue of robustness of algorithms in the presence of numerical errors was not central to this research, the convex hull algorithm demonstrated that emphasis on robustness can reduce the speedup.

The experience with Intel iPSC1 machine reveals that, for complex problems considerable communication costs could be incurred. In the context of these machines, it was shown that the optimal grid resolution could be dictated primarily by message-passing costs. Also, allocation of contiguous cells to the same processor leads to uneven distribution of workload.

7.2 Summary of Experimental Results

Since geometric applications need to process large data sets efficiently, our algorithms were tested on large data sets. For each problem, an attempt was made to test the algorithm with data sets with different geometric characteristics, to study the efficiency of our techniques under different circumstances. Table 7.1 summarizes the results from implementation on the Sequent Balance 21000.

Our convex hull algorithm was experimentally shown to be faster than the parallel quickhull algorithm [23], for most commonly encountered point distributions. We were not able to compare our results with those of optimal algorithms because no performance figures were available.

We are not aware of any prior algorithms or results for parallel polygon combination and parallel polyhedron combination in object-space and hence we

Table 7.1: Algorithm Performance Summary. All algorithms were implemented on the Sequent Balance 21000. Timings shown are with 15 processors.

Convex Hull Determination				
Point Distribution	Points	Grid Size	Time (secs)	Speedup
In a unit circle	200,000	120 × 120	8.7	7.23
In a unit square	200,000	105 × 105	12.2	6.17
In an annulus	200,000	105 × 105	13.88	5.82
Polygon Combination				
Polygons	Edges	Grid Size	Time (secs)	Speedup
Randomly generated	12,000 and 6,800	97 × 97	177.19	13.50
Uniform squares	7,200 each	100 × 100	18.03	10.31
Polyhedron Combination				
Polyhedra	Faces	Grid Size	Time (secs)	Speedup
Uniform cubes	1296 each	6 × 6 × 6	17.55	11.54
Torus and Compass	1,030 and 696	12 × 12 × 12	8.88	9.18

have no common basis to compare our results for these problems.

A preliminary study of the use of our framework for parallelization on a distributed-memory machine was conducted. The results indicate that if there was no communication overhead, the algorithm would have executed three times faster, which is a respectable performance. We recall that the work subdivision scheme and the communication mechanisms used were not optimal and hence there is room for further improvement.

The timing results mentioned in this thesis are with parallel hardware that is 4-5 years old, and whose combined performance is comparable to that of current sequential workstations. We expect that, with parallel machines built with the latest technology, our algorithms will achieve close to interactive performance on similar problems.

7.3 Recommendations for Future Work

The dominance counting problem is closely related to the preprocessing phase of our parallel convex hull algorithm. Whether our algorithm can be used for this problem has to be determined. The convex hull problem in higher dimensions occurs in many areas. Our preprocessing technique will have richer dividends in higher dimensions. Hence, its application in this context has to be considered.

The present work concentrated on Boolean operations on two objects. It would be of practical value to extend this work for boundary evaluation of CSG trees with polygons or polyhedra as primitives. Combining ideas with other strategies such as active zones [68] and restructuring of unbalanced CSG trees [40, 54] could lead to further improvements. This needs to be explored carefully. We have begun this work in [56] for the determination of the mass properties of polygonal CSG objects, without evaluating its boundary.

In order to demonstrate the basic ideas, our algorithms were restricted to

straight lines and planar polyhedra. In a practical situation, curved surfaces are more useful. Can the ideas presented in this thesis significantly improve the overall performance of the geometric aspects in this domain? The parallelization of the topological aspects is an even more challenging task. A solution to these problems would be of great commercial value.

An in-depth treatment of the problems due to special cases and problems due to numerical errors was beyond the scope of this research. However, the synergy of robust computations with parallel processing is essential because robustness and speed are both equally important issues in a practical situation. Therefore, the integration of these ideas warrants investigation.

When the number of processors in the parallel machine increases while the size of the data set remains the same, the algorithms have to become more fine-grained in order to exploit all the processing power. Coarse-grained data-parallelism used in our work might not be sufficient. Some of the currently sequential steps, for e.g., the sub-face topology reconstruction step, have to be parallelized. Further research is called for in this area so that more massively parallel processors may be used to achieve further improvements in performance.

Distributed-memory machines are likely to be more prevalent in the future. Our investigation in this area was preliminary. Additional work is required to study the communication and load-balancing needs for geometric problems. In these machines, locality of memory references may be less crucial. Hence, the efficiency of spatial partitioning schemes other than the uniform grid has to be determined.

The problem of developing special-purpose parallel machines for graphics applications is now fairly well understood. It has to be determined whether special-purpose parallel machines for a class of geometric problems, for e.g., intersection

detection, are practical. To do this, the common operations, memory access patterns, computation to communication ratio, etc., for this class of geometric algorithms, have to be determined. Our work provided preliminary results in this context. A more serious investigation is required to make this determination.

