Exploiting Parallelism in n-D Convex Hull Algorithms

by

Edet Okon Eyoh

PhD Thesis

September 1994
Abstract

The convex hull is a problem of primary importance because of its applications in computational geometry. A number of sequential and parallel algorithms for computing the convex hull of a finite set of points in the lower dimensions are known. In comparison, the general n-D problem is not as well understood and parallel algorithms are not so prevalent because the 2-D and 3-D methods are not easily extended to the general case. This thesis presents parallel algorithms for evaluating the general n-D convex hull problem (where 2-D and 3-D are special cases) using Swart’s sequential algorithm. One of our methods combines a gift-wrapping technique with partitioning and merge algorithms where the original list is split into \( p > 1 \) partitions followed by the computation of the subhulls using the sequential n-D gift-wrapping method. The partial hulls are then combined using a fanin tree. The second method computes the convex hull in parallel by wrapping around the edges until a complete facial lattice structure of the polytope is generated.

Several parameterised versions of the proposed algorithms have been implemented on the shared memory and message passing architectures. In the former, performance on an Encore Multimax using Encore Parallel Threads and the more lightweight Microthread programming utilities are examined. In the latter, performance on a transputer based machine using CS-Tools is discussed. We have shown that our techniques will be useful in the construction of faster algorithms which employ the n-D convex hull algorithms as a sub-algorithm.
Acknowledgements

I sincerely thank my supervisor, Dr. G. M. Megson, for suggesting this area of research and for reading and commenting upon the numerous drafts of this thesis.

I would also like to express my appreciation to Dr. John L. Lloyd and Dr. Chris Phillips for their suggestions at different levels of the work. My thanks also go to the members of the Algorithm Engineering Research Group and other colleagues in the department, their efforts were greatly appreciated. I am grateful to the then University of Cross River State, Uyo, Nigeria (now University of Uyo) for granting me a study leave during the period of this research.

Financial support for this research was provided by the Commonwealth Scholarship Commission in the United Kingdom.

Finally, I thank my family for their support and encouragement which they gave me during my studies particularly my wife Unyime and Ekomobong my daughter.
Declaration

I certify that no part of the original material offered here in this thesis has been previously submitted by me for a degree or other qualification in this or any other university.

EDET OKON EYOH
Contents

Abstract

Acknowledgements i

Declaration ii

Contents iii

List of Figures viii

List of Tables xiii

1 Introduction 1

2 Parallel Architectures And Their Implementations 6

2.1 SISD Architectures ............................. 8
2.2 MISD Architectures ............................. 9
2.3 SIMD Architectures ............................ 11
2.4 MIMD Architectures ............................ 12
2.5 Multiprocessors ................................. 13
2.6 Multicomputers ................................. 16
   2.6.1 Transputer ................................. 17
   2.6.2 Systolic Array Architectures .............. 18
5.7 Results From Partitioning Methods ........................................... 126

6 Facial Lattice Exploration (FLE) ................................................. 128

6.1 FLE On Shared Memory (Version 1) ......................................... 134
  6.1.1 Results ................................................................. 140

6.2 FLE On Shared Memory (Version 2) ......................................... 144
  6.2.1 Results From Shared Memory .......................................... 153

6.3 Transputer Implementation Of FLE ........................................... 156
  6.3.1 Results From Distributed Memory .................................... 168

7 Conclusions and Summary .......................................................... 173

Bibliography ................................................................................. 182

Appendices .................................................................................... 192

A Tables ....................................................................................... 192

B Graphs ...................................................................................... 226

C Some Program Listings ............................................................... 247

  C.1 Definitions for the convex hull program ................................. 248
    C.1.1 Routines for manipulating Points ................................. 248
    C.1.2 Routines for manipulating Edge/Face lists ..................... 253
    C.1.3 Points Sorting .......................................................... 256
    C.1.4 Generate_Hull(S, n, CH, FA) ....................................... 258
    C.1.5 int Check_Hull(S, Faces, n) ....................................... 259
    C.1.6 POINTS Remove_Duplicate_Points(S, n) ....................... 260
    C.1.7 Generate_Bounds(small, large, S, n) ............................. 260
    C.1.8 Simple Matrix And Vector Manipulation ....................... 261
C.1.9  Rotate(S, AS, n, k, F, norm, J) ................................. 264
C.1.10 Initial_facet(S, AS, n, k, F, norm) ................................. 269
C.1.11 POINTS Affine_Hull(S, n, A, k) ................................. 271
C.1.12 int Check_Plane(S, n, norm, P0) ................................. 273

C.2  Test Data Generators ............................................. 274
C.2.1  Generate_Test(CH, FA, n, npts) ................................. 274
C.2.2  Test To Generate Circular Structure ............................. 275
C.2.3  Test To Generate Rectangles In Levels .......................... 276

C.3  Routine For Distributed Memory Architecture .................... 278
C.3.1  List Communication Primitives On Transputer .................. 278
C.3.2  Build Files For Partitioning Method ............................. 280
C.3.3  Build Files For FLE ............................................ 281
# List of Figures

2.1 General Structure of SISD Architecture ........................................ 9  
2.2 General Structure of MISD Architecture ........................................ 10  
2.3 General Structure of SIMD Architecture ....................................... 11  
2.4 General Structure of MIMD Architecture ....................................... 13  
2.5 Bus Interconnection ......................................................................... 15  
2.6 Transputer Network ........................................................................ 17  
2.7 Systolic flow of data to and from memory ........................................ 18  
2.8 Dataflow Graph ................................................................................ 19  
3.1 Angle Between Three Points ............................................................. 43  
3.2 2-D Convex Hull Illustration ............................................................ 44  
3.3 2-D Convex Hull For Divide-And-Conquer ...................................... 47  
3.4 Upper and lower tangent lines between $S_1$ and $S_2$ ...................... 53  
3.5 A partitioning of $S$ into 5 subsets .................................................. 55  
3.6 Illustration Of Merge Procedure ...................................................... 56  
4.1 Transformation Of 3-D To 2-D square .............................................. 66  
4.2 Set $S$ Projected Onto The Plane Of $e$ And $\hat{n}$ .............................. 67  
4.3 Stack Implementation For 3-D .......................................................... 76  
4.4 Types Of Shapes In 2-D ..................................................................... 81  
4.5 Illustration of 3-D Circular Shape .................................................... 83
6.1 Facial lattice of a pyramid over a square ........................................ 129
6.2 Illustration Of Parallel Execution Of 3D Convexhull by FLE .............. 130
6.3 Edge Computation In A 2-D Problem .............................................. 132
6.4 Exploiting Parallelism With Unlimited Number Of Processors .......... 133
6.5 FLE Implementation Using Global Lists ......................................... 146
6.6 FLE For 2-D On Shared Memory .................................................... 156
6.7 FLE For 3-D Circular Shape On Shared Memory ............................... 157
6.8 FLE For 3-D Pyramidal Shape On Shared Memory ............................. 157
6.9 Communication Between Processes on Different Transputers ............. 159
6.10 Master And Manager Run On Same Transputers ............................... 168
6.11 FLE For 2-D On Transputer (Ver 1) .............................................. 170
6.12 FLE For 2-D On Transputer (Ver 2) .............................................. 171
6.13 3-D Circular Shape On Transputer (Ver 1) ..................................... 171
6.14 3-D Circular Shape On Transputer (Ver 2) ..................................... 172

7.1 Partitioning of GNORMS into three sublists ................................. 181

B.1 Recursive Version 2D 3 Vertices Using Threads ............................ 227
B.2 Recursive Version 2D 4 Vertices Using Threads ............................ 227
B.3 Recursive Version 2D 6 Vertices Using Threads ............................ 228
B.4 Recursive Version 2D 16 Vertices Using Threads .......................... 228
B.5 Recursive Version 3D 3 Vertices Using Threads ............................ 229
B.6 Recursive Version 3D 4 Vertices Using Threads ............................ 229
B.7 Recursive Version 3D 6 Vertices Using Threads ............................ 230
B.8 Recursive Version 3D 12 Vertices Using Threads .......................... 230
B.9 Recursive Version 4D 4 Vertices Using Threads ............................ 231
B.10 Recursive Version 2D 3 Vertices Using Microthreads .................... 231
B.11 Recursive Version 2D 4 Vertices Using Microthreads ............................................. 232
B.12 Recursive Version 2D 6 Vertices Using Microthreads ............................................. 232
B.13 Recursive Version 2D 16 Vertices Using Microthreads ........................................... 233
B.14 Recursive Version 2D 26 Vertices Using Microthreads ........................................... 233
B.15 Recursive Version 3D 3 Vertices Using Microthreads ............................................. 234
B.16 Recursive Version 3D 4 Vertices Using Microthreads ............................................. 234
B.17 Recursive Version 3D 6 Vertices Using Microthreads ............................................. 235
B.18 Recursive Version 4D 4 Vertices Using Microthreads ............................................. 235
B.19 Recursive Version 4D 6 Vertices Using Microthreads ............................................. 236
B.20 Stack Version 2D 3 Vertices Using Threads .......................................................... 236
B.21 Stack Version 2D 4 Vertices Using Threads .......................................................... 237
B.22 Stack Version 2D 6 Vertices Using Threads .......................................................... 237
B.23 Stack Version 2D 16 Vertices Using Threads .......................................................... 238
B.24 Stack Version 3D 3 Vertices Using Threads .......................................................... 238
B.25 Stack Version 3D 4 Vertices Using Threads .......................................................... 239
B.26 Stack Version 3D 6 Vertices Using Threads .......................................................... 239
B.27 Stack Version 3D 12 Vertices Using Threads .......................................................... 240
B.28 Stack Version 4D 4 Vertices Using Threads .......................................................... 240
B.29 Stack Version 2D 26 Vertices Using Microthreads ................................................ 241
B.30 Stack Version 3D 6 Vertices Using Microthreads ................................................ 241
B.31 Stack Version 4D 6 Vertices Using Microthreads ................................................ 242
B.32 Recursion Version 1 3D 6 Vertices Using Transputer ........................................... 242
B.33 Recursion Version 1 3D 12 Vertices Using Transputer ........................................... 243
B.34 Stack Version 4D 6 Vertices Using Microthreads ................................................ 243
B.35 Recursion Version 1 3D 6 Vertices Using Transputer ........................................... 244
B.36 Recursion Version 1 3D 12 Vertices Using Transputer ........................................... 244
B.37 Stack Version 1 3D 6 Vertices Using Transputer .......................... 245
B.38 Stack Version 1 3D 12 Vertices Using Transputer .......................... 245
B.39 Recursion Version 2 3D 6 Vertices Using Transputer ......................... 246
B.40 Recursion Version 2 3D 12 Vertices Using Transputer ......................... 246
List of Tables

2.1 Properties Of Multiprocessors .................................... 16
2.2 Examples Of Some Architectures ................................... 22
3.1 Set Of Points To Illustrate 2-D Algorithms ...................... 44
3.2 Sequential Running Time For 2-D and 3-D Algorithms ........... 49
3.3 Parallel 2-D Complexity Table .................................... 58
4.1 Test Data .............................................................. 80
5.1 Partitioning (2-D 26vertices, 4-D 6vertices, with 1000points) On Multimax 126
5.2 Statistics For Partitioning Methods From Table 39 .............. 127
6.1 Movement Of Edges Into Pend.List ............................... 141
6.2 Movement Of Edges Into Pend.List ( Table 6.1 Cont.) ........... 142
6.3 Results For FLE Version 1 .......................................... 143
6.4 Timing for 2-D ....................................................... 155
6.5 Timing for 3-D Circular Structure ................................ 155
6.6 Timing for 3-D Pyramidal Structure Generating Rectangle In Levels .... 156
6.7 Timing for 2-D ....................................................... 169
6.8 Timing for 3-D Pyramidal Structure ................................ 170
6.9 Timing for 3-D Circular Structure ................................ 170
Chapter 1

Introduction

Sometimes when we have a large number of points to process, we are interested in finding the boundaries of the points so that all other points will be interior to the boundary points. If these points are plotted on a diagram, it takes very little time to find out their positions relative to the chosen origin. The mathematical name for the natural boundary of a point set is the convex hull. This is defined to be the smallest convex polygon containing all the points. Equally, it could be considered as the shortest path surrounding all the points. The points or vertices of the convex hull are points from the original set.

For the 2-D case, the concept of a convex hull is natural and easy to understand. If $S$ consists of a finite set of points in the plane, consider surrounding the set by a large, stretched, rubber band. When the rubber band is released, it will assume the shape of the boundary points which is the convex hull of $S$. In this case the boundaries of the resulting polygon are made up of straight lines whose points of intersection give the vertices of the hull. For greater than two dimensions, the polytope is bounded by faces, and the intersection of two faces gives rise to an edge. For example a cube has six faces, twelve edges and eight vertices. If one edge and one of the faces containing this edge is known, then another face can be generated by a rotation of the known face about the known edge. A repetition of a number of rotations will be necessary to eventually produce a complete description or facial lattice structure of the object.
These intuitive and simple definitions hide the fact that the convex hull is a geometric structure of primary importance in computational geometry. It has important applications in computer-aided design, computer graphics, image generation, and operations research [4, 5, 61]. In graphics applications, the interest is in determining the edges that uniquely describe the object. In Linear Programming (LP) we consider maximising (or minimising) some linear functional over a polyhedron defined by,

$$\max \sum_{j=1}^{n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, 2, \ldots, m)$$

$$x_j \geq 0, \quad (j = 1, 2, \ldots, n).$$

Any solution of $Ax = b$ which is non negative gives a feasible solution to the optimisation problem and these solutions are the vertices of the convex hull. One of these vertices which maximises the objective function is the optimal solution. Other applications include simulating chemical reactions or estimating population parameters in Statistics which often require the calculation of the convex hull in a dynamic fashion [3]. The depth of a point $p$ in a set $S$ can be considered as the number of convex hulls (convex layers) that have to be stripped from $S$ before $p$ is removed. In graphics applications, the dimension $n \leq 3$ is usual. But in problems involving principal component analysis and clustering applications (such as quality testing) and automatic analysis of data dependency by parallel compilers $n > 3$ is common. Most recently, the convex hull is also being used in automatic synthesis of parallel algorithms where nested loops are regarded as geometric objects and whose computations are defined inside a convex polytope [6]. The partitions of the hull and mapping of partitions into processor arrays often requires the construction of convex hulls to define loop bounds for the code.
In two (2-D) and three (3-D) dimensions, quite a lot has been achieved in computing the convex hull both sequentially and in parallel. Some of these contributions are considered in more detail in chapter three. For the higher dimensions i.e. for \( n > 3 \), the problem is less well understood and hence relatively little research has appeared in the literature on computing the convex hull. In 2-D as well as in the higher dimensional cases, the approaches adopted by the different authors are mainly theoretical with each paying particular attention to the analysis of the expected time performance of their algorithms. Only a few authors have given consideration to practical implementations of these algorithms. Also significant effort has been devoted to designing parallel methods for solving the 2-D and 3-D problems in the shared memory and distributed memory architectures which use the divide-and-conquer paradigm to achieve an optimal time bound.

Unfortunately, the methods used for identifying the convex hull for 2-D and 3-D problems cannot be directly extended to compute the convex hull for the n-D problem. Attempting to scale the methods to higher dimensions will result in increased computing time. This is because of the combinatorial nature of the n-D problem. However, with renewed interest and development in the field of Computational Geometry, researchers have become more interested in the n-D convex hull problem. Chand \& Kapur [1] paved the way in their paper by proposing a sequential algorithm for the n-D problem. Swart [2] then modified this algorithm to improve its performance. So far, there is no concerted effort on parallel implementation of the n-D convex hull problem. Motivated by this slow pace of work, and with the availability of parallel machines, the main theme of this research has centered on the development and implementation of parallel algorithms for the n-D convex hull problem.

In light of the above, the main contribution of this research is to present parallel methods for evaluating the n-D convex hull algorithm on both shared memory and message passing architectures. The approaches to be adopted in the study are as follows:
• Use an existing algorithm based on Chand and Kapur's wrapping technique [1] modified to use the affine basis method as described by Swart [2] and extend to a parallel implementation.

• Levels of implementation: This will be considered in two stages

  1. A partitioning approach.

  2. A method to explore the facial lattice of the convex hull.

The recursive and non-recursive versions of this algorithm are implemented on the chosen architecture. The C programming language is used in coding the algorithm, and

• Encore Parallel Threads (EPT) and microthreads [54] on the shared memory machine.

• CS-Tools [97] on a message passing transputer architecture.

The aim is that by implementing the parallel version of the n-D problem we will provide a substantial improvement over the sequential algorithm. Hence we conclude that our algorithms will prove useful in the construction of faster algorithms which employ the n-D convex hull as a sub-algorithm.

The rest of this thesis is organised into six chapters. Chapter two presents an overview of parallel architectures. Of special interest are the discussions on the shared memory and message passing architectures on which our convex hull algorithms are to be implemented. A brief discussion on EPT and CS-Tools is also given. Chapter three gives a brief definition of the terms to be used in the discussion to aid the understanding of subsequent chapters, and reviews some of the major approaches to the convex hull problem so far and indicates the areas that might benefit from further research. Chapter four focuses on the sequential version of the n-D convex hull algorithm which is the basis of our parallel implementations and discusses various features of the test data and test generation. The main supporting
routines are also discussed. In chapter five, we discuss several parameterised versions of the proposed parallel algorithms as implemented on shared memory and message passing architectures. The techniques are based on partitioning of the data using a divide-and-conquer method followed by a merge procedure to produce a solution to the problem. In the shared memory architecture we use a fanin tree approach and simulate the tree level by level. In the message passing architecture, we also simulate the tree based on a master-slave relationship. An alternative method pipelines the partitions through the architecture by constructing a tree in hardware. We present a summary of the results from our implementations with \( p > 1 \) processors on both the shared and distributed memory machines. Chapter six looks at a method based on facial lattice exploration by wrapping around the edges until a complete facial lattice structure of the polytope is generated. It uses the stack version which is better for some shapes than the recursive approach and is implemented again on both the shared memory and message passing architectures. Results of these implementations are discussed. Finally, chapter seven provides an overall summary of the thesis and suggests possible areas for future work.
Chapter 2

Parallel Architectures And Their Implementations

Parallel computers are computers that emphasise concurrent manipulation of data elements belonging to one or more processes solving a single problem. Algorithms designed for implementation on parallel computers are called parallel algorithms. The essence of the parallel version of any algorithm is to obtain a significant speed-up over the sequential version. To date the major set back on rapid introduction of parallel computing has been the huge investments already made in software for sequential machines and the lack of good parallel processing software to aid design and development. There are reasons, however, why parallel processing is gaining widespread attention. Parallel processing is intended to be used for applications that require massive amounts of data manipulation. Such problems include real time simulations of complex systems, artificial intelligence, weather forecasting, computational aerodynamics, energy resource exploration, medical, military and in basic research among others [22, 23]. Using fast and efficient computers makes these simulations far cheaper and faster than physical laboratory experiments and enables the solution of a wider range of problems and these machines are thus cost effective. Computational ability is only limited by computer speed and memory capacity whereas physical experiments are subjected to many constraints. An algorithm whose order of magnitude time performance is bounded by a polynomial function of \( N \) (e.g. log-
arithmic, linear and quadratic etc.) where $N$ is the size of input. is called a polynomial time algorithm and is said to be a reasonable algorithm. Similarly, an algorithm that, in the worst case, requires an exponential time will be considered unreasonable (e.g. $N!$, $N^N$, $2^N$). As far as algorithmic problems are concerned, a problem that admits a reasonable or polynomial solution is said to be tractable, whereas problems with unreasonable or exponential time solution are termed intractable. Sorting is an example of a tractable problem and the Towers of Hanoi problem with at least 64 rings or more, is hopelessly time consuming [59].

Parallel algorithms and programs are closely connected with the architecture of parallel computers, and therefore design and analysis of parallel algorithms and programs cannot be considered independently of their implementation and the architecture of the computer on which they are to be implemented. Unlike in serial computation, where the Random Access Machine (RAM) is used, one generic model of computation has not been found for the design and analysis of parallel algorithms. Although the Parallel RAM (PRAM) has gained a lot of popularity as a general model of parallel computers [23], it is not easy to use for all applications. This chapter examines some of the parallel computers currently available. It is not the intention of this thesis to examine the various categorisations in detail since the emphasis is on algorithms rather than hardware. The purpose here is to present an overview of some of the parallel computer architectures and in particular those on which our algorithms are to be implemented.

Most research in design and development of parallel algorithms has come about as a result of the availability of different models of parallel computers. In order to properly design these algorithms, a clear understanding of the model of the underlying parallel computer is required. Many methods of categorisation have been proposed in the literature [7 – 14] and one of the earliest was Flynn’s [7] taxonomy which classifies architectures according to the presence of instruction and data streams. Although this classification
is limited in terms of recent developments in the field, resulting in new architectural
models, it nevertheless provides the basis of most schemes. The four major categories are
as follows:

- Single Instruction Stream - Single Data Stream SISD
- Multiple Instruction Stream - Single Data Stream MISD
- Single Instruction Stream - Multiple Data Stream SIMD
- Multiple Instruction Stream - Multiple Data Stream MIMD

Flynn’s classification is very general in nature and does not reveal some important de-
tails of a number of systems e.g. many processors have arithmetic or instruction pipelines
or both and Flynn does not distinguish processors of this type. Händler [13] and Hwang & Briggs [23] stress the availability of pipelining and the number of pipeline stages. An-
other classification scheme proposed by Feng [11] stresses the degree of parallelism i.e. the
maximal number of bits that can be processed within a time unit by a computing system
e.g. the Carnegie Mellon C.mmp is a multiprocessor consisting of 16 processors of 16-bit
wordlength. Duncan [9] has extended the scope to include Systolic Arrays, Dataflow and
Reduction machines. Although we shall not make use of them in the rest of the thesis we
will briefly outline their characteristics to place the work in context.

2.1 SISD Architectures

Computers in this group consist of a single processing element (PE) receiving single
streams of instructions (IS) from the control unit (CU) that operates on a single stream
of data (DS) from the memory (M). This is illustrated in Figure 2.1. At each step during
the computation, the control unit executes a single instruction that operates on a single
datum from memory. Instructions tell the processor the operations to be performed on
the data and subsequently put it back in the memory. The majority of the present day computers are in this group, often termed Von Neumann architectures (because they were invented by John von Neumann [16]). Serial or sequential algorithms are implemented on SISD machines.

**Example 1:** Consider the problem of multiplying \( n \) numbers. The processor needs to gain access to the memory \( n \) times in order to obtain the \( n \) data items. It also performs \( (n - 1) \) multiplications in sequence which requires an order of \( n \) operations in total. The IBM 7090 is an example of a SISD computer.

### 2.2 MISD Architectures

In this case there are multiple processors, with each processor having its own control unit but sharing a common memory where data resides. Figure 2.2 is a representation of this type of architecture. There are multiple streams of instruction and a single data stream. Parallelism is achieved by letting each processor do different things concurrently on the same data. These computers are best suited to computations that require a single input to be subjected to numerous different operations each receiving the input in the same original form.
Example 2: Suppose we want to classify objects according to some set of predefined rules. The objects could be mathematical, for instance, where a number could be associated with one of several sets, each satisfying its own criteria. Alternatively, the objects could be physical ones (e.g. students, lecturers and civil servants) trying to recognise objects in order to classify them. The member (single data) of the objects is usually subjected to many different tests (multiple instructions) in order to group them properly. MISD computers prove useful as each processor is associated with each class and can recognise members of that class after subjecting the member to a number of computational tests. Each member (data) is sent at the same time to each processor where it is tested against the set criteria in parallel.

At the same time, the computation appears to be of a rather specialised nature and hence very limited in use. Parallel computers that are more flexible and hence suitable for a wide range of applications would be preferred.
2.3 SIMD Architectures

SIMD architectures typically employ a central control unit, multiple processors, and inter processor connection network. A single instruction is broadcast to all processors by the control unit and the results are communicated between the processors from the interconnection network. The model is shown in Figure 2.3. SIMD machines can be subdivided into Array Processors [8, 18] suitable for large scale numerical calculations such as image processing and nuclear energy modelling. SIMD machines consist of synchronised Processor Elements (PE's) under the control of one control unit. Each PE has working registers and local memory. Examples include Loral's Massively Parallel Processor MPP [24] and Illiac IV [25] and recently DAP [22]. Associative Memory architectures [25] use special logic to access stored data in parallel according to its contents. They are geared towards data based oriented applications, such as tracking and surveillance. Examples include Bell Laboratories' Parallel Processing Element Ensemble (PEPE) and Loral's Associative Processor (Aspro) [22].

Figure 2.3: General Structure of SIMD Architecture
Example 3: Let's consider a very large unsorted file with \( n \) items. Suppose that a certain item \( y \) is required in order to perform an operation. On a SISD computer, retrieving \( y \) requires \( n \) steps in the worst case when \( y \) is the last item in the file. If the file entry is uniformly distributed over a given range, then the processing time can be greatly reduced, for instance on a SIMD architecture with \( p > 0 \) processors. The item \( y \) (single data) needs to be broadcast to all the processors. The file to be searched is subdivided into smaller files of size \( n/p \), say, of approximately equal number of entries and are searched simultaneously by the processors. The processor that finds \( y \) returns its result and signals the other processors that \( y \) has been found and that they can terminate their execution. This task requires \( O(n/p) \) steps compared with a sequential time of \( O(n) \) steps.