Just as boundary representations and CSG are used in solid modeling to meet different functionality requirements, we have to identify such dual schemes whose functionalities complement each other, in the context of parallel machines. The local topology [31, 56] representation scheme is one example of this strategy. For example, instead of defining objects in terms of ordered lists of vertices, edges, and faces, one could attempt to define objects in new tuple-sets. It is clear that such schemes could have a significant impact. However, much work remains to complete this exploration.

REFERENCES

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. ÓDúnlaing, and C. Yap, "Parallel Computational Geometry," *Algorithmica*, vol. 3, pp. 293–327, 1988.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, Massachusetts: Addison-Wesley, 1974.

[3] V. G. Ajjanagadde and L. M. Patnaik, "Design and Performance Evaluation of a Systolic Architecture for Hidden Surface Removal," *Computers and Graphics*, vol. 12, no. 1, pp. 71–74, 1988.

[4] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c \log n$ Parallel Steps," *Combinatorica*, vol. 3, pp. 1–19, 1983.

[5] S. G. Akl, "A Constant-Time Parallel Algorithm for Computing Convex Hulls," *BIT*, vol. 22, pp. 130–134, 1982.

[6] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami, "Geometric Computing and the Uniform Grid Data Structure," *Computer Aided Design*, vol. 21, pp. 410–420, September 1989.

[7] D. C. S. Allison and M. T. Noga, "Some Performance Tests of Convex Hull Algorithms," *BIT*, vol. 24, pp. 2–13, 1984.

[8] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the AFIPS Conference*, pp. 483–485, April 1967.

[9] T. Asano, M. Edahiro, H. Imai, M. Iri, and K. Murota, *Practical Use of Bucketing Techniques in Computational Geometry*, vol. 2 of *Machine Intelligence and Pattern Recognition*, pp. 153–195. Elsevier Science Publishers, 1985.

[10] A. Asthana, H. V. Jagadish, and S. C. Knauer, "An Intelligent Memory Transaction Engine," in *Sixth International Workshop on Database Machines*, (France), pp. 286–300, June 1989.

[11] M. J. Atallah and M. T. Goodrich, "Parallel Algorithms for Some Functions of Two Convex Polygons," *Algorithmica*, vol. 3, pp. 535–548, 1988.

[12] K. E. Batcher, "Sorting Networks and their Applications," in *Proceedings of the AFIPS Spring Joint Computing Conference*, pp. 307–314, 1968.

[13] G. E. Blelloch, "Scans as Primitive Parallel Operations," *IEEE Transactions on Computers*, vol. 38, pp. 1526–1538, November 1989.

[14] G. E. Blelloch and J. J. Little, "Parallel Solutions to Geometric Problems on the Scan Model of Computation," in *Proceedings of the 1988 International Conference on Parallel Processing*, vol. 3, pp. 218–222, August 1988.

[15] K. S. Booth, D. R. Forsey, and A. W. Paeth, "Hardware Assistance for Z-Buffer Visible Surface Algorithms," *IEEE Computer Graphics and Applications*, vol. 6, pp. 31–39, November 1986.

[16] J. E. Bresenham, "Algorithm for Computer Control of Digital Plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[17] C. Brown, R. Fowler, T. LeBlanc, M. Scott, M. Srinivas, L. Bukys, J. Costanzo, L. Crowl, P. Dibble, N. Gafter, B. March, and T. Olson, "DARPA Parallel Architecture Benchmark Study," Tech. Rep. Butterfly Project Report 13, Computer Science Department, University of Rochester, October 1986.

[18] I. Carlbom, "An Algorithm for Geometric Set Operations using Cellular Subdivision Techniques," *IEEE Computer Graphics and Applications*, vol. 7, pp. 44–55, May 1987.

[19] B. Chazelle, "Computatational Geometry on a Systolic Chip," *IEEE Transactions on Computers*, vol. 33, pp. 784–786, September 1984.

[20] B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting line segments in the plane," in *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, (White Plains), pp. 590–600, October 1988.

[21] A. Chow, *Parallel Algorithms for Geometric Problems.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, 1980.

[22] J. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics," *Computer Graphics*, vol. 16, pp. 127–133, July 1982.

[23] E. Cohen, R. Miller, E. M. Sarraf, and Q. F. Stout, "Efficient Convexity and Domination Algorithms for Fine and Medium Grain Hypercube Computers." (to appear in *Algorithmica*).

[24] R. Cole, "Parallel Merge Sort," *SIAM Journal of Computing*, vol. 17, pp. 770–785, August 1988.

[25] S. J. Eggers and R. H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (Boston), pp. 257–270, April 1989.

[26] D. Evans and S. Mai, "Two Parallel Algorithms for the Convex Hull Problem in a two Dimensional Space," *Parallel Computing*, vol. 2, pp. 313–326, 1985.

[27] E. Fiume, A. Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," *Computer Graphics*, vol. 17, pp. 141–149, July 1983.

[28] W. R. Franklin, *Combinatorics of Hidden Surface Algorithms.* PhD thesis, Center for Research in Computing Technology, Harvard University, June 1978.

[29] W. R. Franklin, "A Linear Time Exact Hidden Surface Algorithm," *Computer Graphics*, vol. 14, pp. 117–123, July 1980.

[30] W. R. Franklin, "Efficient Polyhedron Intersection and Union," in *Proceedings of Graphics Interface'82*, (Toronto), pp. 73–80, May 1982.

[31] W. R. Franklin, "Localized Polygon Data Structures," tech. rep., Rensselaer Polytechnic Institute, April 1988.

[32] W. R. Franklin and V. Akman, "Reconstructing Visible Regions From Visible Segments," *BIT*, vol. 26, pp. 430–441, 1986.

[33] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel Planes-5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, vol. 23, pp. 79–88, July 1989.

[34] S. Gaudet, R. Hobson, P. Chilka, and T. Calvert, "Multiprocessor Experiments with High-Speed Ray Tracing," *ACM Transactions on Graphics*, vol. 7, pp. 151–179, July 1988.

[35] D. Ghosal and L. M. Patnaik, "Parallel Polygon Scan Conversion Algorithms: Performance Evaluation of Bus Architectures," *Computers and Graphics*, vol. 10, no. 1, pp. 7–25, 1986.

[36] J. Goldfeather, J. P. M. Hultquist, and H. Fuchs, "Fast Constructive Solid Geometry Display in the Pixel-Power Graphics System," *Computer Graphics*, vol. 20, pp. 107–116, August 1986.

[37] M. T. Goodrich, *Efficient Parallel Techniques for Computational Geometry.* PhD thesis, Purdue University, West Lafayette, Indiana, 1987.

[38] M. T. Goodrich, "Finding the Convex Hull of a Sorted Point set in Parallel," *Information Processing Letters*, vol. 26, pp. 173–179, December 1987.

[39] M. T. Goodrich, "Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors," Tech. Rep. 88-27, The Johns Hopkins University, 1988.

[40] M. T. Goodrich, "Applying Parallel Processing Techniques to Classification Problems in Constructive Solid Geometry," in *Proceedings of the First ACM-SIAM Symposium on Discrete Algorithms*, (San Francisco), pp. 118–128, January 1990.

[41] L. J. Guibas and J. Stolfi, *Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms*, vol. F40 of *NATO ASI Series*, pp. 111–165. Springer Verlag Berlin-Heidelberg, 1988.

[42] J. L. Gustafson, G. L. Montry, and R. F. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, pp. 1–31, July 1988.

[43] C. Ho and L. S. Johnsson, "Optimal Broadcasting and Personalized Communication in Hypercubes," Tech. Rep. YALEU/DCS/TR-610, Yale University, Department of Computer Science, December 1987.

[44] C. M. Hoffman, J. E. Hopcroft, and M. S. Karasick, "Robust Set Operations on Polyhedral Solids," *IEEE Computer Graphics and Applications*, vol. 9, no. 6, pp. 50–59, November 1989.

[45] M. Hu and J. D. Foley, "Parallel Processing Approaches to Hidden Surface Removal in Image Space," *Computers and Graphics*, vol. 9, no. 3, pp. 303–317, 1985.

[46] M. Karasick, *On the Representation and Manipulation of Rigid Solids.* PhD thesis, McGill University, Montreal, Canada, 1988.

[47] C. P. Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," *IEEE Transactions on Computers*, vol. C-34, pp. 965–968, October 1985.

[48] C. P. Kruskal, L. Rudolph, and M. Snir, "Efficient Parallel Algorithms for Graph Problems," *Algorithmica*, vol. 5, pp. 43–64, 1990.

[49] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes, "Constructive Solid Geometry for Polyhedral Objects," *Communications of the ACM*, vol. 20, pp. 18–22, August 1986.

[50] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," in *Proceedings of 16th Annual Symposium on Theory of Computing*, (ACM, New York), pp. 71–80, May 1984.

[51] A. Levinthal and T. Porter, "Chap - A SIMD Graphics Processor," *Computer Graphics*, vol. 18, pp. 77–82, July 1984.

[52] M. Mäntylä, "Boolean Operations of 2-Manifolds through Vertex Neighborhood Classification," *ACM Trans. Graphics*, vol. 5, pp. 1–29, January 1986.