2.4 MIMD Architectures

This is the most general and most popular design among parallel computers. Here we have multiple instructions and multiple data streams on different processors. Each processor operates under the control of instruction streams issued by its control unit. The processors execute different parts of the program on different data and cooperate by solving different subproblems of a single problem. Communication is through shared memory (SM) or an interconnection network (ICN) (message passing). Processors sharing a common memory are referred to as multiprocessors while those with a local memory are called multicomputers or Distributed Memory machines. MIMD computers support higher levels of parallelism than can be exploited by 'divide and conquer' algorithms organised as largely independent subcalculations. Later we shall employ MIMD architectures to implement the parallel version of n-D convex hull algorithms using a master-slave organisation. An example of a shared memory paradigm is the Encore Multimax [8] and the Sequent Machines while the Transputers typify the Distributed Memory Machines [8]. The two
subclasses namely the Multiprocessors and Multicomputers are briefly examined to draw out the differences and similarities between them since this is of interest to us. A general structure of MIMD architecture is shown in Figure 2.4.

**Example 4:** Consider the problem of finding the sum of \( n \) numbers. With a SISD machine, the processor will access the memory \( n \) times to receive the numbers. The sequential execution also requires \((n - 1)\) additions. In a MIMD architecture, using \( p \) processors, we can partition the problem into \( n/p \) subproblems or tasks. Each task is now mapped to a processor and all the subproblems will be executed simultaneously each producing a partial sum. The partial sum can now be added together in a treelike fashion to give the final solution to the problem. This requires an \( O(n/p + \log_2 p) \) steps, where \( p \) is the number of leaves in the tree. The tree structure is simulated on the \( p \) processors.

### 2.5 Multiprocessors

This class, also called Parallel Random Access Machine (PRAM) [8] or tightly Coupled machines, share a common memory in the same way a group of people might share a notice board. If two processors want to communicate, the first processor first writes the
item into the shared memory location known to the second processor which then reads the item from that location. Allowing multiple read accesses to the same address in memory should in principle pose no problems. Conceptually, each of the several processors reading from that location makes a copy of the location’s contents and stores it in its own local memory. There are three classes of such machine, depending on the kind of memory contention tolerated. These are EREW (exclusive read/exclusive write), which requires that at any time any memory cell should be accessed by at most one processor. CREW (concurrent read/exclusive write) will allow any number of processors to read the same memory cell simultaneously, but not to write to the cell simultaneously. The third model, (concurrent read/concurrent write) CRCW machine, allows simultaneous read and write access. If several processors attempt to write to the same location, then only one of them succeeds, and the successful processor is chosen arbitrarily. Each processor in addition to a shared memory also has a local memory used as a cache where multiple copies of the shared data may exist at a given time. There are three major alternatives for connecting multiple processors to the shared memory and these are Bus Interconnection [35], Crossbar [21] and Multistage Interconnection Network (MIN) [31 - 33].

A bus system (figure 2.5) contains one or more buses on which the system components are connected. A single bus is the simplest and least expensive to implement and is flexible as components can be added to or disconnected from the bus. Time-shared buses offer a fairly simple way to give multiple processors access to a shared memory. A simple time-shared bus effectively accommodates a moderate number of processors since one processor accesses the bus at a given time. In the Encore Multimax, such a bus is the Nanobus. Since the bus is the potential bottleneck preventing physical expansion of the system beyond a certain limit, extension of the single bus architecture is required to increase the capacity of the bus-based parallel processing systems. The Nanobus of the Encore's Multimax system [8] is a backplane bus that delivers a usable throughput
of 100Mbytes/sec. Present systems have 2 to 20 National Semiconductor NS32332s connected to the backplane Nanobus, providing up to 40MIPS of processing power with up to 128Mbytes of shared memory. The Encore Multimax on which our experiments will be conducted is a structured architecture running the UMAX operating system and containing 14, NS32332 processors each with 256Kb processor cache memory. Its major setback is the bus bottleneck. However this could be avoided if multiple buses are implemented so that failure of a single bus will not cause a total failure of the whole system. On the other hand, the multiple bus implementation requires multiporting which is expensive. The Crossbar permits the concurrent communication and link between all processors and memory modules to be established. Multiple accesses of memory modules are possible as long as they are accessing different locations. This class of multiprocessors has a high throughput resulting from multiple, concurrent communication paths. Reducing the communication overhead is the main concern of designing an efficient communication system. Multistage Interconnection Networks (MIN) attempt to strike a compromise between the price and performance alternatives offered by Crossbar and buses. An $N \times N$ MIN connects $N$ processors to $N$ memories by deploying multiple stages or banks of switches in

Figure 2.5: Bus Interconnection
the interconnection network pathway. A processor making a memory access request specifies the desired destination (the pathway) by issuing a bit-value that contains a control bit for each stage. Table 2.1 [52] summarises the properties of the three categories of the multiprocessors mentioned above.

2.6 Multicomputers

Multicomputers are also called Loosely Coupled or Distributed Memory machines. The distinction between the multicomputers and distributed memory machines lies on the physical distance separating the processors. If the processors are in close proximity they are called multicomputers otherwise they are termed distributed systems. For example if the processors are in the same room they are termed multicomputers but if they are in different cities they are distributed systems. This is important when evaluating parallel algorithms, because the processors in a distributed system are far apart. If the number of data exchanges between them is significantly more than the number of computational steps performed by any of them then the performance will be affected. The Distributed memory architectures are further subdivided into Ring topology structure [27], Mesh computers [29], Pyramid topology [28], Mesh-of-tree [25], Hypercube [17, 26] and Reconfigurable architecture [30] according to the way the processors are connected.
2.6.1 Transputer

A transputer is a microcomputer with its own local memory and with links for connecting one transputer to another. A typical member of the transputer family is a single chip containing processor, memory, and communication links which provide point to point connection between transputers. In addition each transputer contains special circuitry and interfaces adapting it to a particular use. A transputer can be used in a single processor system or in networks to build high performance concurrent systems. A network of transputers and peripheral controllers is easily constructed using point-to-point communication as shown in figure 2.6. The point-to-point connection allows transputer networks of arbitrary size and topology to be constructed. There is no contention for the communication mechanism, regardless of the number of transputers in the system. There is no capacitive load penalty as transputers are added to a system and the communication bandwidth does not saturate as the size of the system increases. In particular, our experimental work was carried out on a Meiko system [29] which uses T800 transputer processors each with a memory capacity of 4MB. This system has 16 transputers each with four bi-directional
2.6.2 Systolic Array Architectures

This group of SIMD/MIMD computers proposed by Kung [15] solves problems mainly in special purpose systems. The basic principle involves the pumping of data from memory through processor elements (cells) and back to memory as shown in Figure 2.7. Once a data item enters the systolic array from memory or an external device, it is passed to any processor element that needs it. Systolic Arrays apart from their applications in Linear Algebra (e.g. matrix product, inverses, triangularization) also find application in medical image and signal processing algorithms [8]. Examples include Carnegie Mellon’s Warp [36 - 38]. They can be reconfigured into different topologies to suit applications but are very special purpose in nature.

2.6.3 Dataflow Architectures

Dataflow machines (Data - Driven) [19, 40] employ an execution paradigm in which instructions are enabled for execution as soon as all their operands become available instead
of following the sequence dictated by the ordering of program instructions. The sequence of instructions are based on data dependencies allowing the architecture to exploit parallelism at task, routine and instruction levels. Data Driven machines are designed to execute dataflow graphs in which the nodes represent the operations (such as multiplication, addition) and the arcs denote the data dependencies between the functions. Figure 2.8 illustrates a dataflow graph. A dataflow graph is made up of operators (actors) connected by arcs that convey data. In figure 2.8 the actors are drawn as circles with the function symbols of +, - and * representing addition, subtraction and multiplication respectively. The arcs convey inputs $a$ and $b$ and the output arcs will carry tokens being values computed by the previous actors. When all the values are present in the input arcs and none in the output arc, the actor is enabled or fired. Node 1 and Node 2 compute the sum and difference of $a$ and $b$ respectively and then pass on the tokens to Node 3 where the product is computed.

There are two types of Dataflow architectures:

- Static Dataflow Machines where all the graph nodes are loaded into the memory
during initialisation and which allow one instance of the node to be executed at a time.

- Dynamic Dataflow Architecture permits the creation of node instances at run time and multiple instances of a node can be executed concurrently.

Examples of Dataflow machines are Manchester Dataflow Computer [34] and the MIT Tagged Token Dataflow Machine [33].

2.6.4 Reduction Machines

Reduction or Demand-Driven [20] architectures seek to reduce an expression in a programming language to its final result. An instruction is executed when its result is needed by an operand for another instruction which is ready to execute and not when their operands are ready as in dataflow. Programs are viewed as nested applications and execution proceeds from the innermost application until there are no further calculations. Thus they are good for programs with nested expressions. The reduction may be a string reduction like \(a \star b\) in which case a string is replaced by its value or a graph reduction in which case pointers are manipulated. This type of architecture is exemplified by the University of North Carolina’s FFP computer [41]. There are however attempts to create hybrid machines for the dataflow and reduction paradigms. Rediflow [57] has the features of both Dataflow and Reduction machines. Here, processors will work first on the instructions demanded of them if the operands are available before working on instructions that are ready for execution.

2.7 Summary Of Parallel Architectures

So far, we have presented different models of parallel architectures. Differing processor organisations have been suggested and some implementations for both the shared memory and message passing architectures described. The different models offer varying abilities
in terms of granularity of computation, performance ranges and programming requirements. Some models are targeted at specific applications and may perform poorly in other circumstances while some are general purpose machines. The MIMD class are general purpose since they consist of a number of central processing units asynchronously executing independent instruction streams. MIMD computers are grouped according to the manner in which the CPUs access memory. The multiprocessors have a single shared address space and the distance from a CPU is constant but in some cases each memory cell is closer to one CPU than to others. Multicomputers have no shared memory. Each CPU has its private address space and the processors communicate by message passing. Clearly, one cannot conclude which specific architectural structure is superior, but a cost effective parallel processing architecture is one that provides a balanced performance and an effective processor utilisation, memories and input/output with minimum communication overhead. The loosely coupled processors communicate by exchanging messages, whereas the tightly coupled processors communicate through a shared main memory. Each processor in a distributed system has its own local memory and if a processor needs data from another processor, it must send a message through a communication subsystem to the other processor about its demand. In a shared memory machine all processors have access to the global shared memory which can take the form of memory modules connected to the system bus or distributed in the form of local memories through processors that can access non local memories through an interconnection network of switches. The flexibility to access shared memory causes memory access conflicts, but the advantage lies in the fact that asynchronous communication is easy and fast. Distributed systems are preferred when the interactions between tasks are minimal as against the shared memory system that can tolerate a higher degree of interaction between tasks. Table 2.2 shows some typical examples of architectural models.
Table 2.2: Examples Of Some Architectures

<table>
<thead>
<tr>
<th>Architecture Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>IBM 7090 [52]</td>
</tr>
<tr>
<td>SIMD</td>
<td>Illiac IV [25]</td>
</tr>
<tr>
<td>MISD</td>
<td>Encore Multimax [35]</td>
</tr>
<tr>
<td>Multiprocessors (Shared Memory)</td>
<td>Transputers [29]</td>
</tr>
<tr>
<td>Multicomputers (Distributed Memory)</td>
<td>Carnegie Mellon’s Warp [36-38]</td>
</tr>
<tr>
<td>Systolic Arrays</td>
<td>MIT Tagged Token Dataflow [39]</td>
</tr>
<tr>
<td>Reduction</td>
<td>North Carolina’s FFP [41]</td>
</tr>
</tbody>
</table>

2.8 Granularity

Granularity refers to the size of tasks given to each processor, and is a very important issue in parallel performance because the time invested in creating processes and moving information among processors must be balanced by that invested in actual computation. For example a message passing program would not perform well if the time spent in communication is not balanced by that spent in evaluating computations. It is difficult to say precisely what the correct size of task should be. However for every hardware environment and coordination language (like Linda, threads or Parallel-C) there is a limit in which an application will be too fine-grained to give a meaningful performance. The cost of communication is usually a dominating factor. In Distributed systems, it takes quite some time to send and receive data between processors whereas in a shared memory architecture the data are only copied from one location in memory to another or involves movement of pointers.

If the time it takes to perform the task is less than the total time it takes to find the task, perform the computation and then return the result, more is being paid in overhead than in performing the task and good performance cannot be guaranteed. It is good practice to avoid excessive fine-grain granularity as this can lead to work starvation. At the same time too large a spread of granularity among processes is not recommended as
this will lead to load balancing problems where the computation time depends on the load of the most burdened processor.

2.9 Parallel Programming

Programming languages allow parallelism available in an architecture to be exploited. Concurrent languages can be divided into three major groupings:

- Procedure-Oriented Languages
- Message-Oriented Languages
- Operation-Oriented languages

Any of the above languages can be implemented on MIMD machines but if the language features do not match the architecture, an efficient development of a parallel algorithm will be difficult.

In Procedure-Oriented Languages [42] process interaction is based on shared variables. Processes have access to the data that they want to manipulate while providing means for ensuring mutual exclusion of processes in critical regions. These languages are particularly suitable for programming the multiprocessors. Examples include Modula [53], Concurrent Pascal [43], Mesa [44], Edison [49], Linda [46], Threads and Microthreads [54].

Message-Oriented languages are based on the principle of send and receive. They do not give access to every data object as each process manages its own data. Processes communicate by exchanging messages and so concurrent access is not a problem in this group of languages. Examples are Occam [45], Communicating Sequential Processes (CSP) [47, 48], and PLITS [56].

Operation-Oriented languages use remote procedure calls as the primary means of process interaction. These languages have the characteristics of both procedure- and message-oriented languages. Operations are performed on objects by calling procedures
while objects are managed by message passing. The languages can be implemented efficiently on both multiprocessors and multicomputers. Examples include Distributed Processes [50, 55] and Ada [51].

Parallel programming can be approached in two main ways, either by writing the conventional serial algorithm and allowing a parallel compiler to detect areas of parallelism or by using any of the parallel programming languages outlined above to exploit the hardware architecture through the syntax of the language. It is this second option that we adopt in this research. The Procedure- and Message-Oriented languages are implemented on MIMD architectures using threads, microthreads on shared memory and Parallel-C on a distributed memory machine.

2.10 Multiprocessor Implementation

The Encore Parallel Threads (EPT) system designed for the Encore Multimax provides an efficient support for concurrency on the shared memory architectures, and is used in this work. Communication is via shared variables. Setting up a thread environment has some overhead but initialisation only takes place at startup and does not affect the performance of the thread programs. Subsequently, EPT also supports Microthreads which is intended for applications such as parallelised for and do loops for example in the C programming language. The shared memory contains sections of data that can be accessed or modified by different processors. If a processor \( p_1 \) has access to a shared memory and is about to modify it, and another processor \( p_2 \) attempts to access and modify the same section, an error in computation may occur because the value can change before \( p_1 \) has completed its operation. To avoid such a conflict, controlled access and mutual exclusion with respect to such sections of memory is required. Modifiable sections of a program, shared by many processors and executed as uninterrupted operations are termed critical sections. In the EPT there are mechanisms for mutual exclusion of critical sections and these are
Monitors and Semaphores [58].

2.10.1 Monitors

The Monitor is a standard synchronisation mechanism in EPT and it keeps track of the state of a process in order to safely exit in the case of an exception. It collects critical sections into a single unit which can only permit one process to gain entry at a time. These critical sections are procedures or functions of the monitor. Monitors also execute an initialisation operation when a data structure is created. A process can access the shared data by calling one of the monitor procedures. If there is more than one process in the access queue, it has to wait until the one in the monitor has finished its operation before it can enter the monitor and resume. Sometimes some additional logical condition may have to be fulfilled before a process can enter and execute a critical section even when it is free.

2.10.2 Semaphores

Semaphores are another synchronisation mechanism which employ nonnegative integers with two associated operations $p$ and $v$. They are intended for operations where speed is of paramount importance.

- $p$ operation causes a semaphore's value to be decreased by 1 but it is not reduced beyond 0.

- $v$ operation causes a semaphore's value to be increased by 1 provided it is not 1 already.

The semaphore is normally a location in shared memory and has a value 0 if a process is executing in the critical region associated with it, otherwise its value is 1 implying that the critical region is free. A process can only gain access to the critical region if the semaphore value is 1, it immediately performs the $p$ operation to lock the region by
reducing the semaphore value to 0 and thus preventing other processes from interrupting. At completion, the process performs the \( v \) operation, raising the value from 0 to 1 thus setting the critical section free for other processes to gain access. To implement mutual exclusion every critical section in a program must be preceded by a \( p \) operation followed by a \( v \) operation on the same semaphore. PROGRAM1 below illustrates the use of threads to implement a matrix product. PROGRAM2 is an example where semaphores are implemented. The original code is in the C programming language.

```c
/*
** A parallel program using multiple threads for multiplying matrices.
*/

#include <thread.h>
#include <stdio.h>

int A[9] = {1, 2, 3,
            4, 5, 6,
            7, 8, 9};

int B[9] = {9, 8, 7,
            6, 5, 4,
            3, 2, 1};

int C[9];

main(argc, argv)
  int argc;
  char *argv[];
{
  extern void startup();
  atol(argv[1]) = procs;
  if(argc != 2)
  {
    fprintf(stderr, "usage: tst #processors\n");
    exit(1);
  }
  THREADgo(atol(argv[1]), 2*1024*1024, startup, 0, 0, 20*1024, 2);
}

void startup()
{
  extern void mult();
  /* A structure must be used to pass multiple parameters because of the way EPT handles parameters */
```
struct {
    int i;
    int j;
} ij;
for (ij.i = 0; ij.i < 3; ij.i++)
    for (ij.j = 0; ij.j < 3; ij.j++)
        THREADcreate(mult, &ij, sizeof(ij), ATTACHED, 20*1024, 2);
while (THREADjoin());
printf("%3d %3d %3d
%3d %3d %3d
%3d %3d %3d
",
    C[0], C[1], C[2], C[3], C[4], C[5], C[6], C[7], C[8]);
}

void mult(ij)
struct {
    int i;
    int j; } *ij; {
    register int i;
    register int t = 0;
    register int col = 3 * ij->i; /* row i, col 0 */
    register int row = 3 * ij->j; /* row 0, col j */

    for (i = 0; i < 3; i++) {
        t += A[col] * B[row];
        col++;
        row += 3;
    }

    C[3*ij->i + ij->j] = t;
}

To implement this parallel version on the shared memory (SM), the sequential program was converted into the parallel version by using the facilities provided in the Encore Parallel Thread (EPT) package. The Encore Parallel Thread package provides for process creation and synchronisation mechanisms in the list of the facilities in its library. A structure was used to pass multiple parameters because of the way EPT handles parameters.

The function Threadgo() establishes a single thread to initialise EPT. It is stated as: Threadgo(argv[1], data_size, startup, args, 0, stacksize, priority).
The first argument specifies the number of processors allocated for use by EPT. `data_size` is the amount of memory allocated to hold the stack and control blocks for all the threads and also to hold the shared heap. The single initial thread starts execution by entering the routine `startup` which is passed a single argument represented by `args`. The value 0 represents the `argsize` and `arg` is passed to `startup`. The `argsize` can also take a value which is nonzero in which case `args` will be treated as a pointer to a region of memory of length `argsize`. The initial thread is also given a stack size represented by the argument `stacksize` which executes at a priority ranging from 0 to 31 with 0 as the highest. On return from `Threadgo()`, EPT is shut down and other processes are released.

What follows thereafter is the creation of a corresponding number of new threads for parallel execution using the routine `Threadcreate()` provided in EPT. `Threadcreate(mult, &ij, sizeof(ij), ATTACHED, stacksize, priority)` creates a thread of control with stated priority and executes the function `mult()` which calls each of the subproblems concurrently to compute the matrix product by using the serial algorithm. The additional argument `ATTACHED` dictates that the parent cannot terminate until all the children terminate. The parent can wait for the children by executing the `Threadjoin()` operation. In the alternative, if the argument is `DETACHED` there is no relationship between the parent and the children and each is entirely independent. In our programs, `ATTACHED` is used to ensure that all new threads have completed their respective computation before termination.

In the convex hull program, the startup function generates a menu option for computing the convex hull sequentially or in parallel. The first option executes the serial algorithm. In the latter option, the parallel execution is initiated. What follows thereafter is the creation of a corresponding number of new threads for parallel execution using the routine `Threadcreate()`.

/*
** A parallel program that counts the number of loops performed by each thread using semaphores in the critical region. */

PROGRAM2
========
#include <thread.h>
#include <stdio.h>
SEMAPHORE sem;
int count;
main(argc, argv)
int argc;
char *argv[] ;
{
    extern void startup();
    if(argc != 2)
    {
        fprintf(stderr,"usage: tst #processors\n");
        exit(1);
    }
    THREADgo(atol(argv[1]), 2*1024*1024, startup, 0, 0, 20*1024, 2);
}
void startup()
{
    extern int child();
    THREAD tcb;
    int i, total_iterations = 0;
    sem = THREADseminit(1);
    for(i=0; i<10; i++)
        THREADcreate(child, 0, 0, ATTACHED, 20*1024, 2);
    while((tcb = THREADjoin()) != NULL) {
        /* wait for the children to terminate */
        total_iterations += THREADreturnvalue(tcb);
        THREADfree(tcb);
    }
    printf("count = %d, total iterations = %d\n", count, total_iterations);
    fflush(stdout);
}

int child()
{
    int i;
    for(i=0; ; i++) {
        THREADpsem(sem); /* critical region */
        if(count >= 1000)
The critical sections are protected by the statements \texttt{THREADpsem(sem)} and \texttt{THREADvsem(sem)} and cannot be interfered with by other processes until after its operation is completed by the current processor when it is free. The threads and microthreads libraries are used to introduce parallelism into the hull programs. Multiple threads of control run in a single shared address space, the overhead of process creation is incurred only in the start-up phase of the algorithm. Threads in this context are lightweight containing only program counters and a small amount of additional memory.

\section*{2.11 Distributed Memory Implementation}

Message passing is a method of synchronisation between processors in distributed memory machines. The process transmitting the information is the \texttt{sender}, and the process receiving it is the \texttt{receiver}. The channels for communications are clearly defined and specified for exchange of information between the processes.

We intend to use a Meiko Computing Surface and the illustration here uses the concept and implementation on the transputer. The examples below show how the different functions are being harnessed to provide the communication between two processes via Transports. A \texttt{par} file which describe a multi-process task to the parallel loader is also shown. Each process calls the function, \texttt{csn\_init()} to initialise the Computing Surface Network (CSN). This is followed by a call to \texttt{csn\_open()} that creates a connection between the process and the CSN. This connection is called a \texttt{Transport}. Each Transport on the CSN has an associated address, called a \texttt{Net Id}.
For a message to be passed from a Sender’s Transport to the Receiver’s Transport it is necessary for the Sender to determine the Net Id of the receiver’s transport. To do this the receiving process calls the CSN function, `csn_registername()`, which instructs the CSN to associate the function’s argument with the transport’s Net Id. The sending process then makes a similar call to the function, `csn_lookupname()`, which instructs the CSN to return the Net Id of the named transport. Finally, having established the Net Id of the receiver’s transport, the sender passes its data by calling the CSN function, `csn_tx()` and blocks. This function passes data to the transport whose Net Id is specified as an argument. In our example, the first process, `heading.c` writes the title for the table and informs the second process, `solution.c` that it has finished. The second process then waits until it receives the signal from the first process before computing and writing the temperature conversion.

Process One  (Writes Title) heading.c
========================================
```
#include <stdio.h>
#include <csn/csn.h>
#include <csn/names.h>
#include <cs.h>
main( argc, argv )
int argc;
char* argv[];
{
    Transport transport;
    netid_t solution_id;
    int flag = 1;
    int status;
    csn_init();
    status = csn_open( CSN_NULL_ID, &transport );
    if( status != CSN_OK )
        cs_abort("heading: cannot open transport\n", -1 );
    status = csn_lookupname( &solution_id, "SolutionTransport", 1 );
    if( status != CSN_OK )
        cs_abort("heading: cannot lookup SolutionTransport\n", -1 );
    printf("Fahrenheit Celsius\n");
    csn_tx( transport, 0, solution_id, &flag, sizeof(flag) );
}
```
Process Two  (Perform Computation) solution.c
================================================
#include <stdio.h>
#include <csn/csn.h>
#include <csn/names.h>
#include <cs.h>

#define LOWER 0 /* lower limit of table */
#define UPPER 300 /* upper limit */
#define STEP 20 /* step size */
main( argc, argv )
int argc;
char* argv[];
{
    Transport transport;
    int flag;
    int status;
    int fahr;
    csn_init();
    status = csn_open( CSN_NULL_ID, &transport );
    if( status != CSN_OK )
        cs_abort( "solution: cannot open transport\n", -1 );
    status = csn_registername( transport, "SolutionTransport" );
    if( status != CSN_OK )
        cs_abort( "solution: cannot register SolutionTransport\n", -1 );
    csn_rx( transport, NULL, &flag, sizeof(flag) );
    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("\%7d \%12.1f\n",fahr,(5.0/9.0)*(fahr-32));
}

Parallel Loader (heading.par)
=============================
par
    processor 0 heading
    processor 1 solution
endpar

The Parallel Loader specify the placement of the two processes 'heading' and 'solution' within the network. More examples are available in [97]. The CSTools also provides the CSBuild library to create a customised loader to place code more effectively for execution. To implement the CSBuild routine here we need to create two executable files 'heading'
and 'solution'. In a CSBuild program, objects called Groups are arranged into a complex Group hierarchy. In the CSBuild program, cs_group() is used to create a single group and we assign the single process, heading, to that Group by using cs_exe(). Similarly, we adopt the same method in creating the executable file solution. At this moment, each of the processes as they stand are not committed to processors for execution. The function cs_option() is used to set one of the groups attributes and in our own case we specify that the group will be executed on a transputer. Finally, cs_load() sets the stage by putting our processes onto the hardware and control will not be returned to the program until the task is completed. A timing routine to trigger the system's clock was also written.

Example of CSBuild Program
=================================
#include <stdio.h>
#include <cstools/build.h>
main() {
    GROUP* headingGRP_ptr;
    GROUP* solutionGRP_ptr;

    headingGRP_ptr = cs_group(NULL, "HeadingGRP");
    solutionGRP_ptr = cs_group(NULL, "SolutionGRP");

    cs_exe(headingGRP_ptr, "Heading", "heading", 0);
    cs_exe(solutionGRP_ptr, "Solution", "solution", 0);

    cs_option(headingGRP_ptr, "commit", "transputer");
    cs_option(solutionGRP_ptr, "commit", "transputer");

    cs_load();
}

In all our implementations, we have used the synchronous and blocking communication type. By using this mechanism, data transmission will only occur when both the sender and receiver are ready and both processes will block (wait) until transmission is complete.
A sender process will block until its message is received, and a receiver will block until
the message is sent by the sender. By using this model, both processes must synchronise
before data may be transferred between them with the result that one of the processes
may waste time waiting for the other to be ready. There are other options available to
the programmer to overcome this problem. The send mode options include Blocked
Synchronous, Blocked Asynchronous, Non-Blocked Synchronous and Non-Blocked Asyn-
chronous. The receiver mode options are Blocked and Non-Blocked. The use of blocking
and non-blocking communications affects the way in which the transmission function de-
termines that the communication is completed. A message transmitted synchronously
is complete only when the receiving process has received the message into its own local
buffer whereas a message transmitted asynchronously is complete only when data has
been received by the CSN.

2.12 Performance Measures

Once a parallel program has been implemented, it is the responsibility of the programmer
to explore the performance of the algorithm. If the parallel program does not run faster
than the sequential code, at least to a reasonable limit, then it is a failure. The usual
measures of parallel performance are cost, attractiveness, speedup and efficiency. The
two we will use to measure the performance of $n$-dimensional convex hull algorithms,
are speedup and efficiency. We will compare the parallel version with an equivalent
sequential version of the same algorithm. Equally worth mentioning is the fact that
we will run the sequential algorithm on one processor of a parallel machine, and the
parallel versions of the same algorithm on many processors of the same machine. This
is important because it is possible to split code over several types of processor which
have different performance characteristics and obscure the results. We will then use the
performance characteristics mentioned above to study the performance figures obtained
from our algorithms and try to understand them.

2.12.1 Speedup

The speedup achieved by a parallel algorithm running on \( p \) processors is the ratio between the time taken by a parallel computer executing the serial algorithm and the time taken by the same parallel computer executing the parallel algorithm using the \( p \) processors. This can be expressed as

\[
S_p = \frac{T_s}{T_p}
\]

where

\( S_p = \text{Speedup} \)

\( T_s = \text{running time of fastest sequential algorithm and} \)

\( T_p = \text{running time of parallel algorithm.} \)

Normally \( 0 \leq S_p \leq p \). Ideally, the maximum value of \( S_p \) using \( p \) processors is \( p \) but in practice this is seldom achieved for the following reasons:

- It is extremely difficult to partition a problem into \( p \) tasks, each requiring a processor to use the same amount of time to solve each task. There may be some idle time on processors.

- Process creation and synchronisation in a partitioned algorithm adds overheads.

- Sequential code limits the speedup. If any portion of the algorithm must be executed sequentially, then the remaining processors have to wait for the sequential portion to complete its computation before they resume.

- The architecture used also imposes restrictions that render the desired running time unattainable. This could be caused by memory conflicts and/or communication path delays.
2.12.2 Efficiency

The efficiency $e$ of a parallel algorithm running on $p$ processors is the speedup divided by $p$ and usually $0 \leq e \leq 1$

$$e = \frac{S_p}{p}$$

Algorithms that approach the upper bounds in $S_p$ and $e$ as $p$ tends to $\infty$ or the problem size increases for fixed $p$ are said to be optimal.

2.12.3 Algorithm Equivalence

For $S_p = p$, we always assume that parallel and serial programs are the same algorithmically but in practice this is rarely achieved because code changes are introduced when we write parallel components. Good serial algorithms are optimised for sequential machines. An algorithm with optimal speedup may have a very poor efficiency and on the other hand an algorithm with good efficiency can have a poor speedup. A good serial algorithm may be a bad parallel algorithm whereas a cheap and nasty serial algorithm may turn out to be the best parallel algorithm.

The nature of the problem can also affect the achievable speedup. Some problems are compute-bound. In such a case the amount of computation dominates and the processors will be busy most of the time. An example is matrix multiplication. If we consider an $n \times n$ matrix, the total data is $O(n^2)$ but $O(n^3)$ operations are required. Others are input/output bound with very little computation but input/output phases dominate the process e.g matrix addition with $O(n^2)$ data and $O(n^2)$ operations. Often this fact is obscured by the fact that the lack of dependency in matrix addition make it much easier to parallelize than matrix multiplication.
Chapter 3

The Convex Hull Problem

Many algorithmic problems in Computational Geometry involve geometric concepts such as points, lines and distances. Also, many of the problems are deceptively easy to solve using the human visual system, but often present a real challenge when designing an algorithm. The convex hull problem is one such problem. In order to discuss the convex hull problem formally and in a more generalised manner, it is appropriate here to review the basic concepts and terminologies that are relevant. The combinatorial theory of convex hulls is largely concerned with their facial structure [93]. This section will provide formal definitions of the geometric concepts and notations used in this thesis. The objects we will normally manipulate are sets of points in Euclidean space. Each point is represented as a vector of appropriate dimension. The geometric objects will normally consist of a finite set of points. We shall consider besides individual points, the straight line containing two given points, the line segment defined by its two given points, the polygon defined by a number of points, etc.

3.1 Definition Of Terms

Let $R$ be the set of real numbers. By $R^d$ we mean the d-dimensional Euclidean space, that is the space of d-tuples $(x_1, \cdots, x_d)$ of real numbers $x_i$, $i = 1, \cdots, d$ with metric $(\sum_{i=1}^{d} x_i^2)^{1/2}$. Some important definitions are given below:
Point: A d-tuple \((x_1, \cdots, x_d)\) denotes a point \(p\) of \(R^d\) which is also a d-component vector applied to the origin of \(R^d\).

Line: Given two distinct points \(p_1\) and \(p_2\) in \(R^d\), the linear combination 
\[
\beta p_1 + (1 - \beta)p_2 \ (\beta \in R)
\]
is a line in \(R^d\).

Line Segment: Given two points \(p_1\) and \(p_2\) in \(R^d\) the line segment denoted by \(p_1p_2\) is defined by 
\[
\beta p_1 + (1 - \beta)p_2 \text{ provided } 0 \leq \beta \leq 1.
\]

Flat: An \(r\)-flat is a region determined by \((r + 1)\) points having dimension \(r\). We will call \(r\)-flat \((r > d)\) a hyperplane of \(r\) dimensions, denoted by \(H^{Tr}\).

Linearly Independent: The collection of points \(p_1, p_2, \cdots, p_k\) in \(R^d\) is said to be linearly dependent if there exist numbers \(\alpha_1, \alpha_2, \ldots, \alpha_k\), not all zero such that 
\[
\alpha_1 p_1 + \alpha_2 p_2 + \cdots + \alpha_k p_k = 0.
\]
If the vectors are not linearly dependent, they are said to be linearly independent (i.e. if no vectors in the collection can be expressed as a linear combination of the other vectors).

Affine set: Given \(k\) distincts points \(p_1, \ldots, p_k\) in \(R^d\), the set of points 
\[
p = (\alpha_1 p_1 + \alpha_2 p_2 + \cdots + \alpha_k p_k) \text{ where } (\alpha_j \in R, \sum_{j=1}^{k} \alpha_j = 1)
\]
is the affine set generated by \(p_1, p_2, \cdots, p_k\) and its affine combination is \(p\). If \(k = 2\), the resulting affine set is a straight line through two points. Examples of affine sets are points, lines, planes, hyperplanes.

Affinely Independent: Given \(k\) points \(p_1, p_2, \cdots, p_k\) in \(R^d\), the points are said to be affinely independent if the \((k - 1)\) vectors \((p_2 - p_1), \cdots, (p_k - p_1)\) are linearly independent. A useful criterion for affine independence is the following:
If \(x_i = (a_{i1}, \cdots, a_{id})\) then \(\{x_1, \cdots, x_k\}\) is an affinely independent set of points if
and only if the matrix $A$ has rank $k$.

$$A = \begin{pmatrix} 1 & a_{11} & \cdots & a_{1d} \\ 1 & a_{21} & \cdots & a_{2d} \\ \vdots & \vdots & & \vdots \\ 1 & a_{k1} & \cdots & a_{kd} \end{pmatrix}$$

**Affine Hull:** Given a subset $K$ of $\mathbb{R}^d$, the affine hull $AH(K)$ of $K$ is the smallest affine set containing $K$. For any two points $p_1, p_2$ in $K$, the entire line determined by these two points belong to $AH(K)$. The affine hull of a segment is a line, and of a plane polygon is a plane.

**Convex set:** Given $k$ distincts points $p_1, \ldots, p_k$ in $\mathbb{R}^d$, the set of points $p = (\alpha_1 p_1 + \alpha_2 p_2 + \cdots + \alpha_k p_k)$ where $(\alpha_j \in \mathbb{R}, \alpha_j > 0, \sum_{j=1}^{k} \alpha_j = 1)$ is the convex set generated by $p_1, p_2, \ldots, p_k$ and its convex combination is $p$. A domain $D$ in $\mathbb{R}^d$ is convex if, for any two points $p_1$ and $p_2$ in $D$, the segment $\overline{p_1p_2}$ is entirely contained in $D$.

**Hyperplane:** A hyperplane $H$ is the set of points $X = (x_1, x_2, \ldots, x_d)$ which satisfy an equation represented in the form $\sum_{i=1}^{d} \alpha_i x_i - \beta = 0$, where not all $\alpha_i$ are zero. A hyperplane $H$ separates the space $\mathbb{R}^d$ into two half spaces. A normal to the hyperplane $H$ is a vector parallel to $r$, where $r = (\alpha_1, \alpha_2, \cdots, \alpha_d)$. The unit normal to $H$ denoted by $\hat{r}$ is given by

$$\hat{r} = \frac{1}{(\sum_{i=1}^{d} \alpha_i^2)^{1/2}}(\alpha_1, \alpha_2, \cdots, \alpha_d)$$

A hyperplane $H$ bounds the set $S \subset \mathbb{R}^d$ if and only if all points of $S$ lie either on $H$ or in one half space. If $\hat{v}_i$ denotes a unit vector along $QP, Q \in H, P \in S$, we say $H$ bounds the set $S$ if and only if either the inner product $(\hat{d} \cdot \hat{v}_i) \geq 0$ for $i = 1, \cdots, d$ or the inner product $(\hat{d} \cdot \hat{v}_i) \leq 0$ for $i = 1, \cdots, m$; $\hat{d}$ being the unit normal to $H$ and $m$ the number of points.
Support Hyperplane: A hyperplane $H$ is a support plane of $S$ if $H$ bounds $S$ and at least one point of $S$ lies on $H$.

Convex Hull: The convex hull of a set $S \subseteq \mathbb{R}^d$ is the intersection of all convex sets containing $S$. We denote the convex hull of $S$ as $CH(S)$. If $S$ is a finite set, then $CH(S)$ is called a polytope. In general, the convex polytope of a set $S$ is the set of all convex combinations of finite subsets of $S$ i.e.

$$CH(S) = \{ x \in \mathbb{R}^d \mid x = \lambda_1 x_1 + \cdots + \lambda_r x_r, 1 \leq r < \infty, \lambda_i \geq 0; x_i \in S, \sum_{i=1}^{r} \lambda_i = 1 \}$$

A support plane $H$ of $S$ is said to be an $d$-face of $CH(S)$ if $d$ independent points of $S$ lie on $H$. A convex polytope is described by means of its boundary, which consists of faces. Each face of a convex polytope is a convex set (i.e. a lower dimensional convex polytope); a k-face denotes a k-dimensional face (i.e. a face whose affine hull has dimension $k$). If a polytope $P$ is $d$-dimensional, its $(d-1)$-faces are called facets, its $(d-2)$-faces are subfacets, its 1-faces are edges, and its 0-faces are vertices. For a 3-D polytope, facets are plain polygons, while subfacets and edges coincide.

Edge: A $d$-edge of $CH(S)$ is a $(d - 2)$-flat contained in a support plane of $CH(S)$ which is not a $d$-face of $CH(S)$.

Size Of Set: The size $n$ of a set $S$, denoted by $| S |$ is the number of points in $S$.

Norm: The vector norm of $x$ is a non negative number denoted by $\| x \|$, associated with $x$, satisfying:

(a) $\| x \| > 0$, $\| x \| = 0$ implies $x = 0$.

(b) $\| k x \| = | k | \| x \|$ for any scalar $k$.

(c) $\| x + y \| \leq \| x \| + \| y \|$ (the triangular inequality).
The length or norm of a $d \times 1$ column vector $\| x \|$ is defined to be

$$\| x \| = \left( \sum_{i=1}^{d} |x_i|^2 \right)^{1/2}$$

Orthonormal Set: If $(x, x) = \| x \|^2 = 1$, the vector $x$ is said to be normalised. If a set of vectors $x_1, \ldots, x_d$ is orthogonal and normalised i.e $(x_i, x_j) = 0$ ($i \neq j$) = 1 ($i = j$), then the vectors are said to form an orthonormal set.

### 3.2 2-D Algorithms

Among the problems in computational geometry, the planar convex hull problem is one of the earliest and best studied. Numerous papers have appeared in the literature dealing with different aspects and generalisations of the planar convex hull problem. Given a set $S$ of $n$ points in $R^2$, this section reviews some of the earlier approaches in the design and analysis of algorithms for constructing the convex hull $CH(S)$ from $S$ using sequential computations. Yao [87] has shown that this problem has an $O(n\log_2 n)$ sequential lower bound. There is a long list of articles containing results on the convex hull of a planar point set in two dimensions. Some examples are [60], [62], [64], [76], [81] in which this lower bound is achievable. The running times of these algorithms are either $O(n\log_2 n)$ where $n = | S |$, since the problem is as hard as sorting, or $O(nH)$ where $H$ is the number of points on the convex hull. Kirkpatrick [75] has proposed an algorithm whose complexity is $O(n\log_2 h)$ where $h$ is the number of edges of $CH(S)$ and is superior to the previous ones in the sense that its running time is sensitive to the size of the output. In the worst case, when $h = n$, the result reduces to $O(n\log_2 n)$. The approach adopted in the algorithm is to find the maximum and minimum coordinates of $S$, determine the upper and lower convex polygonal paths respectively, and then concatenate the two paths to obtain the convex hull of $S$. For brevity here, we will review three early approaches to the solution of this problem namely, Graham [64], Jarvis [60] and the Divide-and-Conquer technique.
which cover most of the variations.

3.2.1 Graham’s Algorithm:

Graham in [64] presented one of the first \( O(n \log n) \) algorithms to compute the convex hull of \( n \) points in the plane. The first step in the algorithm involves sorting the input points and this step dominates others in the determination of the convex hull. Since sorting is of \( O(n \log n) \), it follows that finding the convex hull by Graham’s algorithm requires \( O(n \log n) \) steps. The algorithm can be summarised as follows:

**Step 1:** An internal point \( O \) is chosen arbitrarily (e.g. centroid of three non-collinear points). At worst case this can be done in \( c_1 n \) steps, where \( c_1 \) is a constant.

**Step 2:** The points are expressed in polar coordinates about the origin \( O \) and \( \theta = 0 \) in the direction of an arbitrary fixed line \( L \) from \( O \). This can be done in \( c_2 n \) operations, \( c_2 \) a constant.

**Step 3:** The elements \( r \exp(i \theta) \) are sorted in terms of increasing \( \theta \) such that the set of points \( S = \{ r_1 \exp(i \psi_1), \ldots, r_n \exp(i \psi_n) \} \) with \( 0 \leq \psi_1 \leq \ldots \leq \psi_n \leq 2\pi \) and \( r_i \geq 0 \). This is possible in \( O(n \log n) \) time.

**Step 4:** If \( \psi_i = \psi_{i+1} \) then we delete the points with smaller amplitude since it cannot be an extreme point. Also points with \( r_i = 0 \) can be deleted and renumbering the rest of the points so that the set of points \( S' = \{ r_1 \exp(i \psi_1), \ldots, r_{n'} \exp(i \psi_{n'}) \} \) where \( n' \leq n \). This elimination can be done in less than \( n \) comparisons.

**Step 5:** Start with three consecutive points in \( S' \), say, \( A = r_k \exp(i \psi_k), B = r_{k+1} \exp(i \psi_{k+1}), C = r_{k+2} \exp(i \psi_{k+2}) \) with \( \psi_k < \psi_{k+1} < \psi_{k+2} \). There are two possibilities as illustrated using figure 3.1.
Figure 3.1: Angle Between Three Points

1. If $(\alpha + \beta) \geq \pi$, delete the point $r_{k+1}e^{i\psi_{k+1}}$ from $S'$ since it cannot be an extreme point and return to the beginning of step 5 with the points $r_k e^{i\psi_k}$, $r_{k+1} e^{i\psi_{k+1}}$, $r_{k+2} e^{i\psi_{k+2}}$ replaced by $r_{k-1} e^{i\psi_{k-1}}$, $r_k e^{i\psi_k}$, $r_{k+2} e^{i\psi_{k+2}}$.

2. If $(\alpha + \beta) < \pi$, return to beginning of step 5 with points $r_k e^{i\psi_k}$, $r_{k+1} e^{i\psi_{k+1}}$, $r_{k+2} e^{i\psi_{k+2}}$ replaced by $r_{k+1} e^{i\psi_{k+1}}$, $r_{k+2} e^{i\psi_{k+2}}$, $r_{k+3} e^{i\psi_{k+3}}$. This step can be accomplished in less than $2n'$ since the number of possible points in $CH(S)$ is reduced by one or the current total number of points in $S'$ is increased by one.

The algorithm starts by constructing a simple closed polygon from the sorted points in angular order about the point $O$, so that tracing through the points gives a closed polygon. Computation of the convex hull is completed by proceeding cyclically around the points, trying to place each point on the hull and eliminating the points that cannot possibly be on the hull. Illustrating with the example in the Table 3.1, we consider the points in the order $v_1, v_{12}, v_9, v_{11}, v_{13}, v_{15}, v_{10}, v_5, v_8, v_4, v_2, v_{14}, v_0, v_7, v_6, v_3$. We know that because of sorting the points $v_1, v_{12}$ are on the hull. When $v_9$ is encountered, the algorithm
Table 3.1: Set Of Points To Illustrate 2-D Algorithms

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v6</th>
<th>v7</th>
<th>v8</th>
<th>v9</th>
<th>v10</th>
<th>v11</th>
<th>v12</th>
<th>v13</th>
<th>v14</th>
<th>v15</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>12</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>15</td>
<td>11</td>
<td>17</td>
<td>16</td>
<td>14</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>y</td>
<td>10</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>16</td>
<td>12</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>14</td>
<td>15</td>
<td>3</td>
<td>17</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 3.2: 2-D Convex Hull Illustration

includes it in the trial hull for the first three points. When \( v_{11} \) is encountered, we note from the algorithm that \( v_9 \) cannot be on the hull because the points \( v_{11}v_9v_{12} \) forms an internal angle \( v_{12}v_9v_{11} \) which is \( \geq \pi \) and so \( v_9 \) is not a candidate for the hull and hence is eliminated from further consideration. The situation as each new point is encountered will either add to or eliminate the point from the partial hull so far constructed until all the points are considered. Figure 3.2 shows a 2-D convex hull.

Step 3 of the algorithm requires sorting the points and from the sorting algorithms, we know that the expected time performance is \( O(n \log_2 n) \) [90]. Since Graham's algorithm requires sorting, and it is this step that dominates, its expected time performance is therefore \( O(n \log_2 n) \).
3.2.2 Jarvis's Algorithm:

Jarvis [60] presented an alternative solution to the convex hull problem that runs in time $O(nH)$ where $H$ is the number of vertices of the hull. The approach taken by Jarvis is suggestive of the idea of “gift wrapping”. Starting with a point $v_1$ as shown in Figure 3.2, that is known to be on the convex hull, in linear time $O(n)$, we find the next point $v_{12}$ such that edge $v_1v_{12}$ is on the convex hull, i.e all the remaining points must lie on one side of the directed line containing $v_1v_{12}$. After $v_{12}$ has been found, the same technique is applied to locate the next point $v_{11}$ such that $v_{12}v_{11}$ is a hull edge, and so on, until we ‘wrap’ back to the starting point $v_1$. The algorithm can be summarised as follows:

Step 1: Find an origin point $v_i$, $0 \leq i \leq n - 1$ from the set with largest $x$-coordinate (and smallest $y$-coordinate, if several points have the same minimal $x$ value);

Step 2: Let $L_1$ be the line containing $v_i$ which is parallel to the $x$-axis. Take a horizontal ray in the positive direction and “sweep” it upward until we hit another point $v_k$ such that the angle between the line joining $v_i$ and $v_k$ and the line $L_1$ is minimised. For equal minimum angles pick the point closest to the origin.

Step 3: Shift the origin to $v_k$ and repeat step 2 with consistent angle direction and origin until the first convex hull point is re-found.

Since there are $H$ vertices we have to have at most $H - 1$ edges (faces). Finding a vertex when given an existing one requires re-examination of $(n - 1)$ points, thus $O(nH)$.

3.2.3 Divide-and-Conquer Algorithms

A problem of size $n$ can often be split into two similar subproblems of size approximately equal to $n/2$. This splitting process can be repeated on subproblems (recursively) until subproblems of constant size are obtained for which the solutions are trivially known. For example, the quicksort algorithm [90] is based on this principle. On the other hand,
one can start with about \( n \) equal-sized small problems, and marry the subsolutions in a pairwise manner as in the merge sort algorithm [94]. These techniques have many applications in computational geometry which often result in considerable savings in expected computation time. The first general discussion of their value in the design of fast expected time algorithms is illustrated by Bentley and Shamos [66]. The convex hull problem for a set of \( n \) points in the plane can also be solved in \( O(n\log_2 n) \) time by a Divide-and-Conquer Technique [82]. This technique normally involves partitioning of the original problem into several subproblems, recursively solving each problem and then combining the solutions to the subproblems to obtain the solution of the original problem. The following steps are involved.

**Step 1:** If \( |S| \leq 2 \) return \( S \); else go to step 2.

**Step 2:** Partition the original set \( S \) arbitrarily into two subsets \( S_1 \) and \( S_2 \) of approximately equal number of points.

**Step 3:** Recursively find the convex hulls of \( S_1 \) and \( S_2 \).

**Step 4:** Merge the two subconvex hulls together to form the convex hull for \( S \).

Preparata and Shamos [82] gave the following algorithm for the merge procedure:

Given two convex polygons \( S_1 \) and \( S_2 \), the merge step could be performed as follows:

**Step 1:** Find a point \( v \) that is internal to \( S_1 \) (e.g. centroid of any three vertices of \( S_1 \)). This point will also be internal to \( CH(S_1 \cup S_2) \).

**Step 2:** Determine whether or not \( v \) will be internal to \( S_2 \). If \( v \) is not internal go to step 4.

**Step 3:** If \( v \) is internal to \( S_2 \), the vertices of both \( S_1 \) and \( S_2 \) occur in sorted angular order about \( v \) as shown in figure 3.3a, we merge the vertices of both \( S_1 \) and \( S_2 \) and proceed to step 5.
Figure 3.3: 2-D Convex Hull For Divide-And-Conquer

**Step 4:** If $v$ is not internal to $S_2$, figure 3.3b applies. As seen from $v$, $S_2$ lies in a wedge whose apex is $v$ and whose apex angle is $\leq \pi$. This wedge is defined by two vertices $u$ and $t$ of $S_2$, thus partitioning $S_2$ into two chains of vertices which are monotonic in polar angle about $v$, one in increasing angle and the other decreasing. Of these two chains, the one convex towards $v$ can be immediately discarded, since its vertices will be internal to the convex hull of $S$. The other chain of $S_2$ and the boundary of $S_1$ constitute two sorted lists that contain at most $n$ vertices. They can be merged to form the vertices of the convex hull of $S$, which is sorted about $v$.

**Step 5:** Step 5 of Graham’s algorithm can now be performed on the obtained list of vertices from $S_1$ and $S_2$ to obtain the convex hull of $S$. Since in the worst case $n = |S_1| + |S_2|$ the algorithm is $O(n \log_2 n)$.