[53] D. J. Meagher, "Geometric Modelling Using Octree Encoding," *Computer Graphics and Image Processing*, vol. 19, pp. 129–147, June 1982.

[54] G. L. Miller and J. H. Reif, "Parallel Tree Contraction and its Application," in *Proceedings of the 26th IEEE Symposium of Foundations of Computer Science*, pp. 478–489, 1985.

[55] R. Miller and Q. F. Stout, "Efficient Parallel Convex Hull Algorithms," *IEEE Transactions on Computers*, vol. 37, pp. 1605–1618, December 1988.

[56] C. Narayanaswami and W. R. Franklin, "Determination of Mass Properties of CSG Trees in Parallel." (in preparation).

[57] C. Narayanaswami and M. Seshan, "The Efficiency of the Uniform Grid for Computing Intersections," Master's thesis, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, NY, December 1987.

[58] H. Niimi, Y. Imai, and M. Murakami, "A Parallel Processor System for Three-Dimensional Color Graphics," *Computer Graphics*, vol. 18, pp. 67–76, July 1984.

[59] M. T. Noga, "Sorting in Parallel by Double Distributive Partitioning," *BIT*, vol. 27, pp. 340–346, 1987.

[60] M. Potmesil and E. M. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, vol. 23, pp. 69–78, July 1989.

[61] F. P. Preparata and M. I. Shamos, *Computational Geometry.* New York: Springer-Verlag, 1985.

[62] D. Pullar, "Comparative Study of Algorithms for Reporting Geometrical Intersections," in *Proceedings of the International Symposium on Spatial Data Handling*, (Zurich), pp. 66–75, July 1990.

[63] L. K. Putnam and P. A. Subrahmanyam, "Boolean Operations on n-Dimensional Objects," *IEEE Computer Graphics and Applications*, vol. 6, pp. 43–51, June 1986.

[64] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill Book Company, 1987.

[65] A. J. Ranade, "How to Emulate Shared Memory," in *28 IEEE Symposium of Foundations of Computer Science*, pp. 185–195, 1987.

[66] J. H. Reif, "Depth First Search is Inherently Sequential," *Information Processing Letters*, vol. 20, pp. 229–234, June 1985.

[67] A. A. G. Requicha and H. B. Voelcker, "Boolean Operation in Solid Modeling: Boundary Evaluation and Merging Algorithms," in *Proceedings of the IEEE*, vol. 73, pp. 30–44, January 1985.

[68] J. R. Rossignac and H. B. Voelcker, "Active Zones in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection, and Shading Algorithms," *ACM Transactions on Graphics*, vol. 8, pp. 51–87, January 1989.

[69] H. Samet and R. E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics," *IEEE Computer Graphics and Applications*, vol. 8, pp. 48–68, 1988.

[70] H. Sato, M. Ishii, K. Sato, and M. Ikesaka, "Fast Image Generation of Constructive Solid Geometry Using a Cellular Array Processor," *Computer Graphics*, vol. 19, pp. 95–102, July 1985.

[71] M. Segal and C. H. Sequin, "Partitioning Polyhedral Objects into Nonintersecting Parts," *IEEE Computer Graphics and Applications*, vol. 8, pp. 53–67, January 1988.

[72] M. Tam, J. M. Smith, and D. J. Farber, "A Survey of Shared Memory Systems." (submitted for publication).

[73] T. Theoharis, *Algorithms for Parallel Polygon Rendering.* Berlin: Springer-Verlag, 1989.

[74] W. C. Thibault and B. F. Naylor, "Set Operations Using BSP Trees," *Computer Graphics*, vol. 21, pp. 153–162, July 1987.

[75] J. G. Torborg, "A Parallel Processor Architecture for Graphics Arithmetic Operations," *Computer Graphics*, vol. 21, pp. 197–204, July 1987.

[76] K. Weiler, *Topological Structures for Geometric Modeling.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, August 1986.

[77] R. Weinberg, "Parallel Processing Image Synthesis and Anti-Aliasing," *Computer Graphics*, vol. 15, pp. 55–61, August 1981.

[78] F. Yamaguchi, "A Unified Approach to Interference Problems Using a Triangle Processor," *Computer Graphics*, vol. 19, pp. 141–149, July 1985.

[79] F. Yamaguchi and T. Tokieda, "A Unified Algorithm for Boolean Shape Operations," *IEEE Computer Graphics and Applications*, vol. 4, pp. 24–37, June 1984.

[80] F. Yamaguchi and T. Tokieda, "A Solid Modeler with a 4 X 4 Determinant Processor," *IEEE Computer Graphics and Applications*, vol. 5, pp. 51–59, April 1985.

[81] C. K. Yap, "What can be Parallelized in Computational Geometry?," in *Proceedings of the International Workshop on Parallel Algorithms and Architectures*, May 1987.