### 3.3 3-D Algorithms

The convex hull of a set of points in 3-D space is a convex three dimensional object with flat faces. The divide-and-conquer technique of constructing the convex hull in 2-D can be
easily extended to the 3-D case. However, this is more involved than the two dimensional
case. This problem has been studied by Day [63], Johansen and Gram [65], Preparata
and Hong [62] among others. Suppose we are given a set $S = \{p_1, p_2, \ldots, p_n\}$ of $n$
points in $\mathbb{R}^3$. For simplicity we assume that for any two points $p_i$ and $p_j$ in $S$ we have
$x_k(p_i) \neq x_k(p_j)$, for $k = 1, 2, 3$. The Divide-and-Conquer algorithm of [82] could be
summarised as follows:

If $|S| \leq 2$ then return $CH(S)$

else

begin

divide $S$ into $S_1$ and $S_2$ such that $|S_1| = \lfloor 1/2 |S| \rfloor$ and $S_1 \cup S_2 = S$;

$S' := \text{Convex Hull}(S_1)$;

$S'' := \text{Convex Hull}(S_2)$;

$T := \text{Merge}(S', S'')$;

return $(T)$

end.

As a preliminary step the elements of $S$ are sorted according to the coordinate $x_1$
and relabelled if necessary so that we may assume $x_1(p_i) < x_1(p_j)$ if and only if $i < j$.
We assume that the polytopes $S_1$ and $S_2$ which are two nonintersecting 3-D polytopes
have been recursively obtained. Due to initial sorting and to the chosen partition of the
resulting point set, the convex hull of $S_1$ and $S_2$ are nonintersecting. The merge, which
is the crucial component of the method, involves the calculation of the collar (i.e a union
of triangular faces, each face supporting a plane tangent to an edge of one hull and a
corner of the other). Having identified the corners, edges and faces of $CH(S)$, a complete
description of $CH(S)$ is built. The method was first proposed by Preparata and Hong
[62].

Johansen and Gram [65] gave the following algorithm for finding the convex hull of a
Table 3.2: Sequential Running Time For 2-D and 3-D Algorithms

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Author</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D</td>
<td>R. L. Graham [64]</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td></td>
<td>R. A. Jarvis [60]</td>
<td>$O(nH)$</td>
</tr>
<tr>
<td></td>
<td>Divide-and-Conquer [82]</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td>3-D</td>
<td>Preparata and Hong [62]</td>
<td>$O(n \log_2 n)$</td>
</tr>
<tr>
<td></td>
<td>Johansen and Cram [65]</td>
<td>$O(nF)$</td>
</tr>
</tbody>
</table>

3-D problem

**Step 1:** Find one face of the convex hull;

**Step 2:** Initialise Hull, Boundary, and set of potential vertices

repeat

**Step 3:** Find a new face adjoining the Boundary

**Step 4:** Update Hull, Boundary and set of potential vertices

until Boundary is empty

To find a new face, an edge $E$ is selected from the current boundary. Since $E$ is a boundary edge it has one adjoining face $F'$ already in the convex hull. With $E$ and $F'$ the remaining set of points are scanned to find a new point which together with $E$ defines a new face of the hull. The computational complexity of this algorithm is related to $n$, the number of points in $S$, and to the number of faces, $F$. The worst case computing time is $O(nF)$. In fact, this method is an extension of Jarvis' technique to the 3-D case. Table 3.2 shows the complexity of the sequential version for 2-D and 3-D problem.

### 3.4 n-D Algorithms

There are few research works available on higher dimensional spaces of the convex hull problem. This is because the $n$-D problem is more complicated than the 2-D and 3-D
cases. Secondly, most of the applications rely heavily on the lower dimensional cases. In
the Computer Science literature Chand and Kapur’s study [1] is based on the geometry
of convex polytopes. Chand and Kapur observed that exactly two faces of the convex
polytope of a set \( S \subset \mathbb{R}^d \) intersect along each edge of the hull. If one of the edges and
one of the faces containing this edge are known, then the second face can be computed by
a rotation through an appropriate angle of the known face about the known edge. The
determination of each new face gives rise to at least \((d - 1)\) edges of the hull that are
different from the known edge. This process is continued until each edge is the intersection
of two adjacent faces of the convex polytope. The convex polytope of the set \( S \) is generated
by repeating a cycle of steps, each cycle computing a new face of the desired polytope
until all the faces are determined. Chand and Kapur have given a generalised algorithm
for finding the convex hull of \( n \)-Dimensional problem in which 2-D and 3-D are special
cases.

Let \( S_c \) denote the subset of \( S \) whose convex polytope is being enumerated, and \( d_c \)
denote the current dimension of the space. Let \( m \) denote the number of points of \( S_c \). The
following steps summarise the algorithm:

**Step 1:** Let \( S_c = S, \ d_c = d \) and \( m_c = m + 1 \)

**Step 2:** Determine point(s) of \( S_c \) with least first component. Let \( S_b \) be the set consisting
of these points \( P^j \in S_c \). The hyperplane \( H, x_1 = p^j \), is a support hyperplane of \( S \)
and its normal is parallel to the vector \( \hat{d} = (1, 0, 0, \ldots, 0) \).

**Step 3:** Construct a unit vector \( \hat{e} \) such that \((\hat{e} \cdot \hat{d}) = 0, \ (\hat{e} \cdot \hat{v}_i) = 0, \ P_i \in S_b, \ \hat{v}_i \) being a
unit vector along \( p_i^j p^i \) and \((\hat{e} \cdot v_k) \geq 0, \ P_k \in S_c \).

**Step 4:** For each point, \( P^k \in S_c \), compute the ratio

\[
\frac{\lambda_k}{\mu_k} = \frac{\hat{e} \cdot \hat{v}_k}{\hat{d} \cdot \hat{v}_k}
\]

50
and determine point(s) $P^J \in S_c$ such that

$$\frac{\lambda_J}{\mu_J} = \max\{\frac{\lambda_k}{\mu_k}\}$$

where the maximum is taken over all $k$ such that $P^k \in S_c$.

The normal to the $z$-flat defined by adjoining to $S_b$ the points for which the ratio of $\lambda$ and $\mu$ is maximum, is given by, $\hat{d}^* = \lambda_J \hat{d} + \mu_J \hat{e}$, where $\lambda_J^2 + \mu_J^2 = 1$.

**Step 5:** If $z < d_c$ the starting face of $CH(S_c)$ has not been computed yet; therefore, replace $S_c$ by the points on $z$-flat and return to step 3 with $\hat{d} = \hat{d}^*$. If $z \geq d_c$ a $d_c$-face of $CH(S_c)$ has been computed. But in the case when $z > d_c$ let $S_c$ denote the points on the $z$-flat and return to step 2 with $d_c = d_c - 1$. When $z = d_c$ go to step 6.

**Step 6:** Check whether the $d_c$ edges of the computed face have been found. Finding an edge implies that one face containing this edge was found before and now that the second face has been computed this edge will be omitted from further consideration. Save new edges except for one. Return to step 3 with $S_c$ consisting of points defining this edge and with $\hat{d} = \hat{d}^*$. If all the edges of the face are already known go to step 7.

**Step 7:** Pick an edge and compute the normal $\hat{d}$ to the face containing this edge. Return to step 3 with $S_c$ consisting of points on this edge. If no edge exists in the storage then the $d_c$-polytope has been computed; proceed to step 8.

**Step 8:** Check whether $d_c = d$. If yes, the desired convex polytope has been generated.

If $d_c < d$ return to step 6 with the faces of the $d_c$-polytope being the edges of the $d_c = (d_c + 1)$-polytope.

Swart [2] has studied the facet, facial and lattice problems and presented algorithms which exhibit the best known time complexity. These algorithms are based on a reformulation.
and analysis of Chand and Kapur's algorithm using the affine basis method in order to reduce the computational effort. This method is the basis of our work and will be presented in more detail in chapter 4. The facets of the convex hull are enumerated, the facial lattice is computed and a new compact structure representing the combinatorial type of the convex hull is produced. Swart noted that the simplest of the convex hull problems is that of picking out the elements of the set $S$ which are the vertices of the convex hull. He called this the vertex problem. The facet problem is that of enumerating the facets of the polytope. The facial lattice problem produces the complete facial lattice of the hull. We are interested in the vertex problem but also generate the facets as a byproduct.

### 3.5 Parallel Algorithms

The essence of a parallel implementation is to solve the convex hull problem efficiently in terms of both the run time and the number of processors used. Parallel computers provide the possibility of substantial improvements in the running time of algorithms, allowing larger problems to be solved in a feasible amount of time. For two and three dimensional problems, parallel versions of the convex hull have appeared in the literature. Compared to the number of serial algorithms for solving such problems, the number of parallel algorithms is quite small.

#### 3.5.1 2-D Algorithms

Miller and Stout [67] presented an $O(\sqrt{n})$ time solution on an $n$-node square mesh of processors. They also implemented their algorithms on the hypercube, pyramid, tree machine, mesh-of-trees, mesh with reconfigurable bus, EREW PRAM and a modified AKS network [68]. In each of these cases, different running times were achieved with fixed number of processors. On the hypercube, the algorithm finishes in $O(\log n)$ time, a worst case
algorithm for the pyramid, tree machine, and mesh-of-trees finishes in $O(\log^3 n/(\log\log n)^2)$ time while a mesh with a reconfigurable bus uses $O(\log^2 n)$ time. The general algorithm implemented can be summarised as follows:

**Step 1:** Divide the set $S$ of $n$ planar points into two subsets $S_1$ and $S_2$, each of size $n/2$, so that all points of $S_1$ have $x$-coordinates less than those of $S_2$, and $S = S_1 \cup S_2$.

**Step 2:** Recursively identify the convex hull of $S_1$ and $S_2$.

**Step 3:** Identify the upper and lower common tangent lines (UCTL, LCTL) between the convex hull of $S_1$ and that of $S_2$.

**Step 4:** Eliminate all extreme points between the common tangent lines (i.e. extreme points of $S_1$ and $S_2$ that are inside the quadrilateral formed by the four endpoints representing the common tangent lines) and renumber the remaining extreme points.

This is shown in figure 3.4.

Atallah and Goodrich [69] give an $O(\log n)$ algorithm using $O(n)$ processors on the CREW PRAM model (i.e. the synchronous parallel model where processors have a common
memory in which concurrent reads are allowed, but no two processors can simultaneously
write to the same memory location). Their algorithm, although still based on the divide-
and-conquer method, differs in many aspects from that of Miller and Stout. The problem
is subdivided into many subproblems (e.g. \( \sqrt{n} \) instead of just two); solves all the problems
recursively in parallel and merges them in parallel to produce the final solution. This shows
an improvement of a similar parallel version on the same model of parallel architecture
using \( O(n) \) processors and a running time of \( O(\log^2 n) \) presented by Chow [86]. Their
method is paraphrased below:

**Input:** A set \( S \) of \( n \) points in the plane.

**Output:** The list \( CH(S) \). That is the list of the convex hull of \( S \) listed in clockwise
order.

**Method:** The main idea of the algorithm is to divide the problem into \( \sqrt{n} \) subproblems
of size \( \sqrt{n} \) each, solve the problems recursively in parallel, and combine the solutions
to the subproblems quickly and with a linear number of processors. This is shown
in figure 3.5 with \( n = 25 \) points.

**Step 1:** Sort the \( n \) points by \( x \)-coordinate and partition \( S \) into sets \( S_1, S_2, \ldots, S_{\sqrt{n}} \), each
of size \( \sqrt{n} \) such that \( S_i \) is left of \( S_j \) if \( i < j \).

**Step 2:** Recursively solve the convex hull problem for each \( S_i, \ i \in \{1,2,3,\ldots,\sqrt{n}\} \), in
parallel.

**Step 3:** Find the convex hull of \( S \) by computing the convex hull of the union of the \( \sqrt{n} \)
subconvex hull polygons \( CH(S_1), \ldots, CH(S_{\sqrt{n}}) \) using ALGORITHM MERGE.
ALGORITHM MERGE

Input: The input here is the collection of convex polygons $CH(S_1), CH(S_2), \ldots, CH(S_{\sqrt{n}})$.

Output: The upper convex hull $UH(S)$ of the vertices of the union of the $CH(S_i)$'s.

Method: The main idea is to find, in parallel for each $CH(S_i)$, which of its vertices are on $UH(S)$. This is done by assigning $\sqrt{n}$ processors to each $CH(S_i)$ and having each of these processors compute the upper common tangent between $CH(S_i)$ and one of the other input polygons.

Step 1: In parallel for each $i \in \{1, 2, \ldots, \sqrt{n}\}$ use $\sqrt{n}$ processors to find those points of the convex hull of $CH(S_i)$ which belong to $UH(S)$ using the steps outlined below:

Step 1.1: Find the $\sqrt{n} - 1$ upper common tangents between the convex hull $CH(S_i)$ and the remaining $\sqrt{n} - 1$ other input polygons. Let $T_{i,j}$ denote the upper common tangent between the convex hull of $CH(S_i)$ and that of $CH(S_j)$, where $T_{i,j}$ is represented by its point of contact with $CH(S_i)$ and its point of contact with $CH(S_j)$.

Figure 3.5: A partitioning of $S$ into 5 subsets
Step 1.2: Let $V_i$ be the tangent with smallest slope in $\{T_{i,1}, \ldots, T_{i,i-1}\}$ (i.e. $V_i$ is the smallest slope tangent which 'comes from the left' of $CH(S_i)$), and let $W_i$ be the tangent with the largest slope in $\{T_{i,i+1}, \ldots, T_{i,\sqrt{n}}\}$ (i.e. $W_i$ is the largest-slope which 'comes from the right' of $CH(S_i)$). Let $v_i$ be the point of contact of $V_i$ with $CH(S_i)$ and $w_i$ the point of contact of $W_i$ with $CH(S_i)$.

Step 1.3: Since neither $V_i$ nor $W_i$ can be vertical, they intersect and form an angle (with interior pointing upward). If this angle is less than $\pi$, then none of the points of $CH(S_i)$ belong to $UH(S)$. Otherwise all the points from $v_i$ to $w_i$, inclusive, belong to $UH(S)$. This is shown in Figure 3.6(a) and 3.6(b). Figure 3.6(a) shows the case when none of $CH(S_i)$'s points are in $UH(S)$ because $V_i$ and $W_i$ form an angle which is $< \pi$. In Figure 3.6(b) the points $p_2$, $p_3$ and $p_4$ are in $UH(S)$ because $V_i$ and $W_i$ form an angle which is $< \pi$.

Step 2: Step 1 has computed, for every $i \in \{1,2,3,\ldots,\sqrt{n}\}$ all the points of $CH(S_i)$ which belong to $UH(S)$ (possibly none). This step compresses each of these lists into one list to get $UH(S)$.
Aggarwal et al. [85] also achieved $O(\log n)$ time using $O(n)$ processors on a CREW PRAM. Goodrich [70], using the hull tree (a parallel data structure) on a CREW PRAM has solved the convex hull problem in two dimensions in $O(\log n)$ time using $O(n/\log n)$ processors for the case when the input points are given in a sorted order. Holey and Ibarra [83], without using the recursive or divide-and-conquer technique, also solve the planar convex hull problem on a variety of mesh-connected arrays of processors. Their approach, which is based on the Graham's scan sequential algorithm, is iterative and so avoids the overhead of the merge step in the divide-and-conquer algorithm. It also avoids presorting the points. The input points are directed into processor 0 one at a time and sorted according to their $x$-coordinates. The points are “pushed” into the next processor as new points are entered. When a new point is received from the processor's input, the new point is sorted together with the points which the processor is already storing and the Graham's scan is performed on the sorted list of points to determine those points that are extreme points. Chazelle [78] shows how to solve the problem systolically on an $n$-node linear array of processors in $O(n)$ time. Others who have studied the 2-D problem are [72], [79]. Table 3.3 illustrates the time complexity of some of the parallel versions of the 2-D convex hull problem.

### 3.5.2 3-D Algorithms

The preceding section has shown some theoretical analysis of the parallel versions of the 2-D convex hull problem. For the 3-D problem, parallel versions of the algorithms have also shown that significant speedup is attainable compared with their equivalent sequential algorithms. Aggarwal et al. [85] derived a parallel algorithm of $O(\log^3(n))$ based on the Preparata-Hong algorithm. Chow [86] in her thesis also achieved the same run time using Voronoi diagrams to control the calculation. Reif and Sen [84] have given an $O(\log n)$ randomised parallel algorithm for the 3-D convex hull problem using $O(n)$ processors on the CREW PRAM model. It should be emphasised that these presentations are largely
Table 3.3: Parallel 2-D Complexity Table

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Author</th>
<th>Complexity</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systolic array</td>
<td>Chazelle [78]</td>
<td>$O(n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>CREW PRAM</td>
<td>Chow [73, 86]</td>
<td>$O(\log_2^2n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Multiprocessors</td>
<td>Akl [74]</td>
<td>$O(\log_2n)$</td>
<td>$n^3$</td>
</tr>
<tr>
<td>SIMD</td>
<td>Akl [76]</td>
<td>$O(n^e\log_2n)$</td>
<td>$n^{1-e}$</td>
</tr>
<tr>
<td>Square mesh</td>
<td></td>
<td>$O(\sqrt{n})$</td>
<td></td>
</tr>
<tr>
<td>Hypercube</td>
<td></td>
<td>$O(\log_2n)$</td>
<td></td>
</tr>
<tr>
<td>Tree machine</td>
<td>Miller and Stout [68]</td>
<td>$O(\log_2^3n/(\log_2\log_2n)^2)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Mesh of trees</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pyramid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reconfigurable mesh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AKS network</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pyramid (ordered input)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREW PRAM</td>
<td>Goodrich and Atallah [69]</td>
<td>$O(\log_2n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>CREW PRAM</td>
<td>Goodrich [70]</td>
<td>$O(\log_2n)$</td>
<td>$O(n/\log_2n)$</td>
</tr>
<tr>
<td>Mesh array (OIA)</td>
<td></td>
<td>$O(n)$</td>
<td>$(n - 2)$</td>
</tr>
<tr>
<td>Cellular array</td>
<td>Holey and Ibarra [80]</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>D-Cellular array</td>
<td></td>
<td>$O(n^{1/4})$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>OCA</td>
<td></td>
<td>$O(n)$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
theoretical with the authors concentrating on the design and analysis of the complexity of their algorithms. Day [77, 89] instead of taking a theoretical approach, presented a practical implementation of the Divide-and-Conquer parallel version of the 3-D algorithm. This was implemented on a Meiko Computing Surface using several sizes of network up to a maximum of eight processors which were configured in the form of a hypercube. The results indicated a significant speed-up compared to the sequential version running on a Sun workstation. A speed-up of 5 was obtained using 8 processors and 1.8 using 2 processors.

3.6 Summary

The input to an algorithm for finding the convex hull is an array of points. The output is a polytope, also represented as an array of points with the property that tracing through the points produces the outline of the polytope. The algorithm simply rearranges the points in the original array eliminating unqualified candidates to leave the polytope vertices. Clearly, computing the convex hull is closely related to sorting and a sequential lower bound of $O(n\log_2 n)$ time, regardless of data, has been achieved, because often the first step is sorting the input. Jarvis's algorithm, on the other hand, uses time that varies between linear and quadratic. In Jarvis's approach, the algorithm is simple and consists of angle comparisons only. However, the major disadvantage of his gift-wrapping method is that in the worst case, when all the points fall on the convex hull, the running time is proportional to $n^2$. On the other hand, the method has the attractive feature that it generalises to three (or more) dimensions. To protect against the worst case (when all points are on the hull), it is prudent to use Graham's scan. This gives an algorithm which is almost sure to run in linear time in practice and is guaranteed to run in time proportional to $O(n\log_2 n)$.

The 2-D parallel version has differing running times depending on the architectural
model and the number of processors. In each case, there is always an improvement over the running time of the sequential version. Similarly, the 3-D problem has also benefitted from parallelisation. Surprisingly, despite the continued interest in the subject, the $n$-D problem has not yet been given adequate attention.
Chapter 4

Sequential Algorithms

The purpose of this chapter is to present sequential or serial algorithms that compute the n-D convex hull algorithm facet by facet. Some of the algorithms proposed for finding the convex hull for 2-D and 3-D were outlined earlier in chapter three with the method of Chand and Kapur [1] presenting the n-D algorithm. The divide-and-conquer method for 2-D and 3-D rely mainly on recursive partitioning of the point set followed by a merging technique to combine the partitions into a full hull. The problem in which the dimension is greater than 3 is not as well studied. In particular, the divide-and-conquer technique alone does not scale well to higher dimensions. However, Jarvis’s gift-wrapping technique for 2-D can be extended in a relatively straightforward manner to compute the convex hull for n-D problems.

Two methods are considered during the design and implementation of the sequential algorithms for the n-D problem. These are:

• Recursive Method.

• Non Recursive or Stack Based Version.

A recursive program is one that calls itself (and a recursive function is one that is defined in terms of itself). In our algorithm, after the determination of the initial facet, the algorithm calls itself recursively in order to compute the remaining edges and vertices
of the convex hull of the given set. Recursion can be removed from any program [91]. It is on this assumption that we develop the nonrecursive or stack based algorithm for the same problem. Primarily, removing recursion requires more work in implementation. Usually the values of the local variable and the address of the instruction are pushed on a stack along with the values of the parameters that are set in the procedure call. When the procedure completes its computation, it must pop or unstack the values of the local variables and return address from the stack. The removal of recursion, though a complicated task, often leads to efficient implementation and a better understanding of the nature of recursive implementations. In particular, the stack version provides a more efficient parallel implementation (see later) and allows ready access to the various stages of the gift wrapping process.

4.1 The Gift-Wrapping Technique.

The gift-wrapping technique proposed by Chand and Kapur [1] is based on the observation that every edge of \( CH(S) \) belongs to exactly two faces of the polytope \( CH(S) \), or more precisely, the intersection of exactly two faces from a set of faces describing the polytope determine an edge. The running time of this algorithm is a function of \( d \), the space dimension, \( N = | S | \), the number of points, and \( f \), the number of facets of \( CH(S) \). It has been shown by Swart [2] that the running time of the algorithm in [1] is \( O(Ndf + d^3f^2 + Nd!) \). Swart [2] also modified this algorithm to improve upon its efficiency by using the affine basis method. In general for \( d > 3 \) a face or facet can be defined in terms of its edges which are themselves the facets of a polytope in fewer dimensions. In particular, the problem is recursive and eventually reduces to a collection of subproblems involving only two or three dimensions. Three major steps are involved in the determination of \( CH(S) \) and these are:

**Step 1:** Find an initial facet.
Step 2: Given this initial facet, find its subfacets.

Step 3: Given a facet and one of its subfacets, $F$, determine the other facet containing $F$.

The facet problem is that of enumerating the facets of a polytope $CH(S)$, where each facet is represented by its affine hull. Chand and Kapur [1] and Grünbaum [88] observed that if $CH(S)$ is a $d$-polytope, each $(d-2)$-face $F$ of $CH(S)$ is contained in precisely two facets, $F_1$ and $F_2$, of $CH(S)$ and $F = F_1 \cap F_2$. To implement steps 1 through 3 above, we explain each in more detail.

In step 1, to find the initial facet, a supporting hyperplane to $S$ is constructed. This hyperplane is rotated until its intersection with $S$ is of dimension $(d - 1)$ (and therefore a facet of $CH(S)$). For simplicity it is sufficient to choose the supporting hyperplane to have the normal $(1, 0, \cdots, 0)$ and for it to pass through the minimal coordinate of $S$. Intuitively, this hyperplane can be viewed as a piece of paper with which we try to cover a facet of $CH(S)$. Suppose $F$ is the intersection of the supporting hyperplane with $S$.

We perform the following steps:

- Rotate the supporting hyperplane about $F$ until we intersect a new set of points in $S$.
- Add any new points so intersected and repeat the rotation until $F$ has dimension $(d - 1)$.

Step 2 requires us to find the facets (or edges) of a facet. Given the initial facet which is confined to $(d - 1)$ dimensions, the edges are clearly $(d - 2)$ dimensional facets, and can be determined by computing the convex hull of the facet (a simpler problem since the dimension is now $(d - 1)$).

In step 3, given a facet and one of its subfacets, a $(d-2)$-face, we are to find another facet containing this $(d-2)$-face. This can be done by gift-wrapping; rotating a hyperplane
by starting at the given facet through the \((d - 2)\)-face until the intersection with a new point is achieved. The affine hull of this point and the \((d - 2)\)-face intersected with \(S\) is the desired result. This means that in \(d\) dimensions, the facets of a \((d - 1)\) dimensional facet can be regarded as supporting hyperplanes of the convex hull so that a \((d - 1)\) dimensional facet can be found by rotating the \((d - 2)\) facet until one additional point from \(S\) is added to the hyperplane making a \((d - 1)\) dimensional facet. Specifically, we want to use the algorithm by Swart [2] which is presented here and is based on the affine basis method.

The routine \texttt{affine.hull()} is used to compute the affine hull, which is an input to other routines. The orthonormal basis of \(d\)-dimensional points in the set \(S\) is computed and returned as the function result. The associated set of affinely independent points copied from \(S\) are placed in \(A\), \(k\) is the dimensionality of \(S\). Note that \(k < d\) is possible (e.g. a square in three dimensions). The running time of each of the steps in the routines are given in parentheses after each step.

\textbf{Algorithm affine.hull()} \\

\textbf{Input:} \(d > 1, S \subseteq \mathbb{R}^d\) \\

\textbf{Output:} \(A \subseteq S\) the first points of \(S\) which form a maximal set of affinely independent points, \(k\) is the dimensionality of \(S\).

\texttt{function affine.hull}(S, d) : (A, k);

\(A := \emptyset;\) \hspace{0.5cm} (1) \\
\(p_0 := \text{first point in } S;\) \hspace{0.5cm} (1) \\
\(r := 0;\) \hspace{0.5cm} (1) \\

For \(p \in S - \{p_0\}\) do

\{check if \(p_0\bar{p}\) is representable in terms of elements of \(A\}\)
\[ v := p_0 p; \quad (|S| d) \]

for \( x \in A \) do \( (|S| k) \)

\[ v := v - \alpha x; \quad \{ \alpha \text{ a scalar} \} \quad (|S| kd) \]

end

if(\( v \neq 0 \) \( \{ p_0 p \text{ is not representable} \} \) \( (|S|) \)) then

\[ A := A \cup v \quad (kd) \]

end

return \( (\{ p_0 \} \cup \{ x + p_0 : x \in A \}, |A|) \) \( (kd) \)

end

The running time of the algorithm affine_hull() is given by \( kd |S| \) which is the time used for the Gram-Schmidt Orthogonalisation Process and this stage dominates the computation. The code for the routine is shown in Appendix C.1.11. This method of computing the affine hull makes use of the Gram-Schmidt Orthogonalisation procedure which, given a set of \( s \) linearly independent vectors \( u_1, \ldots, u_s \), we construct an orthonormal set \( x_1, \ldots, x_s \) where the \( x_i \) are suitable linear combinations of the \( u_i \), \( i = 1 \) to \( s \).

As an example, consider the following vectors

\[
\begin{bmatrix}
1 \\
1 \\
1 \\
-1
\end{bmatrix}, \quad
\begin{bmatrix}
2 \\
-1 \\
-1 \\
1
\end{bmatrix}, \quad
\begin{bmatrix}
-1 \\
2 \\
2 \\
1
\end{bmatrix}
\]

we set \( v_1 = u_1 \) and then choose \( \alpha \) so that

\[ v_2 = u_2 - \alpha v_1. \]

This implies

\[
\alpha = \frac{(v_1, u_2)}{(v_1, v_1)} = -\frac{1}{4}
\]

giving

\[
\begin{bmatrix}
3 \\
\frac{3}{4} \\
-1 \\
-1 \\
1
\end{bmatrix}
\]

65
Similarly

\[ v_3 = u_3 - \beta v_1 - \gamma v_2 \]

giving \( \beta = \frac{1}{3} \) and \( \gamma = -\frac{2}{3} \) and so we compute \( v_3 \) as

\[ v_3 = \frac{3}{8} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \]

Normalisation of \( v_1, v_2 \) and \( v_3 \) gives

\[ x_1 = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} , \quad x_2 = \frac{3}{8\sqrt{3}} \begin{bmatrix} 3 \\ -1 \\ -1 \\ 1 \end{bmatrix} , \quad x_3 = \frac{1}{\sqrt{3}} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \]

In Figure 4.1, the set \( S \) is specified as 3-D points but in fact all points can be transformed to a 2-D plane. The affine hull gives us a basis to span the plane in which \( S \) lies and so reduces the dimensionality of the problem. For example, the square in 3-D with vertices \{ (1,4,0), (4,4,0), (1,1,0), (4,1,0) \} can be reduced to a square in 2-D with vertices \{ (1,4), (4,4), (1,1), (4,1) \}. Also the affine basis method is preferred because it allows the storage of a basis rather than all points on the face. The main algorithm `convex_hull()` uses two subroutines `initial_facet()` and `rotate()` corresponding to steps 1 and 3.
Algorithm rotate(): In performing steps 1 and 3, a common routine rotate() that moves the hyperplane is required. This routine accepts the normal to the supporting hyperplane and an affinely independent set to be rotated through, \( F = \{ p_0, \ldots, p_k \} \), and returns a point \( J \in S \) such that \( AH(F \cup \{ J \} \cap CH(S)) \) is a \((k + 1)\)-face of \( CH(S) \) but \( J \) is not on the starting hyperplane. In the routine we are given a \( k \)-dimensional subset of \( S \) as defined by the affine hull \( AS \) and a set \( F \) of \( j < k - 1 \) points with outward normal defining a \( j \)-face of the convex hull. A point \( J \) and a new normal are determined such that when \( J \) is added to \( F \) a \((j + 1)\)-face is produced.

Chand and Kapur [1] give an efficient method for computing \( J \). Their method involves the computation of a vector \( \hat{e} = (e_1, e_2, \ldots, e_d) \) which together with \( \hat{n} \) (the unit outward normal to a supporting hyperplane of \( CH(S) \) that contains \( F \)) define a 2-flat (a plane) upon which the angle will be measured. The vector \( \hat{e} \) is chosen so that it is orthogonal to both affine(\( F \)) and \( \hat{n} \). For every vector \( v_p = p_0 \hat{p} \), which is projected onto the 2-flat so defined, vectors are determined so that the angle between them and \( \hat{n} \) are minimised or maximised as shown in figure 4.2. The components \( e_i \) of vector \( \hat{e} \) are computed by finding
a solution to the system of linear equations:

\[ \hat{e} \cdot \hat{n} = 0 \]
\[ \hat{e} \cdot v_i = 0 \]

where

\[ v_i = p_0 \hat{p}_i, \quad i = 1, 2, \ldots, k. \]

The projection of \( v_p \) onto the 2-flat is given by

\[ (v_p, \hat{e})\hat{e} + (v_p, \hat{n})\hat{n} \]

and the tangent of the angle \( \theta_p \) between \( v_p \) and \( \hat{n} \) by

\[ \tan \theta_p = -\frac{\hat{e} \cdot v_p}{\hat{n} \cdot v_p}. \]

The vector orthogonal to \( v_J \) is given by

\[ n^* = (v_J, \hat{n})\hat{e} - (v_J, \hat{e})\hat{n} \]

\[ n^* \cdot v_q \leq 0 \quad \text{for any} \quad q \in S \]

and it is an outward normal to a supporting hyperplane containing the \((k + 1)\)-face.
Algorithm rotate()

Input: $d \geq k > 1$, $S$ a $k$-dimensional subset of $R^d$, $AS = \text{affine}_hull(S,d)$; $F$ a $(j + 1)$-membered subset of $AS$, s.t. $j < k - 1$ and $\text{affine}_hull(F,d) \cap CH(S)$ is a $j$-face of $CH(S)$, $\hat{n}$ the unit outward normal to a supporting hyperplane of $CH(S)$ that contains $F$.

Output: $J \in S$ s.t. $S \cap AH(F \cup J)$ is a $(j + 1)$-face of $CH(S)$ and $\hat{n}^* \neq \hat{n}$ is the unit outward normal to a supporting hyperplane containing the face.

function rotate($S, AS, d, k, F, \hat{n}$) : $(P, \hat{n})$;

Pick a point $p_0 \in F$; (1)

{compute a unit vector $\hat{e} \in \text{affine}(S)$ orthogonal to $F$ and $\hat{n}$}

$AS' := \{p_0p : p \in AS - \{p_0\}\}$ (kd)

$F^* := (j + 1)$ by $k$ matrix of the vectors $\{p_0p : p \in F - \{p_0\}\} \cup \{\hat{n}\}$ is represented in the basis $AS'$; (jkd)

Pick a solution $e'$ to $F^*e' = 0$; (jkk)

$e := e'$ translated back into $R^d$ with $AS'$; (kd)

$\hat{e} = e/\|e\|$; (d)

Compute the minimum and maximum of $\tan \theta_p = -(\hat{n}.v_p)/(\hat{e}.v_p)$ over all points in $p$ and $S$; (d)

$J = \text{one of the points computed above whose tangent was not positive or negative infinity}$; (d)

$n^* = (v_J,\hat{n})\hat{e} - (v_J,\hat{e})\hat{n}$; (d)

$\hat{n}^* := n^*/\|n^*\|$; (d)

if $\hat{n}.p_0p > 0$ for any $p \in S$ {i.e. check the orientation of the outward normal}

then $\hat{n}^* := -\hat{n}^*$;
return \((J, \hat{n}^\ast)\);

end

Normally, once a facet is known, a single call of \texttt{rotate()} will produce a new facet but at the start of the algorithm a number of rotates are required to ensure that the facet has dimension \(d - 1\). The running time of \texttt{rotate} is given as \(jkd + d | S|\). The two major steps that contributes to the running time of this routine are the formation of the \((j + 1) \times k\) matrix and the computation of the angle. The code for \texttt{rotate()} is given in Appendix C.1.9.
Algorithm *initial_face*

**Input:** $d \geq k > 1, S$ a $k$-dimensional subset of $\mathbb{R}^d$ and $AS = \text{affine.hull}(S, d)$.

**Output:** $F$ an affinely independent subset of $S$ s.t. $AH(F) \cap CH(S)$ is a facet of $CH(S)$ and $\hat{n}$ is its normal vector.

function *initial_face*$(S, AS, d, k)$ : $(F, \hat{n})$

Pick an $i$ such that not all the points in $S$ have the same $i$th coordinate;  \hspace{1cm} (d)

$F := \text{set of points in } S \text{ with minimal } i\text{th coordinate}; \hspace{1cm} (|S| \cdot d)$

$F := \text{affine.hull}(F, d) \hspace{1cm} (|S| \cdot kd)$

$\hat{n} := \text{projection of the outward normal } (0, 0, \ldots, -1, 0, \ldots, 0)$

\hspace{1cm} \{ i.e. vector with -1 in the $i$th component \} onto affine$(AS) \hspace{1cm} (kd)$

while $|F| \leq k - 1$ do \hspace{1cm} (k)

$(P, \hat{n}) := \text{rotate}(S, d, F, \hat{n}); \hspace{1cm} (k(kk + d \cdot |S|))$

$F := F \cup P; \hspace{1cm} (kd)$

end;

return $(F, \hat{n}) \hspace{1cm} (d)$

end

The running time of $\text{initialfacet}()$ is given as $kd \cdot |S| + k^3d$. This is made up of the sum of the time for computing the affine hull and the rotation step. The code is presented in Appendix C.1.10. We are now ready to see the whole convex hull algorithm using the routines discussed above.
4.1.1 Recursive Method

Algorithm convex_hull

Input: $d \geq 1, S$ a $k$-dimensional subset of $R^d$, $AS = \text{affine}_hull(S, d)$.

Output: $CH \subseteq S$, a set of vertices of $CH(S)$. $FA$ a family of sets of affinely independent points $\{FA_1, \ldots, FA_{f_{d-1}}\}$ such that $\text{affine}(FA_i)$ is a hyperplane containing the $i$th facet of $CH(S)$.

function convex_hull($S, AS, d, k$) : ($CH, FA$);

{Check for a one dimensional set} (1)
if ($k = 1$) then

Let $AS = \{p_0, p_1\}$;

$\min := p \in S \text{ s.t. } p_0 p p_1$ is minimised; $|S|d$

$\max := p \in S \text{ s.t. } p_0 p p_1$ is maximised;

return $\{|\max, \min\}, \{|\max\}, \{|\min\}; /*$ check for a simplex */

{Check for $CH(S)$ a simplex, this is a SIMPLEX BYPASS}
if $|S| = k + 1$ then (1)

return $(S, \{FA \subseteq F : |F| = k\})$ ($kd$);

$(F, \hat{n}) := \text{initial_facet}(S, AS, d, k)$; ($kd |S| + k^3 d$

{Find the rest of the facets}
Edge list := $\emptyset$ (1)

$CH := \emptyset$

$FA := \emptyset$

{Find each $(k-2)$-face’s facet}
do $(f_{d-1})$

$FA := FA \cup \{F\}$; ($f_{d-1}kd$)

72
Pick a point $p_0 \in F$; \quad (f_{d-1})

$F' := \{ p \in S : \langle p - p_0, \hat{n} \rangle = 0 \}; \quad (f_{d-1} \mid S \mid d)$

$(\text{FCH, FFA}) := \text{convex hull}(F', F, d, k - 1); (\sum_i T(F'_i, k - 1))$

$\text{CH} := CH \cup \text{FCH}; \quad (f_{d-1} \mid S \mid d)$

while($\text{FFA} \neq \emptyset$) do \quad (2f_{d-2})

\{ remove facet$(E, \hat{n})$ from FFA; \quad (2f_{d-2}kd \log f_{d-2})

if ($E \in \text{EdgeList}$) remove facet$(E, \hat{n})$ from EdgeList;

else add $(E, \hat{n})$ to EdgeList; \}

if($\text{EdgeList} \neq \emptyset$) \quad (f_{d-1})

\{ Pick an $(E, \hat{n})$ from the edge list; \quad (f_{d-1}kd)

$(P, \hat{n}) := \text{rotate}(S, \text{AS}, d, k, E, \hat{n}); \quad (f_{d-1}(k^2d + \mid S \mid d))$

$F = E \cup \{ P \}; \}$ \quad (f_{d-1}d)

} \quad (h-l)

while($\text{EdgeList} \neq \emptyset$);

return $(\text{CH, FFA})$;

end

The time for one call of this procedure on a set $S \subseteq R^d$ of dimension $k \leq d$ and cardinality $n > k + 1$ is given by

$$T(S, k) = O(k^2d f_{d-1} + dnf_{d-1} + kdf_{d-2} \log f_{d-2} + \sum_i T(F'_i, k - 1))$$

where $f_k$ is $f_k(P)$ and $F'_i$ is the set of points sharing the hyperplane with the $i$th facet of $P$. The boundary conditions are $T(S, k) = \Theta(kd)$ if $|S| = k + 1$, and $T(S, 1) = \Theta(n)$.

The main driving routine in the program is the function $\text{convex hull}()$ that takes a set $S$ of dimension $n$ and affine basis $AS$ a subset of $S$ with dimension $k$. The function returns the set of points $\text{CH}(S)$ which are the vertices of the convex hull and $FA$ the lists of facets. The routine is recursive. Given the set $S$, the routine firstly determines the number of points in $S$ and checks for a 1-dimensional set which in this case is a straight
If there are only two points, they are returned as the vertices and the edge. In the case where there are more than two points, the end points will be returned as the vertices of the convex hull and the edge. If the input set \( S \) is not a 1-dimensional set, the routine calls the function `Initial_f{}ac{}et()` to compute the initial face to start the computation of the faces. All the points on the initial face are copied and used as the input to call the convex hull routine recursively to compute the vertices and edges of this face. The vertices and edges so computed are stored in the \( FCH \) and \( FFA \) lists respectively. Before storing the vertices and edges so computed, the routine checks the already existing vertices and edges to ensure that there is no duplication. The routine keeps the edges computed at each recursive call in an `EdgeList`. While there are still more edges in the `EdgeList`, an edge is picked with its outward normal and rotated by calling the function `rotate()` in order to determine a new face and the process is repeated. A simplex bypass is added as a quick exit for recursion. In order to check for a simplex bypass, consider a set with \( k = 2 \) having three points \( S = \{(3,3), (3,1), (1,1)\} \), then \( |S| = 3 = k + 1 \), so the points are vertices of the convex hull and the edges (faces) are the permutation of \( k \) vectors. e.g. \( k = 2 \) and \( CH(S) = S \). \( FA = \{\{(1,1), (3,3)\}, \{(1,1), (3,1)\}, \{(3,3), (3,1)\}\} \).

### 4.1.2 Stack Version

As a variation to the recursive partitioning method, we have also implemented a non recursive or stack based version. Non recursive methods are more efficient and allow a better management of dynamic memory allowing larger problem sizes to be processed. A stack that is proportional to the size of maximum dimension is created in order to solve the convex hull problem. Each level of the stack is a record that contains the following information:

- Set of points \( S \)
- Set of points \( AS \) which is the affine basis of \( S \)
• The dimension $k$ of $S$

and the following lists which are initially set to the empty list.

• EdgeList, Elist

• Convex hull list, CH

• Face list, FA

The **Elist** is used as a storage for the computed edgelist determined during the computation. Once an edge is found twice, the two adjacent faces with this edge as their intersection have been found and this edge can be deleted from further consideration.

Initially, the stack level is set to zero and the *initial_facet()* routine is called to determined the initial face to initiate the computation of the rest of the faces and vertices of the object. The edges of the initial face so determined are preserved in the **Elist**. When an edge is selected it is rotated and a new face determined, copying all the points on that face onto the next level of the stack with the maximum dimension decremented by 1. With the new set of vertices, affine basis and dimension, the **CH**, **FA**, and **Elist** are computed. This process is continued until the highest level on the stack is equivalent to the maximum dimension size. After the computation of the **Elist**, **CH**, and **FA**, the face so computed has to be unstacked before a new edge is selected for consideration. The components in **Elist**, **CH** and **FA** on stack level $sp$ are unstacked to the lower level $sp - 1$ ensuring that there is no duplication of members. This is continued until the stack level is again reduced to zero. Another edge, if any, is then picked, rotated and the process repeated to determine yet another face. Edges are picked until the **Elist** at $sp = 0$ is empty and in that case all the faces have been computed.

Figure 4.3 illustrates the stack version with a 3-D example. In each of the partitions, the steps described here are executed. The **Elist**, **CH** for the convex hull and **FA** for the
facets list are initially set to null sets at level $sp = 0$. $S$ is the set whose convex hull is to be determined and $AS$ is its affine basis with $k$ representing the dimension. First of all, the initial face is computed to start the execution of the program. All the points of $S$ that are on the initial face are copied into $S'$ which is on the next higher level of the stack, i.e. level $sp = 1$. The affine basis of $S'$ is computed which we represent as $AS'$ for the $k - 1$ dimensional set. $(Elist)'$, $CH'$ and $FA'$ are then computed for that level on the stack. At $sp = 2$ the problem is trivial as it reduces to straight lines where the end points form the vertices of the hull i.e. $CH$ and two extreme points define an edge which in turn describes a facet. To complete the computation on that face, the $(Elist)'$, $CH'$ and $FA'$ lists are now unstacked from $sp = 2$ to $sp = 1$. At each level of the stack, the algorithm checks the list at the lower level before adding the list from the upper level of the stack to ensure that there is no duplication. This is repeated until $sp = 0$.

A new edge is now selected from the Elist and a rotation is performed along that edge to describe a new face where the above steps are repeated on that face to compute $CH$ and $FA$. This rotation step is repeated as long as there are more edges in the Elist. A number of rotations may be necessary depending on how complex the shape of the object

![Table and Figure](image)

Figure 4.3: Stack Implementation For 3-D
A merge of two subproblems followed by convex hull computation using the sequential stack version is then carried out with the final results emerging after the last merge and compute process. The sequential stack algorithm is summarised as follows:

**Function Convex Hull(S,AS,n,k)** : (CH,FA);

```plaintext
{ /* setup the stack */
  sp = 0; Stack[sp].S = S; Stack[sp].AS = AS; Stack[sp].k = k;
  Stack[sp].Elist = \emptyset; Stack[sp].FA = \emptyset; Stack[sp].CH = \emptyset;
  do{
    if(Stack[sp].CH = \emptyset and Stack[sp].FA = \emptyset)
      { if(k = 1) /* a 1 - dimensional set */
        { AS = \{ p_0, p_1 \};
          min = p \in S such that p_0 \vec{p}_1, p_0 \vec{p}_1 is minimised
          max = p \in S such that p_0 \vec{p}_1, p_0 \vec{p}_1 is maximised
          return (\{max,min\}, \{max\}, \{min\});
        }
        if (| S | = k + 1)
          return (S, \{ F \subseteq S : | F | = k \}); /* check for a simplex */
        else
          { (F, n) = initial facet(S,AS,n,k)
            FA = FA \cup \{ F \}
            F'' = \emptyset;
            Pick a point p_0 \in F;
            F'' = \{ p \in S : p_0 \vec{p}. n = 0 \};
            sp = sp+1; Stack[sp].S = F'' ; Stack[sp].AS = F; /* stack the face */
            Stack[sp].k = Stack[sp].k-1;
            EdgeList = \emptyset; Stack[sp].CH = \emptyset; Stack[sp].FA = \emptyset;
          }
      } /* unstack completed faces */
      if (Stack[sp].CH \neq \emptyset and Stack[sp].FA \neq \emptyset)
        while(EdgeList = \emptyset and sp > 0)
          { while(Stack[sp].CH \neq \emptyset) { Pick p \in Stack[sp].CH}
            if(p \notin Stack[sp-1].CH)
              Stack[sp-1].CH = Stack[sp-1].CH \cup p; } /* Insert point */
            while(Stack[sp].FA \neq \emptyset)
              { remove facet (E,-) from Stack[sp].FA;
                if(E \in Stack[sp-1].Elist) remove (E,-) from EdgeList
              }
          }
  }
}
```

77
else add (E, r) to Stack[sp-1].Elist
}
Delete lists: Stack[sp].S; Stack[sp].AS;
Stack[sp].k = 0; sp = sp-1;
}
if (EdgeList ≠ Ø) /* get next face */
{
Pick an (E, n̂) from EdgeList;
(p, n̂) = rotate(S, AS, n, k, E, n̂);
F = E ∪ {p};
FA = FA ∪ { F }
Pick a point p₀ ∈ F
F' = {p ∈ S : p₀p₁. n̂ = 0};
/* stack the face */
sp = sp +1; Stack[sp].S = F'; Stack[sp].AS = (F, n̂);
Stack[sp].k = Stack[sp-1].k-1;
}
}while(EdgeList ≠ Ø or sp > 0)
return (Stack[0].CH, Stack[0].FA);

4.2 Sequential Implementation

In our programs we employ the C programming language with sets implemented as circular
linked lists. Sets of sets (i.e. EdgeList and FA) are circular lists augmented with a vector
for the facet normal n̂. This representation follows because the pair (E, n̂) in the algorithm
define a facet in terms of points on the facet and the outward normal. The routines for
manipulating the points and the edges are shown in Appendices C.1.1 and C.1.2. The
non-recursive method is intended to provide a better management of dynamic memory
allowing larger problem sizes to be processed in a distributed memory implementation.
The vectors or points are represented as arrays. The list structure is to provide a dynamic
memory allocation which allows the list to expand or shrink depending on the size of the
data.

Some other auxiliary routines are provided for development of the algorithms and are
described below. First a routine Remove_Duplicate_Points() (Appendix C.1.6) exam-
ines the set, eliminating points that appear more than once, if any. The points are then sorted lexicographically according to their x co-ordinate using the routine Quick_Sort() (Appendix C.1.3) with average time $O(n \log_2 n)$ and $O(n^2)$ in the worst case. The algorithm then splits the ordered set of points into $p$ subproblems, where $p$ is the number of partitions, corresponding to the number of processors to be used in the parallel implementation, solve all the subproblems by calling the sequential algorithm Generate_Hull() (Appendix C.1.4) to compute the convex hull for each of the subproblems and then merge all the subproblem solutions to obtain the solution to the original problem.

4.3 Program Testing

Program testing is that part of the validation process which is normally carried out during implementation. Testing entails exercising the program using data similar to the real data the program is designed to execute on, observing the program outputs and inferring the existence of program errors or inadequacies from anomalies in that output. Testing in fact is meant to reveal program errors but in our context it is also used to assess overheads of various implementation strategies. For very large programs it is unrealistic to attempt the testing process as a single unit. Large programs are built out of procedures and functions. Testing the system as a whole will make it difficult to detect and identify errors. Testing could be carried out in stages.

In order to test our algorithms, we have to design our test data to cater for shapes with peculiar characteristics. This is because some of the algorithms seem to perform better with some test data than with others. In particular we have considered the following:

Type 1: Hulls with a small number of vertices but many interior points.

Type 2: Hulls with many points on the faces.

Type 3: Hulls with many vertices and few interior points.
The experiments were repeated with sets of points of different sizes in 2-D to 4-D problems.

For problems greater than 4-D, we did not try them because we ran out of memory each time we made an attempt. Table 4.1 illustrates the dimensions and size of data set we have used to test our algorithm. This data is used to test our algorithms using the partitioning technique discussed in chapter five. Another noticeable feature in the table is the fact that the different options mentioned in Type 1 to Type 3 above are adequately catered for in our test data. Running our algorithms with the different test data will reveal how the size, dimension and number of facets may affect performance. For example, a problem in 2-D of size 50 with 4 vertices on the convex hull will give a different running time when compared with a similar problem in 3-D. If we consider a 2-D space, the shapes in Figure 4.4 (a) to (c) demonstrate some of the shapes that we considered when generating our test data. The convex hull in Figure 4.4(a) has very few vertices as against those of (c) with many vertices while in (b), the edges have more than two points. Our algorithm is designed to trap such features and return only the two extreme points eliminating those points that are between the vertices. The size of the test data ranges from as small as 25 points to as many as 4000 points thus satisfying conditions (1) to (3) above.
Figure 4.4: Types Of Shapes In 2-D

The characteristic mentioned in (3) is satisfied in the test generator for the implementation discussed in chapter six. In fact all the points generated to test our implementation in chapter six consists of points that are all on the vertices of the convex hull. Generate_Test2() (Appendix C.2.2) and Generate_Test3() (Appendix C.2.3) are used to generate these data and their detailed discussions are clearly given in the appropriate section. These sets of data were necessary because our algorithms in that chapter were designed for problems with many edges and vertices. Data items designed for the partitioning technique will perform poorly if used. The data for Type 1 are for the partitioning techniques in chapter five.

4.4 Design Of Test Data

The input data to the convex hull program is a set of points in $d$ dimensions and the output is the vertices and faces of the convex polytope. Planning the testing of this algorithm involves formulating a set of test cases which are akin to the real data that the system is intended to manipulate. The test data consists of $d$, the dimension, $n$ the number of points in the set $S$ and the vectors or points. The aim was to ensure that the program
responds as expected to both valid and invalid input, and that it performs to specification. Separate and different codes were written to generate data for the programs in chapters five and chapter six. This variation is necessary because of the manner in which the programs are designed to manipulate the data.

### 4.4.1 Test Generation For Type 1 Hulls

In order to obtain data to test our programs, codes were written to generate the test data. The algorithm uses the standard C random number generator to generate the test data. A routine `Generate_Test()` (Appendix C.2.1) was written for this purpose. A set $S$ which contains the vertices of the convex polytope are given as input data. The routine `Generate_Test()` first of all computes the convex hull of the given set producing $FA$ as its facets. A random number seed is then given as input to activate the random number generator. The algorithm also requires $r$ the total number of points to be generated. The required number of points is then generated randomly inside the polytope. To eliminate unnecessary duplication of points, the routine `Remove_Duplicate_Points()` checks and removes points that are duplicated. The process repeats until $r$ points are produced and aborted after a large number of trials. Also to ensure that the points generated fall inside the convex polytope, a routine called `Check_Hull()` (Appendix C.1.5) ensures that all the points generated are within the specified boundary and this uses the faces computed from the initial convex hull. A set $S$ consisting of the convex hull as a subset together with the additional points generated is returned and this will act as our input data.

### 4.4.2 Test Generation For Type 2 Hulls

To exercise the facial lattice program in chapter six it is necessary to generate complex hulls with few or no interior points. The above mechanism is not suitable.

The data used for the timings were generated using programs written to produce $S$, the set of $n$-dimensional points. For the 2-D case, a centre $c$ is chosen on the plane and a
constant radius $r$. With the starting radius, and rotating in an anticlockwise direction, an angle $\theta$ is formed with another point say, $p_i$ such that arc $pp_i$ subtends an angle $\theta$ at the centre of the circle. By stepping with this constant angle and radius around the circle, the points so generated are used as our test data to compute the vertices of the convex hull of $S$. For the 3-D object the method is easily extended with two additional points projected in opposite direction as the vertex of the object as shown in Figure 4.5. This is a simplified representation of a circular structure with a square base, but the vertex projected in opposite directions. The shape can be viewed as two separate pyramids on a common square base.

The next set of data aims at generating objects with more vertices and edges. Figure 4.6 shows a section of the 3-D pyramidal shape and the faces that could be computed in parallel. This object can be viewed as a pyramidal structure built with rectangles in such a way that the square on the next upper level is smaller than the one immediately below it. The angle of inclination at each square is varied so that the vertices are not co-linear. The algorithm could be modified to produce a similar object in the opposite direction both having a common base which is the initial square, thus resulting in a shape with
Figure 4.6: Illustration of 3-D Pyramid Shape

more vertices and edges. Type 3 data can be generated by employing a mixture of the techniques for Type 1 and Type 2 data.
Chapter 5

Implementation Using Partitioning

The algorithms in this chapter combine divide-and-conquer partitioning techniques with
the gift-wrapping concept discussed in chapter four. Both recursive and non-recursive
(stack based) algorithms have been implemented using master-slave and fanin tree ap­
proaches in shared memory (Encore Multimax) and message-passing (Transputer – Meiko
Computing Surface) architectures. The performance of the parallel versions are monitored
with several partition sizes on different numbers of processors running on the same parallel
machines.

5.1 Sequential Method

The sequential Divide-and-Conquer method that we propose is given here in this section.
The main idea is to divide a problem into $p$ subproblems of approximately equal size, solve
the subproblems and merge the solutions to the subproblems. Our sequential program
is based on both the recursive and stack versions of the algorithms presented in chapter
four. Both versions were implemented on the shared memory and transputer architectures,
running each version on one processor of each machine. Initially, the points are sorted
lexicographically using the quicksort algorithm. Duplicate points are also removed from
the list. The computing time for the sequential algorithm as stated earlier in chapter four
is given by $\Omega(f N d + d^2 f^2 + d! df)$ where $f$ is the number of facets of $CH(S)$. The $f^2$ term
can be reduced to \( f \log_2 f \) if we use binary tree data structures (as in Swart [2]). Clearly when \( N \gg f \), or \( d \), the key to a fast convex hull algorithm is the ability to eliminate large numbers of points from \( S \) as quickly as possible. The algorithm splits the ordered set of points into \( p > 1 \) partitions from which \( p \) convex hulls are generated by calling the sequential program on each partition. The subconvex hulls are then merged to form the complete hull. The algorithm is given as follows:

Algorithm \( CH(S) \)

Input: A set \( S \) of \( n \) points in space.

Output: The list \( CH(S) \) i.e. the vertices of the convex polytope of \( S \).

Step 1: Sort the \( n \) points of \( S \), and partition \( S \) into sets \( R_1, R_2, \ldots, R_p \), where \( p \) is the number of processors.

Step 2: Solve the convex hull problem for each \( R_i, i \in \{1, 2, \ldots, p\} \), using the sequential convex hull routine. After the return of each computation, we will have \( CH(R_i) \) for each \( R_i \).

Step 3: Find the convex hull of \( S \) by computing the convex hull of the union of the \( p \) convex polytopes \( CH(R_1), \ldots, CH(R_p) \). This is done by using algorithm \( Merge \).

Merge

Input: The collection of convex polytopes \( CH(R_1), \ldots, CH(R_p) \).

Output: The list of points consisting of the vertices of \( CH(S) \).

For \( i = p \) down to 2 /* loop 1 */

begin

\[
CH(R_{i-1}) = CH(R_{i-1}) \cup CH(R_i)
\]

86
Generate_Hull(CH(R_{i-1}))

CH(R_i) = \emptyset

end.

The sequential merge algorithm is performed in loop 1. Suppose there are \( p \) partitions, the convex hull of each partition is computed using the sequential algorithm. The first merge step is to find the union of the set of points in partitions \( p \) and \( (p-1) \) and computes its convex hull. This result is in turn merged with partition \( (p-2) \) and the same process is repeated until the final merge appears in partition 1 where the vertices of the convex polytope are filtered out. This clearly demonstrates that at each step two subproblems are merged together followed by a computation step which finds the vertices of yet another subproblem. In the case where the size of the partitions is reasonably large and not all the points are on the convex hull, the first call of the sequential algorithm greatly reduces the number of points to be considered in the subsequent stages by eliminating the points that are interior to each subconvex polytope. The timing for computing the convex hull for the different partition sizes and the number of points using data of Type 1 were recorded (see chapter 4). This will be compared against the time used to compute the convex hull of the same problem using the same number of partitions in parallel.

The parallel implementation of the \( n \)-D convex hull algorithm discussed in this chapter is modelled by a fanin tree structure. The main approaches in which the fanin tree can be implemented depend on the architecture available. Assuming an unlimited number of processors we can consider the following approaches:

1. Simulate Levels Of The Tree.

2. Emulate The Tree In Hardware.

3. Hybrid Approach
In the following sections we will consider these methods for both shared memory and distributed memory architectures. Before proceeding, it is worthwhile considering the problems involved in the various approaches.

In the shared memory machine, we implement the simulated fanin tree. The tree is simulated level by level by reusing some of the processors. This is suitable for a shared memory implementation because the bus traffic is considerably reduced by simulating the tree at different levels. At each level of the simulation, the number of processors being utilised is also reduced. To model a tree in hardware using a shared memory architecture will present some difficulties because the machine is a bus-based architecture and will suffer from communication delays due to too much traffic.

In the message passing paradigm we have emulated the tree in hardware as well as simulating the tree level by level, adopting a master-slave relationship and reusing some of the processors. These implementations are enhanced by the architectural design of the distributed memory machines. For the transputer machine that we use to implement our algorithms, the four bidirectional links between each processor promote the exchange of messages among processors. The different methods that we use to model the tree in the message passing architecture are discussed in more detail in section 5.4.2.

The hybrid approach seeks to combine options 1 and 2 above in its design. Basically, the initial partitions are distributed to the slave processors where the sub convex hulls are computed. Two neighbouring processors merge their results and one of them recomputes the new subhull. This in turn merges with another and the process is repeated until the last two processors merge where the final result will be filtered out. The proposed method is illustrated in figure 5.1. Here P1 to P4 compute their respective subhulls. In the next stage P1 merges with P2 while P3 merges with P4 and new subhulls are computed. The final stage involves the merging of P1 and P3 followed by the computation of the final convex hull. We have not implemented this method because of problems with comparison.
between the architectures available and the communication difficulties.

5.2 Shared Memory Implementation

The convex hull problem has a solution which is expressible directly by recursion. The ability to map the solution onto a recursive function leads to an elegant and natural implementation. The power of recursion is utilised here since the solution can be expressed by successively applying the same solution to subsets of the problem. The recursive convex hull() routine is given in chapter four. The parallel implementation of this version in the shared memory architecture now follows:

Parallel Convex Hull()

Input: A set $S$ of $n$ points.

Output: A list $CH(S)$ i.e. the vertices of the convex hull.

Method: Step 1: Sort the $n$ points by minimal first coordinate, and partition $S$ into sets $R_1, R_2, \ldots , R_p$ such that $R_1, R_2, \ldots , R_p = partition(S, d)$
**Step 2:** Recursively solve the convex hull problem for each $R_i$, in parallel by assigning each partition to a processor. Each processor calls the sequential recursive $Convex\_Hull()$ routine concurrently to compute $CH(R_i)$. After the parallel recursive call returns we will have $CH(R_i)$ for each $R_i$.

**Step 3:** Find the convex hull of $S$ by computing the convex hull of the union of the $CH(R_1), CH(R_2), \ldots, CH(R_p)$. This could be achieved by using algorithm $Merge2()$ described below:

Algorithm $Merge2()$:

**Input:** The collection of convex polytopes $CH(R_1), \ldots, CH(R_p)$.

**Output:** The convex polytope of the vertices of the union of $CH(R_i)$'s. i.e. $CH(S)$.

```
procs = \left\lfloor \frac{p}{2} \right\rfloor
if procs odd CH(R_{p+1}) = \emptyset
while(procs \neq 1) /* loop 1 */
{
    For i = 1 to procs
    {
        CH(R_i) = CH(R_i) \cup CH(R_{[p/2]+i})
        CH(R_{[p/2]+i}) = \emptyset /* loop 3 */
        Generate_Hull(CH(R_i)) /* loop 2 */
    }
    procs = \left\lfloor procs/2 \right\rfloor
}
```

$Generate\_Thread\_Hull(R_i)$

/* Computes the convex hull of set $R_i$ producing vertices in $CH(R_i)$ and facets in FA */
if(IsEmpty_Plist(Ri) ≠ TRUE)

if(Ri ≠ GetNext_Point(Ri))

Ri = Remove_Duplicate_Points(Ri, n)
Quick_Sort(GetNext_Point(Ri), GetPrev_Point(Ri), n)
Affine_Hull(Ri, n, AS, k)
Convex_Hull(Ri, AS, n, k, CH(Ri), FA)
else
Return single point as CH(Ri)
else
MakeEmpty_Plist(CH(Ri))
MakeEmpty_Elist(FA)

The main idea is to merge and to compute in parallel the convex polytope of the union of two sub convex hulls by using p/2 processors at each stage. If the number of subproblems is odd, the algorithm generates an additional partition which is empty so that an even number of partitions are obtained. The points inside each of the CH(Ri) need not be considered any further because they cannot be vertices of CH(S). This algorithm could be summarised as follows:

R1, R2, ..., Rp = partition(S, d)
/* Find the convex hull of partitions in parallel */
For i = 1 to procs
THREADcreate(Generate_Thread_Hull, i, 0, ATTACHED, 30*1024, 2)
while(THREADjoin())
/* merge the partitions in parallel */
if(procs /2 = 0)
procs = procs/2

91


```plaintext
else
    procs = procs/2 + 1
while(procs ≠ 1)
{
    For i = 1 to procs
        THREADcreate(Merge2, i, 0, ATTACHED, 30*1024, 2)
while(THREADjoin())
    if(procs /2 = 0 or procs = 1)
        procs = procs/2
    else
        procs = procs/2 + 1
}

CH(S) = CH(R_1);
```

This algorithm is a sequential coding of a binary fanin tree algorithm where the $CH(R_i)$ are computed on $p$ processors and then merged and further reduced as they filter up the tree with $CH(S)$ emerging from the root. For a particular iteration of loop 1, $j$ say, loop 3 followed by loop 2 is executed on level $j$ of the tree with a tree node performing the lexicographic set union of two lists of points followed by applying the recursive Convex Hull() to the result. A crude timing estimate can be given by $\Omega(\log_2 p + 1)(f_{\max}^n d + d^3 f_{\max}^2 + d^2 f_{\max})$ assuming that the last partition at the root contains all $N$ points (i.e. the input set was the set $CH(S)$ and must be an upper bound for the partition size at all the other levels in the tree. The value $f_{\max}$ is the maximum number of facets in the hull of any partition. Alternatively, we can use the bound $\Omega((\log_2 p + 1)(f_{\max}^n d/n + d^3 f_{\max}^2 + d^2 f_{\max})$ if $n/p$ is the maximum number of points in any partition which in general is unknown. Observe that although we can guarantee the size $n/p$ is true for the starting partitions it may not be true once merging occurs and points are eliminated. However, it
is likely to hold for convex hulls with a small number of vertices because each partition is likely to eliminate few interior points during computation. We conclude that these results are comparable to previous parallel methods on fixed number of processors discussed in chapter three and in any case approximate linear performance in $n$ for small $p$, $d$ and $f$ as expected compared to $n \log_2 n$ for most sequential algorithms using divide and conquer methods. Notice that both our sequential and parallel methods involve partial sorting so that the $n \log_2 n$ condition can be omitted since no speedup is expected from that portion of the program. The diagram in Figure 5.2 illustrates the method using eight processors. The merge and convex hull computation takes place at subsequent levels of the tree. The arrow indicates the direction of fanin. The example in the diagram illustrates a perfectly balanced tree but in general this may not be the case. Some examples of unbalanced trees include cases where $p$, the number of processors, is odd or when $p$ cannot be expressed as a power of 2 (e.g. $p = 10$). Our method deals with this automatically but degrades performance. Where the number of processors at a particular level of the tree is odd, an empty sub convex hull is created and this is merged with the extra sublist. By this approach, a balanced tree is created at the expense of performance.
5.3 Results From Shared Memory Machine

The programmer does not have control of the allocation of either processors or storage in the shared memory implementation. The libraries allow multiple tasks to be setup and they are allocated to processors by the operating system (this is to allow flexibility in a multi-user environment). Different versions of the proposed algorithm described above have been implemented in the shared memory architectures.

This section on practical implementation and results demonstrates the performance of our techniques on the shared memory machine. All timings were done at off peak times. The EPT library provides a facility whereby the system clock can be started. In all cases our timings exclude the times used for reading the input and writing the output from and to files. A comparison of the serial time with the potential parallel time for a divide-and-conquer construct-and-merge algorithm shows that a significant speedup is possible.

We have used up to 4000 points to test run our algorithms as shown in table 4.1. The experimental data used to test our algorithms were generated using Test Generation For Type 1 Hulls. This is discussed in detail in section 4.4.1 of chapter four. Polytopes of different shapes were considered. For example, in 2-D we consider shapes with three, four, six, sixteen and twenty-six vertices on the convex hull to illustrate a triangle, a quadrilateral, and a hexagon etc. each showing an increase in the number of vertices and faces of the shape under consideration. Similar trends are followed for the 3-D and 4-D polytopes. The generated set was then split into the required number of partitions. Execution times were then recorded in microseconds for the serial and parallel algorithms using 2 to 6 partitions (processors). Tables 7 – 28 of Appendix A show the timings recorded for the various data sets used to test our algorithms. The performance characteristics – speedup and efficiency discussed in section 2.12.1 and 2.12.2 of chapter two, and used to
characterise our performance were computed from these tables. Tables 7(2)27 of Appendix A show the timing recorded for the recursive version using threads and microthreads with the dimension and the number of points on the convex hull clearly stated for each problem size. Similarly, the timing for the stack version on similar problems are recorded in Tables 8(2)28 of Appendix A. In all cases, what is easily noticeable is the fact that significant improvements of the running times of the parallel algorithms over the sequential ones are achievable. In particular the 2D problem reaches its optimal speedup of 2 when using 2 processors. This apparent lack of overhead can be attributed to some bookkeeping exercises in the architecture. The speedup for the same problem size decreases as the dimension of the problem increases. A plot of the speedup against the number of processors for some of our results are shown in figures 5.3 – 5.8 which are presented in the graphs. A common feature in both the recursive and stack versions is the fact that the running times depend on the problem size, the dimension and the number of facets of the convex hull. For example in 2-D, using the recursive implementation, we have shown the performance from 50 to 4000 points with 26 vertices on the convex hull as in Table 15 of Appendix A. This is illustrated in figure 5.3. In this case the speedup increases quite rapidly with an increase in the problem size. 3-D with 12 vertices demonstrates the effect of 2000 points (see Table 23 of Appendix A). Figure 5.5 shows the performance using microthreads and again the speedup increases steadily but not as much as it was in the 2-D case. This is because of the increase from 2-D to 3-D problem. In the 4-D case, using a problem with 6 vertices on the hull and a problem size of 500 points is shown in Table 27 of Appendix A. Figure 5.4 gives the representation. In contrast to the above, the stack version has the capacity of running a larger problem size. This is shown using 3-D with 12 vertices on the hull and 4-D with 6 vertices on the convex hull Tables 24, 28 of Appendix A where problem sizes of up to 4000 points were used. The stack implementations for similar problems are illustrated in figures 5.6, 5.7 and 5.8. The
problem in the recursive version is as a result of the combinatorial nature of the point and edge data structures. Each call to the algorithm generates new vertices and edges which are stored and eventually fills up a lot of space in the memory. This is likely to happen when the shape of the object has a lot of faces and vertices on the convex hull. Also, each sub problem generates its respective results (vertices and edges) which also contributes to the increase in the storage space in memory.

However, a common feature is that the speedup increases as the problem size increases. This is due to the fact that a lot of points are eliminated during the first stage of the computation and the steps involving the merge are less significant. Also worth noting is the fact that when using fewer number of processors, the speedup increases more rapidly as against using more processors to run the same problem. This is attributable to load balancing. The partitioning of a given problem into different subproblems decreases the size of the subproblem as the number of partitions increases. Partitioning a set $S$, say of 1000 points into 2 subproblems may assign 500 points to each subproblem whereas a similar subdivision into say 5 partitions may yield only 200 points per partition. There is no doubt that this will eventually affect the performance and subsequently the speedup of the problem. Using more partitions may reduce the amount of work given to a processor. The work load may not be enough to keep the processors busy. On the other hand, more points will be eliminated by the leaf processors, thus simplifying the initial hulls. At subsequent stages not many points are removed because these will be vertices of sub hulls. The speedups however tend to stabilise when each processor is given adequate task to keep them busy. The graphical representations in figures B.1 – B.32 and B.34 of Appendix B illustrate the shared memory implementation for the different problems using both threads and microthreads for the recursive and stack based implementations. The speedup for 2-D problems increases quite rapidly because of the simplicity of the problem whose shapes are mainly plain polygons. For the higher dimensional problems
the speedup also increases with an increase in the problem size but the increase is gradual and steady because of the interplay of the dimension and the increased number of facets of the object. The microthreads implementation also shows a similar trend. Similar results are observed in both the recursive and stack versions but the stack version usually proves to be faster and larger problem sizes could be implemented. The most significant result is that we can measure real performance gains even for a relatively small number of points. Scaling up the results is non-trivial due to memory management problems resulting from combinatorial explosion of the point and edge data structures.

### 5.4 Message Passing Implementation

This section considers the implementation of the \( n \)-D convex hull algorithm on a distributed memory machine. Three versions of the parallel algorithms were implemented and are reported here:

- Simulated Fanin Tree.
- Tree Method.
Figure 5.4: Recursive Version 4-D 6 Vertices Using Threads

Figure 5.5: Recursive Version 3-D 12 Vertices Using Microthreads
Figure 5.6: Stack Version 2-D 26 Vertices Using Threads

Figure 5.7: Stack Version 4-D 6 Vertices Using Threads
In the Distributed Memory implementation, all the three different approaches adopted build a fanin tree structure but differ in the way in which communication and exchange of data takes place. The processors at the highest level of the tree are termed the treeleaf processors while those between the root and the treeleaves are the treenode processors.

5.4.1 Simulated Tree

In this approach a processor known as the master processor is given the initial problem to be solved. All other processors initially have nothing to do and are thus idle. The master processor starts by splitting the set of points whose convex polytope is to be computed into a predetermined number of subsets or subproblems. After this partitioning scheme, the master processor then farms out each subproblem to its neighbouring idle processors at the highest level of the tree called the slaves. Initially, the master processor transmits the \( p \) subproblems to \( p \) slave processors. The slaves accept these tasks from the master and compute the convex hull of each subproblem and in turn give back their results to the master. The master now sends two respective subconvex hull lists to each of \( p/2 \)
slaves at the next lower level of the tree for the next round of computation by reusing the
processors. The slaves at this level perform the merge process by calling the merge routine
before computing the convex hull. This approach constructs a simulated fanin tree and
this merge and compute process is repeated until in the final stage, two subconvex hull
lists are sent to one slave (root processor) by the master. The final merging and convex
hull computation takes place here producing the final solution to the problem. The result
is communicated back to the master for output. The algorithm is summarised as follows:
\[ R_1, R_2, \ldots, R_p = \text{partition}(S, d) \]

For \( i = 1 \) to \( \text{procs} \) /* send list to slave \( i */

\{
    \text{csn\_tx(masterchan, 0, toslave.id[i, &status, sizeof(status));}
    \text{Transmit\_Plist}(R_i,n, masterchan, toslave.id[i]);
\}

For \( i = 1 \) to \( \text{procs} \) /* get result from slave \( i */

\{
    \text{csn\_tx(masterchan, 0, toslave.id[i, &status, sizeof(status));}
    \text{Receive\_Plist(&CH(R_i), &m, masterchan, &fromslave.id[i]);}
\}

while(\( \text{procs} > 1 \)) /* loop 1 */

\{ /* send list to slaves */
    \text{For } i = 1 \text{ to } [\text{procs}/2] \\
    \{
        \text{csn\_tx(masterchan, 0, toslave.id[i, &status, sizeof(status));}
        \text{Transmit\_Plist(CH(R_i), n, masterchan, toslave.id[i]);}
        \text{Transmit\_Plist(CH(R_{[\text{procs}/2]}), n, masterchan, toslave.id[i]);}
    \}

    \text{For } i = 1 \text{ to } [\text{procs}/2] \text{ /* master gets results from slaves */}
    \{
        \text{csn\_tx(masterchan, 0, toslave.id[i, &status, sizeof(status));}
        \text{Receive\_Plist(&CH(R_i), &m, masterchan, &fromslave.id[i]);}
        \text{For } i = 1 \text{ to } [\text{procs}/2] \text{ do } \{ R_i = R_i \cup R_{[\text{procs}/2]+i};
        \text{\textbf{\( R_{[p/2]+i} = 0 \)}}
    \}

    \text{procs} = [\text{procs}/2];
\}
Two major processes are involved here. The first involves the master processor which handles the distribution and coordination of tasks around the network. The second is performed by the slave processors and actually does the application specific work by merging two sublists where necessary before using the sequential convex hull routine for computation. The master and the slave processors work closely to achieve the desired result. By this scheme the complexity of the slaves is minimised and the master can be kept busy with the communication task. However one of the major limitations of the method is that the programmer has to be involved with all the low level issues such as routing and message passing and a significant proportion of the development time of the parallel implementation was spent catering for these communication problems. The diagram in figure 5.9 illustrates the exchange of information and data between the master and the slave processors in a simulated tree environment using four leaf slave processors.
5.4.2 Tree Method

This implementation seeks to address the communication overhead experienced in the simulated tree approach. The master initially partitions the set into $p$ subsets. These subproblems are in turn mapped onto the $p$ treeleaf processors where each will basically use the sequential convex hull algorithm to compute its convex polytope. The algorithm for the master is paraphrased here:

$$R_1, R_2, \ldots, R_p = \text{partition}(S, d)$$

For $i = 1$ to procs /* send list to $R_i$ to leaf $i$ */

Transmit_Plist($R_i, n$, masterchan_out ,tree_id[i]);

/* send computed result back to the master */

Receive_Plist(CH($R_1$), &n, masterchan_in ,NULL);

$$CH(S) = CH(R_1)$$

Each treeleaf is a transputer which possesses its own copy of the sublist sent by the master and also runs a sequential convex hull routine. Once a leaf process has produced its convex hull it is directly transmitted to the next lower level of the tree. Here a treenode awaits the arrival of two subconvex hull lists with which to carry out a merge and consequently compute the convex hull at that node. The method which builds a tree in hardware requires $2^{p-1} + 1$ processors. The diagram in figure 5.10 illustrates the configuration of a four leaf transputer network showing how the tree is constructed. The final solution is filtered out from the root. The processes that run on each transputer are identical (except the master) apart from the fact that the size of the data used for computation at each level of the tree may be different once the computation starts. The sub hulls produced after the initial computation may differ in the number of points and hence the number of faces. After the merge process, the load distribution will depend on the number of points from the previous two sub hulls where the data were derived before the merge. The following algorithm Transputer_Hull() summarises steps performed at the treeleaf.
Figure 5.10: Tree In A Distributed Machine

Transputer_Hull()

/* get list from the master */

Receive_Plist(&Ri, &n, leaf.in, NULL);

/* compute the sub convex hull */

\[ CH(R_i) = \emptyset \]

if(IsEmpty_Plist(Ri) = FALSE)

Generate_Hull(Ri, n, &CH(Ri), &FA)

/* send result to node */

Transmit_Plist(CH(Ri), n, leaf.out, leaf.out.id);

/* shutdown */

csn_rx(leaf.in, NULL, &status, sizeof(status));
csn_tx(leaf.out, 0, leaf.out.id, &status, sizeof(status));

The algorithm Transputer_Merge() receives two subconvex hull lists, merges them and computes yet another subhull until the final list comes from the root node. Transputer_Merge() is implemented at the treenode.
Transputer_Merge()
/* get two sub convex hull lists */
Receive_Plist(&CH(~), &n1, leaf_in_1, NULL);
Receive_Plist(&CH(Rt), &n2, leaf_in_2, NULL);
if(IsEmpty_Plist(CH(Rt)) == FALSE)
{
    while(IsEmpty_Plist(CH(Rt)) == FALSE)
    {
        CH(Rt) = CH(Rt) \cup CH(Rt)
        CH(Rt) = \emptyset
    }
}  Generate_Hull(CH(Rt))
/* send result to the next lower node */
Transmit_Plist(CH(Rt), n, leaf_out, leaf_out_id);
/* shut down */
csn_rx(node_in_1, NULL, &status, sizeof(status));
csn_rx(node_in_2, NULL, &status, sizeof(status));
csn_tx(node_out, 0, node_out_id, &status, sizeof(status));

A major limitation in this approach stems from the fact that as the process moves from
one level of the tree to the next lower level, the previous processors are made redundant
making them idle. The number of partitions also gets smaller as points are filtered out
at different levels so that parallelism drops. Where the size of partition drops the fan­
in part of the tree produces overheads. This is because the number of points generally
gets smaller as the computation advances from one level of the tree to the next level.
Secondly, the communication versus the computation is not so good. The tree scheme has
been implemented using both recursive and stack versions and the timing recorded using different number of leaf processors. The results are presented in tables 29 – 38 as Version 2. The sequential program runs on only one transputer.

5.4.3 Fixed Size Tree or Pipelined Method

This third approach differs from the simulated tree and pipelined versions in the sense that the original set of points is split into $p$ partitions where $\tilde{p} > p$ the number of leaf processors. The treetleaf processors compute the convex hull from their respective sublist sent by the master and pushes the results down the next lower level of the tree. If there are more sublists in the queue whose convex hull is yet to be computed, the next batch is sent to idle leaf processors as soon as they are ready for another round of tasks. The root processor sends its list to rejoin the queue for reprocessing. This cyclic motion is terminated when the partitions are exhausted and the tree is full. The rootnode returns the final result. The code is given below:

```c
/* split S into parts - storing in an edge list */

MakeEmpty_Elist(&Parts_List);
for(i=0; i<parts; i++)
{
    MakeEmpty_Plist(&Slist);
    Parts_List = Insert_Edge(Parts_List, n, Slist, v);
}
while(IsEmpty_Plist(S) == FALSE)
{
    Read_Edge(Parts_List, n, &Slist, v);
    Read_Point(S, n, v);
    Slist = Insert_Point(Slist, n, v);
    Write_Edge(Parts_List, n, Slist, v);
    Parts_List = GetNext_Edge(Parts_List);
    S = Delete_Point(S);
}

/* fill up tree to start computation */
```

107
\[ h = \text{(int)} \left( \frac{\log_{10} \text{procs}}{\log_{10} 2} \right) + 1; \quad /* \text{height of tree -1} */ \]

status = 1;
/* height of tree -1 */
for (i=1; i<=h; i++)
{
    /* send data to leaves */
    for (j=0; j< procs; j++)
    {
        if (IsEmpty_Elist(Parts_List) == FALSE)
        {
            Read_Edge(Parts_List, n, &Slist, v);
            Transmit_Plist(Slist, n, masterchan_out, tree_id[j]);
            Parts_List = Delete_Edge(Parts_List);
            parts = parts - 1;
        }
        else
        {
            MakeEmpty_Plist(&Slist);
            Transmit_Plist(Slist, n, masterchan_out, tree_id[j]);
            parts = 0;
        }
        csn_tx(masterchan_out, 0, tree_id[j], &status, sizeof(status));
    }
}
/* process rest of parts until less than procs left */

while (parts+h > procs)
{
    Receive_Plist(&Slist, &k, masterchan_in, NULL);
    csn_rx(masterchan_in, NULL, &status, sizeof(status));
    Parts_List = Insert_Edge(Parts_List, n, Slist, v);
    parts = parts + 1;
    for (j=0; j< procs; j++)
    {
        if (IsEmpty_Elist(Parts_List) == FALSE)
        {
            Read_Edge(Parts_List, n, &Slist, v);
            Transmit_Plist(Slist, n, masterchan_out, tree_id[j]);
            Parts_List = Delete_Edge(Parts_List);
            parts = parts - 1;
        }
        else
        { /* remainder */
            break;
        }
    }
}

108
{ MakeEmpty_Plist(&Slist);
  Transmit_Plist(Slist, n, masterchan_out, tree_id[j]);
  parts = 0;
};
csn_tx(masterchan_out, 0, tree_id[j], &status, sizeof(status));
}

/* collect results still in tree */

for(i=1; i<=h; i++)
{
  Receive_Plist(&Slist, &k, masterchan_in, NULL);
  csn_rx(masterchan_in, NULL, &status, sizeof(status));
  Parts_List = Insert_Edge(Parts_List, n, Slist, v);
  parts = parts + 1;
}

/* send last proc lists */

status = 0;
for(j=0; j< procs; j++)
{
  if (IsEmpty_Elist(Parts_List) == FALSE)
  {
    Read_Edge(Parts_List, n, &Slist, v);
    Transmit_Plist(Slist, n, masterchan_out, tree_id[j]);
    Parts_List = Delete_Edge(Parts_List);
  }
  else
  {
    MakeEmpty_Plist(&Slist);
    Transmit_Plist(Slist, n, masterchan_out, tree_id[j]);
  }
  csn_tx(masterchan_out, 0, tree_id[j], &status, sizeof(status));
};
Receive_Plist(&Slist, &k, masterchan_in, NULL);
csn_rx(masterchan_in, NULL, &status, sizeof(status));
*CH = Slist; *FA = FAlist;

This implies that a fixed sized tree with \( p \) leaf processors \((2^{p-1}+1) \) processors altogether) where \( \bar{p} \gg p \) is used to pipeline partitions through the architecture in blocks of size.

109
Each pass through the tree reduces \( p \) partitions to one partition so eventually a single partition representing the final hull is produced. This technique tends towards a 100% efficiency since the processors are always busy but requires careful control and manipulation of the underlying architecture.

5.5 Results From Distributed Memory Machine

Like in the shared memory, the experiments were test run on different problems on 2-D, 3-D and 4-D. From our results, recorded in Tables 29 – 38 of Appendix A, the simulated tree implementation gives the best performance in terms of the speedup obtained despite the communication problem. This is because the processors are being reused at each level of the tree. It was observed that the stack version was faster as was the case in the shared memory implementation. The size of the problem implemented in a distributed architecture for \( d > 2 \) was quite small because of limited memory. For example the stack version in the simulated tree approach was able to run problems of size 200 points each in 3-D with 12 vertices (Table 32 of Appendix A) and 4-D with 6 vertices (Table 33 of Appendix A). This was further reduced to 100 points for 3-D and 200 points for 4-D respectively when version 2 (tree method) was recursively implemented. Even though the communication cost in the tree method (version 2) is reduced compared to the simulated fanin tree (version 1) the method appears to be expensive in terms of processor utilisation. This leads to poor efficiency which could be readily derived from the results. A fanin tree constructed from four treeleaf processors using the pipelined method will require a total of seven processors before the result is filtered out from the root of the tree (see figure 5.10) while three slave processors will need a total of five processors before the final result is sent to the master. This could be expensive in a situation where processors are expensive assets. These problems notwithstanding we have still demonstrated that reasonable speedup is obtainable with our techniques even with small problem sizes. With
available architecture where memory capacity is not a problem, scaling up the problem is trivial. The results of the simulated tree scheme are presented in tables[29 - 38]. This is represented as Version 1 in the tables. Some of the graphical presentation of these results are shown in Figures 5.11 – 5.18 while others are included in Appendix B.33, B.35 – B.40 and they also confirm that a significant improvement over the sequential algorithm is possible.

5.6 Partitioning Methods

Rabhi and Manson [92] show that for certain applications it is only necessary to generate as many subtasks as there are processors in order to obtain optimal performance. Such applications are those that divide up evenly and give rise to as many equal sized subtasks as there are processors. However, some applications divide up in an uneven or unpredictable fashion in a way that does not straightforwardly give a good load balancing of task to processors. It is not very clear when the dividing process should stop for these applications. If division does not result in a good load balancing, some processors will be starved of work. If the division is too fine grained, the processors will spend too much
Figure 5.12: Recursion Version 2 2-D 26 Vertices Using Transputer

Figure 5.13: Recursion Version 2 4-D 6 Vertices Using Transputer
Figure 5.14: Recursion Version 1 4-D 6 Vertices Using Transputer

Figure 5.15: Stack Version 1 2-D 26 Vertices Using Transputer
Figure 5.16: Stack Version 2 2-D 26 Vertices Using Transputer

Figure 5.17: Stack Version 2 4-D 6 Vertices Using Transputer
time engaged in performing the house keeping tasks rather than solving the problem at hand. Hence Rahbi and Manson demonstrate that the key issue to be resolved for a given application is that of finding the 'optimal partition' of subtasks.

Although we can choose the partition size for the convex hull arbitrarily at the outset, difficulty arises once the first merge occurs because the shape of the resulting subconvex hulls can be arbitrary. Consequently, we need to find a good partitioning method which attempts to balance the size of convex hulls at each level of the algorithm. Generally, this is not possible (because of the random distribution of points) but we can define partitions for different classes of problems. On the basis of this we have tried in many ways to partition the set $S$ into $p$ subtasks. This is an attempt to devise a partitioning strategy to control the size $n/p$. We now consider the following and most promising partitioning methods:

### 5.6.1 Lexicographic Partitioning

Here the points in $S$ are sorted lexicographically and then taken in order one at a time and allocated to partitions using wrap around. The $i$th point being assigned to partition
according to \((i + 1) \mod p\). This could be summarised as follows:

**Algorithm Lexco_Partitioning(S,n,parts)**

**Input:** A set \(S\) of points.

**Output:** Subsets \(Slist[i]\) of set \(S\), \(i=0(1)\)\(parts-1\).

\[
\{
\text{Quick Sort}(S,n) \quad /* \text{put } S \text{ into lexicographic order */}
\]

For \(i = 0\) to MAXPARTITIONS-1

\[
\{
\text{CHlist}[i] = \emptyset; \quad \text{FAlist}[i] = \emptyset;
\text{Pointcount}[i] = 0;
\}
\]

For \(i = 0\) to \(parts-1\)

\[
\text{Slist}[i] = \emptyset
\]

\(i = 0;\)

while\((S \neq \emptyset)\)

\[
\{
\text{Take the next point from } S
\text{Add point to Slist}[i]
\text{Pointcount}[i] = \text{Pointcount}[i] + 1
\text{i} = (i + 1)\%\text{parts};
\}
\]

This method does not attempt to check if partitions are disjoint but guarantees almost perfect load balancing by spreading the points evenly across partitions initially. We can illustrate the lexicographic partitioning by considering a quadrilateral in 2-D with sixteen points as shown in figure 5.19. Suppose the points are ordered and split into three
partitions \((p = 3)\), the shapes labelled A, B, and C will be generated from the points in the three partitions. Shape A is from the first partition with the points labelled 1, shape B from the second partition from points labelled 2 and shape C from the third partition from points labelled 3. From the diagram, the shape labelled A has four vertices on the convex hull and also has four faces. B has five vertices on the convex hull and five faces while C has four vertices and four faces. In section 5.1 the running time of the sequential algorithm is a function of the problem size, the faces and the dimension space. Since the shape generated from each partition is different, their running time also varies and in the next level of the tree where a merge and compute process is carried out perfect load balancing is no longer guaranteed. The convex hull when the points are split into two partitions \((p = 2)\) will yield the shapes labelled D and E. In this case the rectangles have an equal number of faces and vertices and a perfect load balance is possible. If the partition sizes match the size of data we may also get disjoint partitions as shown in shapes G, H, I and J, resulting in a perfect load balance. However, these

Figure 5.19: Point Allocation In Lex Partitioning
shapes will escape through simplex bypass and there is relatively little work to occupy
the processors. In the case of an even number of partitions, the load is balanced among
processors as data moves up the tree but most of the computations are carried out in the
leaf processors. Notice that in the case of three partitions, the vertices of the convex hull
lie in the partition with the points labelled 1 while other partitions produce vertices that
do not form part of the convex hull. Except in shapes G, H, I and J the method does not
guarantee disjoint sets. A, B, and C form intersecting domains and so do D and E.

The Lexicographic partitioning method uses the quick sort algorithm to sort the points
which is of $O(n^2)$ in the worst case and with average speed $O(n \log_2 n)$ and requires $n$
operations to partition the points into the subproblems.

5.6.2 Random Colouring

This is similar to the lexicographic scheme except that each point is given a random
number (colour) from 1 to $p$ determining its partition. This scheme is based on the idea
that a random distribution of colours should produce shapes of roughly equal number of
vertices and faces. Points with similar colours are grouped under the same partition. The
method requires $n$ operations to partition the set.

Algorithm RndColour_Partitioning(S, n, parts)

Input: A set $S$ of points.

Output: Subsets $Slist[i]$ of set $S$, $i=0(1)\text{parts-1}$.

{  
For $i = 0$ to MAXPARTITIONS-1  
{  
CHlist[i] = \emptyset; FList[i] = \emptyset;
}
For i = 0 to parts-1
{
    Slist[i] = \emptyset
    Pointcount[i] = 0;
}
i = 0; srand(1);
while(S \neq \emptyset) {
    Take the next point from S
    i = srand() \% parts
    Add point to Slist[i]
    Pointcount = Pointcount[i] + 1
}

This method does not even guarantee the same load to all processors. If we consider dividing nine points in the 2-D plane into say three partitions using the random colouring method, a possible distribution may result in a situation shown in figure 5.20. The shape
A is for points labelled 2, B for points labelled 0 while C is for those labelled 1. The colours are randomly assigned as each point is considered. As we have seen in the lexicographic partitioning, the complexity depends on the subconvex hulls. The Random Colouring method aims at producing subconvex hulls with equal complexity but unfortunately this has not been achieved though the initial partitions may provide a reasonable load to each processor. In figure 5.20, to generate A and B may yield the same complexity if the problem sizes that gave rise to them were the same. The complexity to produce C is quite different from that of A and B. The lexicographic and random colouring schemes have very low overhead for partitioning compared to the next three methods. The rest of the methods are also computationally more complex but do better in identifying clusters of points.

An obvious way to do partitioning in 2-D is to use a number of bands as indicated in figure 5.21 and use the \((x, y)\) position as an indication of the band. This require the checking of the lower and upper bounds to determine the partition and also requires the length \(r\) to decide on the band positioning. The advantage of this method is that the hull in each partition is distinct but there may be problems of load balancing and the
number of vertices on the sub hull shapes may be different depending on how the points are distributed. In 3-D the partitions become cubical in shape and requires the checking of six halfspaces and hence the method does not extend well. An alternative to this method is therefore the Bucket approach.

5.6.3 Bucket Method

The reason for using bucket partitioning comes from the fact that points can cluster into different regions. To partition the points using the methods discussed above may not give an even spread or distribution among the different sub problems.

In this approach, we determine a point \( c \) interior to \( S \) and use it as an origin to partition the \( n \)-D space into \( 2^n \) disjoint subspaces or buckets. Points are allocated to the buckets according to their position relative to \( c \). The method also guarantees disjoint partitions but not an even load balance and complications arise if \( p \neq 2^n \) which is often the case. Figure 5.22 shows a 2-D plane being partitioned into four buckets. The set of points in each quadrant belongs to a bucket i.e. 8 points, 3 points, 2 points and 3 points. As can be seen from the diagram, a perfect load balancing is not guaranteed. There is a concentration of points in the first quadrant compared to the others. In order to overcome the clustering of points in some regions, the quadrant with more points can be repartitioned recursively until almost a perfect load balance is achieved as shown in figure 5.22. We have not considered this repartitioning method in this research because the overhead in computing the partitions make it prohibitive.

5.6.4 Shell Method

The first step involves the ordering of the points. This requires \( O(n^2) \) worst case. In this scheme we determine a point \( c \) interior to \( S \) (preferably the centroid or alternatively the average of the maximum or minimum coordinates). This can be done in \( cn \) steps. The longest euclidean distance \( r \) between \( c \) and points \( x \in S \) is calculated and shell \( i \)
is determined according to the bounds \((i - 1)r/p\) and \(ir/p\). This will require \(c_2n\) steps.

Points are allocated to partitions according to the shell they inhabit and this can be accomplished in \(c_3n\) steps. The method essentially computes a set of concentric circles in 2-D, spheres in 3-D, and their extensions for \(n\)-D. In the simple approach the radius of two adjacent shells differs by a constant but the volume increases with distance from \(c\) so load balancing is not guaranteed for a uniform distribution of points but we know that the hull of partitions are derived from non overlapping sets and may contain some nesting within each other. The shapes may be roughly spherical and so of roughly the same complexity which is the essence of this implementation. The subconvex hulls generated from each partition are non intersecting. In figure 5.23 we show how points could be partitioned into different shells. Notice that Bands[4] contains the convex hull and that each band is a subhull. The different domains are labelled A, B, C and D corresponding to Bands[1], Bands[2], Bands[3] and Bands[4]. The different bands will be assigned to the leaf processors to compute the convex hull. There may exist situations where the points are clustered on one side in which the convex hull may not fall into one band. This is illustrated in figure 5.23 where the convex hull falls in more than one shell. The
shapes with labels E, F, G and H are the domains from different shells. The convex hull is represented as I and it cuts across Bands[2], Bands[3] and Bands[4].
Algorithm Shell_Partitioning(S,n,parts)

Input: A set $S$ of points.

Output: Subsets $Slist[i]$ of set $S$, $i=0(1)parts-1$.

{
    Quick_Sort(S,n) /* put $S$ into lexicographic order */
    Get two extreme points $v$ and $w$. /* Find center of polytope */
    $c = (v + w)/2$;
    /* Find the longest distance between centre and any point $x \in S$ */
    max = 0;
    while($S \neq \emptyset$)
    {
        Read point $x$
        $t = \sqrt{v^2 + w^2}$
        if(max < $t$) max = $t$;
    }
    $r = \sqrt{\text{max}}/\text{parts}$;
    Bands[0] = 0;
    For $i = 1$ to $\text{parts}$-1
    {
        Bands[$i$] = Bands[$i-1$] + $r$;
    }
    For $i = 0$ to $\text{MAXPARTITIONS}$-1
    {
        $\text{CHlist}[i] = \emptyset$; $\text{FAlister}[i] = \emptyset$;
    }
    For $i = 0$ to $\text{parts}$-1
    {

Figure 5.24: Allocation Of Points To Shells With Convex Hull Across Bands
Slist[i] = ∅
Pcount = 0;
}
/* Partition S according to distance Bands[i-1] <= r < Bands[i] inserted into Slist[i-1] */
while(S ≠ ∅)
{
    Get a point x
    r = x
    i = 1;
    while(r - Bands[i] > 0 and i < parts)
    i = i + 1;
    i = i - 1;
    Add point x to Slist[i-1];
Pcount = Pcount[i] + 1;
}

5.6.5 New Shell Partitioning

This is an improvement on the shell partitioning method. Rather than stepping through a constant increase in the radius of each consecutive shell, shells with equal volume are computed using the mathematical formula for an n-dimensional sphere given by

\[ Vol = \frac{\sqrt{\pi^n}}{\Gamma\left(\frac{n}{2} + 1\right)} R^n \]

where \( n \) is the space dimension, \( R \) the radius of sphere and \( \Gamma \) is the gamma function. The total volume can now be partitioned into shells of equal volumes and the points allocated according to the shell in which they belong. If the points are uniformly distributed, the load balancing will be improved. An advantage of this scheme is that the shells get thinner as they move away from the centre. The implication here is that for a large number of points that are evenly spread, the thinner shells will virtually be convex hulls but the problem is that we will get more vertices and faces on each of the sub hulls.

5.6.6 Multiple Level Partitions

In all the partitioning methods that we have considered, the possibility of merging the results after each level of computation and repartitioning could help to achieve a better load
distribution at each level of the computation but this will tend to increase the computing time as some of the techniques that we have proposed are quite complicated. Merging the partial results and repartitioning requires \( nl \) steps where \( l \) is the number of levels in the tree.

### 5.7 Results From Partitioning Methods

Table 39 of Appendix A shows how a set \( S \) with a total of 1000 points in 2-D and 4-D are distributed into 6 partitions using the different partitioning methods. A trial experiment was carried out on the Encore Multimax using the Recursive algorithm to test the performance of the different partitioning methods. This was carried out on a 2-D problem with 26 vertices on the convex hull and 4-D problem with 6 vertices. In both cases a set \( S \) with a total of 1000 points was considered. Table 5.1 shows the speedup obtained when the partitions in Table 39 of Appendix A were implemented. From these results, the simplest scheme (Lex) appears to be the best for small point sets and partitions. This is because the other methods require a proportionally large computing time and the standard deviation from the mean partition size is larger than that for Lex which is always close to optimal. Increasing the number of partitions generally improves the speedup and this is reflected in Table 5.1 until the size of partitions is very small. For large set of points

---

**Table 5.1: Partitioning (2-D 26 vertices, 4-D 6 vertices, with 1000 points) On Multimax**

<table>
<thead>
<tr>
<th>Methods</th>
<th>Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2D</td>
</tr>
<tr>
<td>Lex (sp)</td>
<td>1.89</td>
</tr>
<tr>
<td>Rand (sp)</td>
<td>1.61</td>
</tr>
<tr>
<td>Shell (sp)</td>
<td>1.74</td>
</tr>
<tr>
<td>New_Shell (sp)</td>
<td>1.05</td>
</tr>
<tr>
<td>Bucket (sp)</td>
<td>1.41</td>
</tr>
</tbody>
</table>
Table 5.2: Statistics For Partitioning Methods From Table 39

<table>
<thead>
<tr>
<th>Methods</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2D</td>
<td>4D</td>
<td>2D</td>
<td>4D</td>
<td>2D</td>
</tr>
<tr>
<td>Lex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>0.0</td>
<td>0.0</td>
<td>0.58</td>
<td>0.58</td>
<td>0.0</td>
</tr>
<tr>
<td>X</td>
<td>500</td>
<td>500</td>
<td>333.3</td>
<td>333.3</td>
<td>250</td>
</tr>
<tr>
<td>Random</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>0.0</td>
<td>0.0</td>
<td>14.7</td>
<td>14.7</td>
<td>0.0</td>
</tr>
<tr>
<td>X</td>
<td>500</td>
<td>500</td>
<td>333.3</td>
<td>333.3</td>
<td>250</td>
</tr>
<tr>
<td>Shell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>58</td>
<td>147.1</td>
<td>221.5</td>
<td>141.5</td>
<td>146.8</td>
</tr>
<tr>
<td>X</td>
<td>500</td>
<td>500</td>
<td>333.3</td>
<td>333.3</td>
<td>250</td>
</tr>
<tr>
<td>New_Shell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>473.8</td>
<td>688.7</td>
<td>261.0</td>
<td>532.7</td>
<td>196.9</td>
</tr>
<tr>
<td>X</td>
<td>500</td>
<td>500</td>
<td>333.3</td>
<td>333.3</td>
<td>250</td>
</tr>
<tr>
<td>Bucket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>158.4</td>
<td>418.6</td>
<td>243</td>
<td>407.5</td>
<td>68.6</td>
</tr>
<tr>
<td>X</td>
<td>500</td>
<td>500</td>
<td>333.3</td>
<td>333.3</td>
<td>250</td>
</tr>
</tbody>
</table>

results indicate that the shell method based on volume reduces the loading deviation more rapidly than the new shell. Table 5.2 shows the standard deviation and the mean for table 39. For Lex and Random colouring partitioning, the deviation is small compared to other methods. This also illustrates why Lex is the best because of good load balancing.
Chapter 6
Facial Lattice Exploration (FLE)

In the previous chapter, we proposed a parallel implementation of the n-D convex hull algorithm based on the divide-and-conquer technique. In this chapter a different approach based on exploring the facial lattice of the convex hull is adopted. The facial lattice of a polytope $P$ is a lattice which represents the polytope facial structure. Each node in the lattice is a face of the polytope; there is an edge from $F$ to $G$ if and only if $F$ is a facet of $G$. Alternatively, it is the lattice given by the set of faces of $P$ and the subset relations which are edges and vertices. In figure 6.1, we show the facial lattice of a pyramid over a square, the element $I$ represents the entire polytope, while $O$ represents the empty set. The motivation for the partitioning method in the previous chapter is that large numbers of points are eliminated quickly. Unfortunately, the merging process required to combine the partitions cannot control the load balancing and hence results in potential loss of performance. The FLE will avoid the partitioning of the set of points $S$ into subproblems. This FLE technique seeks to find the convex hull of the set of points $S$ by ‘wrapping’ around the edges of the hull. After the determination of the initial facet, the sequential algorithm picks each of the edges in turn and performs rotations to produce more edges until all the edges are computed twice. The strategy used in [1] and [2] is to maintain a list of edges with adjacent nodes in the lattice but whose other adjacent node has not yet been computed. In each step, the sequential algorithm picks an edge from the
edge list and performs rotations to find the edge’s other adjacent node. For every edge, the algorithm checks to see if it is present in the edge list, if so it is deleted from further consideration because it is now computed twice, else it is added to the edge list. The difference between this implementation and that of the previous chapter is that there is no partitioning of the points into subsets. The main aim of this chapter therefore is to demonstrate that parallelism in the $n$-D convex hull problem can also be exploited along the edges once the initial face has been computed.

Our technique, which we term Facial Lattice Exploration (FLE), computes the faces simultaneously by picking more than one edge from an EdgeList. Figure 6.2 shows how the convex hull of a 3-D cube can be found in parallel using FLE. For simplicity the shape is flattened onto a plane surface to expose all the faces. First, the initial face labelled 1 is computed. The four edges, ABCD are now available and by assigning each edge to an idle processor the other faces labelled 2 can be computed in parallel. After the computation of the initial face, a processor picks an edge say A from a queue, leaving BCD for other processors that are idle. If all the remaining edges are allocated to processors, all the faces labelled 2 can be computed in parallel. Thus the edges $E - P$ are produced, greatly
increasing the possible parallelism. In particular if we choose edge E the final face labelled 3 can be found. Thus in principle (i.e. with enough processors) the complete hull can be found in just three steps of the sequential method. However, there are a number of problems with this approach, for example

- The original sequential algorithm eliminates edges once they have been determined twice. In this scheme the edges LM JI GF OP are actually the same edge and should be eliminated from further consideration. However, they may reside in different processors implying some overheads in communication and co-ordination of edgelists.

- A further problem arises when we consider the edges K, H, N and E. Potentially, each of the edges could be grabbed by a processor simultaneously producing face 3 four times. These four copies when computed in parallel contribute no additional costs except for potential clashes for accessing global list data but when staggered can contribute considerable cost (e.g. if not enough processors are available).

- The second point raises a more serious problem, that of termination. The sequential
algorithm relies heavily on the fact that an edge can only be found twice. Indeed this assertion is used to control the edge list so that eventually the list will be empty and the algorithm can terminate. It might appear in figure 6.2 that this rule is not violated. In fact the correctness depends on the non-deterministic order of face evaluation. For example if we only have two processors, after finding the edges ABCD we could choose B and D producing edge list LKJPEFAC. Next choose C and E this produces the additional edges MNO and QRS. Clearly LM, OP, QN are duplicates and should be deleted. Given the list KJFARS it is now possible to choose edge K in one processor and proceed to generate LKJB, K and J are duplicates but L and B have already been found but deleted from the list and will be reinserted. Potentially the algorithm may never terminate thus some mechanism of global list management has to be devised.

The FLE approach is well suited to applications where the data consists only of points on the hull (e.g. generating loop nests in parallel compilers). In these cases the partitioning method cannot exploit the divide-and-conquer principle and delivers poor performance. The FLE method avoids partitioning and the potential irregular loading of processors during merging. There are also some problems inherent in the lattice approach. A perfect load balance cannot be guaranteed because of different sized facets and the fact that the number of facets limits parallelism. The best we can do for a 2-D problem is a factor of 2 speed-up irrespective of the number of processors that we use. This is because the maximum number of edges available after the computation of the initial face cannot exceed 2. If we consider figure 6.3 and suppose that facet A is computed first, we can then compute the faces labelled B and C in parallel. The next step will be to find D and E also in parallel before the process will terminate with the determination of face F. Obviously, we have seen that the possible parallelism depends on the number of facets. A similar argument holds for the cube (3-D) as already explained using figure 6.2. Here a single
facet generates three new edges and if the faces have the same structure we can see that
the maximum parallelism is related to 3 (e.g. 3, $3^2$, $3^3$ etc.) but for each shape we cannot
predict the structure otherwise we would already know the convex hull or a large degree of
it. In general, if we consider a hypothetical situation, where we have $m$ edges per face, this
will produce $m$ other faces with each face giving rise to $m - 1$ other edges. This will result
in a total of $m^2 - m = m(m - 1)$ edges in the second level. Similarly, the third level of the
tree will yield $m(m - 1)(m - 1)$ edges. This trend is easily extended to subsequent levels.

Though the FLE implementation is not considered as a tree structure, the representation
in Figure 6.4 illustrates the components of the facial lattice structure and the possible
parallelism. Assuming we have an unlimited number of processors, then we can exploit
the inherent parallelism by assigning each edge to a processor and then computing the
faces in parallel. To explore the parallelism, it is necessary to break up the tree pattern.

The representation of figure 6.4 shows how to enumerate the faces and edges of the facial
lattice structure as a tree. At some level of the tree there are duplicate edges which
are connections between tree levels that produce the facial lattice structure. However,
since in practice there may be duplicate edges and usually the number of processors is

Figure 6.3: Edge Computation In A 2-D Problem
limited, the option of implementing the algorithm by starting with some initial edges seems practicable. For a limited number of processors we can consider the computation of nodes as a wavefront that moves down the tree, the parallelism evolving irregularly. Such problems have not been seriously addressed in the literature in parallel processing. In our implementation we confine the facial lattice exploration to face level and not the sub-facet levels in order to simplify the design.

The different implementations which we consider in this chapter on the shared memory and message passing architecture are enumerated here:

- Version 1 a shared memory implementation, which uses the pending edge list to store the edges that have been computed twice.

- Version 2 a shared memory implementation, which uses the global list structure to store the different lists for easy access by the processors.

- Version 1 a transputer implementation, where master and manager processes run on separate processors.
• Version 2 a transputer implementation, where the master and manager processes run on the same processor.

6.1 FLE On Shared Memory (Version 1)

In the shared memory architecture we have implemented two different versions each using a different method to organise the data. Both methods make use of the master/slave organization, and can be summarised as follows:

• FLE on shared memory (Version 1) using the pending edge list to terminate the iteration.

• FLE on shared memory (Version 2) using the global list organization.

The sequential stack implementation (chapter 4) of the n-D convex hull algorithm is the underlying concept in the facial lattice exploration technique. The stack version makes it possible easily to access the sublists at different levels of the implementation. The master processor starts the computation by finding the initial face. The face computed by the master has to be unstacked keeping the vertices in the CHlist, the facets in FAlist and the edges defining the face in the Elist at the lowest level of the stack. The pending edge list will contain the edges that have been found twice. The major function of the pending edge list is to terminate the algorithm when all the edges have been computed twice. Once an edge has been found twice it is deleted from further consideration since it is the intersection of two adjacent faces and is placed in the pending edge list (Pend_Elist).

With the available edges in the Elist, the master now distributes work to the idle slaves. This is accomplished by selecting an edge and giving it to an idle slave processor. The slave accepts the given edge, rotates it and copies all the points on that face and then computes the new edges and vertices for that face by using the sequential algorithm. On completion, the slaves have to return their results back to the master to update the appropriate list in
the stack. The master also checks the Pend~list to ensure that the edge it is considering at that particular moment has not yet been computed twice. When the master has finished distributing the jobs to the slaves, it also picks an edge and computes its own edges and vertices. At the completion of each task, the processors check the Elist to ascertain whether the edges are exhausted and the slaves have completed their task. If there are more edges, one is picked for the next round of computation, otherwise the algorithm terminates or waits for a slave to return a result. In the routine Convex_Hull_Slave(), semaphores are used to protect the critical sections of the computation. This consists of the jobs assigned to the slaves as well as the results of their computation. In the master processor, the semaphores are used to prevent interference during the assignment of jobs to processors and also in copying the results from the slaves and resetting their status. Table 6.1 and Table 6.2 show how the Pending edge list grows as the computation proceeds. The 3-D problem in figure 6.2 is used for the illustration. The following two routines Convex_Hull_Master() and Convex_Hull_Slave() summarise the computation by the master and the slave processors.

Function Convex_Hull_Master(S,AS,n,k,parts) : (CH,FA);
{
   /* setup the stack */
   sp and lsp   /* stack top and local stack index */
   sp = 0; Stack[sp].S = S; Stack[sp].AS = AS; Stack[sp].k = k;
   Stack[sp].e = 0;   /* number of edges being computed by with slaves */
   Stack[sp].Pend_Elist = \emptyset;   /* Edges found twice */
   Stack[sp].Elist = \emptyset;   Stack[sp].FA = \emptyset;   Stack[sp].CH = \emptyset;
   count = 0;   /* Slave counter */

   do{
      if(Stack[sp].CH == \emptyset and Stack[sp].FA == \emptyset)
      {
         if(k == 1)   /* a 1 - dimensional set */
         {AS = \{p_0, p_1\};

135
\[
\begin{align*}
\text{min} &= p \in S \text{ such that } p_0p_1p_0p_1 \text{ is minimised} \\
\text{max} &= p \in S \text{ such that } p_0p_1p_0p_1 \text{ is maximised} \\
\text{return } \{\text{max, min}\}, \{\text{max}\}, \{\text{min}\}; \\
\}
\end{align*}
\]

if (\(|S| \geq k + 1\))
    \text{return } (S, \{F \subseteq S : |F| = k\}); /* check for a simplex */
else

\{
(F, \hat{n}) = \text{initial facet}(Stack[sp].S, Stack[sp].AS, n, Stack[sp].k) \\
Stack[sp].FA = Stack[sp].FA \cup \{ F \} \\
F'' = \emptyset; \\
\text{Pick a point } p_0 \in F; \\
F'' = \{ p \in S : p_0p_1p_0p_1 \hat{n} = 0 \}; \\
sp = sp + 1; Stack[sp].S = F''; Stack[sp].AS = F; /* stack the face */ \\
Stack[sp].k = Stack[sp-1].k - 1; stack[sp].e = 0; \\
Stack[sp].Elist = \emptyset; Stack[sp].CH = \emptyset; Stack[sp].FA = \emptyset; \\
Stack[sp].Pend.Elist = \emptyset; \\
\}
/* unstack the face completed by master */
if (Stack[sp].CH \neq \emptyset \text{ and Stack[sp].FA \neq \emptyset})
\text{while}(Stack[sp].Elist = \emptyset \text{ and Stack[sp].e == 0 and } sp > 0)

\{
\text{while}(Stack[sp].CH \neq \emptyset)

\{ /* add new vertices found */
\text{pick a point } p; \\
\text{if } (p \notin Stack[sp-1].CH)
\text{Stack[sp-1].CH = Stack[sp-1].CH } \cup \{ p \}; \\
\}
\text{while}(Stack[sp].FA \neq \emptyset)

\{ /* add new edges found using norm of complete face */
\text{pick an Edge } E; \\
\text{if } (E \notin Stack[sp-1].Pend.Elist)

\{
\text{if } (E \in Stack[sp-1].Elist}
\text{remove } E \text{ from Stack[sp-1].Elist} \\
\text{and add to Stack[sp-1].Pend.Elist} \\
\text{else}
\text{Stack[sp-1].Elist = Stack[sp-1].Elist } \cup \{ E \}
\}
/* Unstack the face */
\text{Delete lists: Stack[sp].Pend.Elist; Stack[sp].S; Stack[sp].AS; Stack[sp].k = 0; Stack[sp].e = 0;}
sp = sp-1;
if(Stack[sp].e > 0) Stack[sp].e = Stack[sp].e - 1
}
For i = 1 to procs-1 /*add faces produced by the slaves */
{
    /* Copy results from slaves to master */
    count = (count + 1)%parts;
    THREADpsem(Slavesem[count]);
    status = Slaves[count].status;
    lsp = Slaves[count].sp;
    TCH = Slaves[count].CH;
    TFA = Slaves[count].FA;
    TS = Slaves[count].S;
    TAS = Slaves[count].AS;
    Q = Slaves[count].E;
    for j = 1 to n
        norm[j] = Slaves[count].norm[j];
    THREADvsem(Slavesem[count]);
    if(status == RESULT) break;
    /* Update master stack with edges and faces from slaves */
    if(status == RESULT);
    {
        while(TCH \neq \emptyset)
        {
            /* add new vertices found */
            pick a point p;
            if (p \notin Stack[lsp].CH)
            Stack[lsp].CH = Stack[lsp].CH \cup \{ p \};
        }
        while(TFA \neq \emptyset)
        {
            /* add new edges found using norm of complete face */
            pick an Edge E;
            if (E \notin Stack[lsp].Pend.Elist
            {
                if (E \in Stack[lsp].Elist
                    add to Stack[lsp].Pend.Elist }
            else
                add to Stack[lsp].Elist }
            }
        Stack[lsp].FA = Stack[lsp].FA \cup \{ Q \}
    /* Reset the slaves for more work */
    Stack[lsp].e = Stack[lsp].e - 1;
    THREADpsem(Slavesem[count]);
    Slaves[count].k = 0;
    Slaves[count].S = \emptyset;  Slaves[count].AS = \emptyset;
    Slaves[count].E = \emptyset;
    for j = 1 to n
Slaves[count].norm[j] = 0;
Slaves[count].sp = -1;  Slaves[count].status = START;
THREADvsem(Slavesem[count]);
}
if (Stack[sp].Elist ≠ ∅) /* get next face for the master */
{
Pick an Edge E:
(p,\hat{n}) = rotate(S, AS, n, k, E, \hat{n});
F = F ∪ { p };
Stack[sp].Pend.Elist = Stack[sp].Pend.Elist ∪ { E }
Stack[sp].FA = Stack[sp].FA ∪ { F }
Pick a point \( p_0 \in F \)
\( F' = \{ p \in S : p_0 \hat{p}_1 . \hat{n} = 0 \} \);
/* stack new face */
Stack[sp].e = Stack[sp].e + 1;
sp = sp + 1;
Stack[sp].S = F'; Stack[sp].AS = F; Stack[sp-1].k = k-1;
Stack[sp].e = 0; Stack[sp].Pend.Elist = ∅;
Stack[sp].Elist = ∅; Stack[sp].FA = ∅; Stack[sp].CH = ∅;
}
For i = 1 to procs
{ /* allocate work to slaves using local stack index lsp */
count = (count+1)%procs;
THREADpsem(Slavesem[count]);
status = Slaves[count].status;
THREADvsem(Slavesem[count]);
if( status == START) break;
}
if( status == START)
{ /* slave count is currently idle so find some work in the stack */
lsp = 0;
For i = 1 to sp /* direction of search */
{
if( Stack[i].Elist ≠ ∅)
lsp = i; break;
} if (Stack[lsp].Elist ≠ ∅)
{
Pick an Edge E;
Stack[lsp].Pend.Elist = Stack[lsp].Pend.Elist ∪ { E }
Stack[lsp].e = Stack[lsp].e + 1;
/* Set up slaves to do the work */
THREADpsem(Slavesem[count]);
Slaves[count].S = Stack[lsp].S;
Slaves[count].sp = lsp;
Slaves[count].k = Stack[lsp].k;
Slaves[count].CH = ∅;
Slaves[count].FA = ∅;
Slaves[count].AS = Stack[lsp].AS;
Slaves[count].E = E;
for j = 1 to n
    norm[j] = Slaves[count].norm[j];
Slaves[count].status = lsp;
THREADvsem(Slavesem[count]);
}
}
}while(Stack[sp].Elist ≠ ∅ or sp > 0 or Stack[sp].e > 0)

return (Stack[0].CH, Stack[0].FA);
}

Function Convex_Hull_Slave(p);

{ /* grab current job for processor p */
THREAD^psem(Slavesem[p]);
status = Slaves[p].status;
S = Slaves[p].S;
AS = Slaves[p].AS;
F = Slaves[p].E;
TS = Slaves[p].S;
k = Slaves[p].k;
n = Slaves[p].n;
for i = 1 to n
    norm[i] = Slaves[p].norm[i];
THREADvsem(Slavesem[p]);
while(status ≠ STOP)
{
    if(status > START and status < RESULT)
    {
        if(S ≠ ∅ )
            (p, n) = rotate(S, AS, n, k, F, n);
        F = F ∪ {p};
        F' = ∅;
        Pick a point p_0 ∈ F;
        F' = \{ p ∈ S : p_0 p . n = 0 \};
        CH = ∅; FA = ∅;
        Convex_Hull(F', F, n,k-1) : (CH,FA);
        /* signal result is valid */
        THREAD^psem(Slavesem[p]);
        Slaves[p].status = RESULT;
        Slaves[p].CH = CH;
    }
}
}
Slaves[p].FA = FA;
Slaves[p].E = F;
for i = 1 to n
    Slaves[p].norm[i] = norm[i];
THREADvsem(Slavesem[p]);
}

/* grab current job for processor p */
THREADpsem(Slavesem[p]);
status = Slaves[p].status;
S = Slaves[p].S;
AS = Slaves[p].AS;
F = Slaves[p].E;
TS = Slaves[p].S;
k = Slaves[p].k;
n = Slaves[p].n;
for i = 1 to n
    norm[i] = Slaves[p].norm[i];
THREADvsem(Slavesem[p]);
}

6.1.1 Results

The test data was generated using the Type 1 routine of chapter 4 shown in the appendix. It is more difficult to collect data to test the algorithms in this chapter as the type of data used in testing the algorithms in chapter five did not give a promising result. This difficulty arises because the data should be such that a reasonable number of faces must be produced as an output. Such a data set will also consist of a reasonable number of edges that will be picked by different processors to exploit the inherent parallelism but such data leads to a combinatorial explosion in the work and to memory problems. The preliminary results using Type 1 data is shown in Table 6.3. The performance obtained seems poor and can be explained as follows:

- The master is dominating the computation as the slaves 'steal' their tasks from the master only when an edge is made available to them from the Elist which is kept
<table>
<thead>
<tr>
<th>List</th>
<th>Master</th>
<th>Slave 1</th>
<th>Slave 2</th>
<th>Slave 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edgelist</td>
<td>ABCD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pend.List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>(v_2v_{13}v_{10}v_8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>BCD</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pend.List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>(v_2v_5v_{10}v_{13})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>CD</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Pend.List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>(v_2v_5v_{10}v_{13})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Pend.List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>(v_2v_5v_{10}v_{13})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>IGHGPEF</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Pend.List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIHG)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>(v_2v_5v_{10}v_{13}v_6v_9v_3v_4)</td>
<td>(v_2v_{14}v_{13})</td>
<td>(v_{13}v_{12}v_{11}v_{10})</td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>IHPEJKL</td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>Pend.List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIHG)(BJKL)</td>
<td></td>
<td></td>
<td>(CMNO)</td>
</tr>
<tr>
<td>Vertices</td>
<td>(v_2v_5v_{10}v_{13}v_6v_9v_3v_4v_1v_{14})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6.2: Movement Of Edges Into Pend_List (Table 6.1 Cont.)

<table>
<thead>
<tr>
<th>List</th>
<th>Master</th>
<th>Slave 1</th>
<th>Slave 2</th>
<th>Slave 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edgelist</td>
<td>HPEKLMNO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pend_List</td>
<td>ABCDGJM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>$v_2v_5v_{10}v_{13}v_6v_9v_3v_4v_1v_{14}v_{12}v_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>EKN</td>
<td>H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pend_List</td>
<td>ABCDGJMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>$v_2v_5v_{10}v_{13}v_6v_9v_4v_1v_{12}v_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>KN</td>
<td>H</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Pend_List</td>
<td>ABCDGJMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)</td>
<td>EKNH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>$v_2v_5v_{10}v_{13}v_6v_9v_4v_1v_{12}v_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>NEKNH</td>
<td>E</td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>Pend_List</td>
<td>ABCDGJMPN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)(EKNH)</td>
<td>EPDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>$v_2v_5v_{10}v_{13}v_6v_9v_4v_1v_{12}v_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>EKHEPDF</td>
<td>K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pend_List</td>
<td>ABCDGJMPNEH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)(EKNH)</td>
<td>KJBL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>$v_2v_5v_{10}v_{13}v_6v_9v_4v_1v_{12}v_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edgelist</td>
<td>KFKJBL</td>
<td>H</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Pend_List</td>
<td>ABCDGJMPNEKH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faces</td>
<td>(ABCD)(DFEP)(AIGH)(BJKL)(CMNO)(ENKH)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>$v_2v_5v_{10}v_{13}v_6v_9v_4v_1v_{12}v_{11}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Sequential</th>
<th>Procs</th>
<th>Parallel</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>19039752</td>
<td>2</td>
<td>18558925</td>
<td>4D 6 vertices</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>17933964</td>
<td>Using 1000 points</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>18911185</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>18722954</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>18811114</td>
<td></td>
</tr>
<tr>
<td>24953393</td>
<td>2</td>
<td>20752284</td>
<td>3D 29 vertices</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>19728550</td>
<td>Using 1000 points</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>19115759</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>18871582</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>18993580</td>
<td></td>
</tr>
<tr>
<td>16168211</td>
<td>2</td>
<td>14596503</td>
<td>2D 26 vertices</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14756803</td>
<td>Using 1000 points</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>14766466</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>15038233</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>15154157</td>
<td></td>
</tr>
</tbody>
</table>

by the master.

- Secondly, the results computed by the slaves are not returned immediately as the master may still be busy computing its own face while the slaves are waiting to hand in their results.

The manipulation of the pending edge list is also a major cost of the algorithm. Every time a new edge is computed, the master has to search through the pending edge list to determine whether it has already been computed twice. In a situation where there are many edges in the pending edge list, searching through the list can take a large amount of computing time. Indeed this algorithm is proposed for complex shapes where many faces and edges exist. In order to exploit the parallelism for a list with many edges the search is likely to be a significant overhead. Also, the performance can be explained in terms of the distribution of points. With the Type 1 routine, a lot of points generated are interior points with relatively few points on the facets. It is the points on the facet that are used to determine the vertices and edges of the convex hull. The algorithms here are designed
for compute bound problems. These problems seem to have some negative influence on
the expected results. Type 2 data were therefore used to test the algorithms in the next
section.

6.2 FLE On Shared Memory (Version 2)

Considering the fact that our algorithm was designed for shapes with numerous edges and
also the likely setback caused by the pending edge list, we propose a major modification
to our algorithm and the test data generator leading to Types 2 and 3. These changes
will be discussed in this section. The possible improvements are:

• Free the master from computing a face to avoid starving the slaves of work. In the
  previous implementation, both the master and the slaves pick an edge to compute
  the subfacet whenever they are idle and the edge list is not empty. If a slave finishes
  computing a subfacet while the master is still busy, the slave has to wait for the
  master to finish its task before handing in the result. On the other hand, if a slave
  has a complex face to compute, longer delays may occur. The problem with the
  new approach is that the best speedup is between $p - 1$ and $p$ since the master is
  now only acting as a coordinator between the processors and may be idle most of
  the time.

• Using shared memory to store the global lists. The following lists could be stored
globally:

1. GEDGES which stores the edges computed by the slaves.

2. GNORMS which stores the normal of the already computed faces.

3. GCH which stores the vertices of the convex hull.

4. GFA which stores the faces of the convex hull.
In this arrangement the slaves can only write directly to the global lists, and do not use the master as an intermediary as was the case in the previous implementation. One of the problems in the previous version was the use of the pending edge list by the master to store edges that had been computed twice. This not only requires a lot of memory but also requires matching of edges (i.e. sets of points). This has been eliminated and most of the parameters to handle the vertices and edges are globally declared and can be accessed directly by the master and the slave processors. The pending edge list is replaced by the norm list ‘GNORMS’ which is used here to simplify the search for duplicate edges. When a slave is given an edge, it uses the norm of that edge to check against those already stored in the GNORMS. If the norm is a member of that list, then that edge is discarded because it has already been computed, otherwise it rotates the edge and computes the subfacet. Thus a slave can stop computations at an early stage, therefore saving time. Checking with the norm is a vector comparison which is equivalent to a single point and this makes the search much faster than checking the edges in the pending edge list. The second advantage is that the number of entries in the norm list is comparatively fewer than the edges since we keep one norm per facet rather than its subfacets.

- Split access to global structures to improve overhead between updating of the global lists by the slaves e.g. GEDGES, GNORMS, GCH and GFA and can all be accessed independently.

In this version, the major responsibilities of the master includes the determination of the initial facet and coordination of the parallel environment while the slaves concentrate on computing the convex hull of subfacets. The master gives out the tasks to the slaves and accesses the global lists when the edge list is empty. The slaves now return the result of their computation directly to the global lists $GCH$ for the vertices and $GFA$ for the
facets as shown in figure 6.5. Semaphores are used to lock the critical regions during the insertion of the results into the global lists by the slaves.
Convex_Hull_Master(S, AS, n, k, CH, FA, parts)

/*
this routine takes a set S with n-dimensional points
and the affine basis AS of S with dimension k. Returns the sets
CH = vertices of the convex hull, FA = list of facets.
The routine is non recursive and uses a stack.
*/
EDGES *FA;
POINTS S, AS, *CH;
int n, k, parts;
{
    typedef struct cell4{
        POINTS S, AS, CH;
        EDGES Elist, FA;
        int k;
    }STACKCELL; /* stack */
    STACKCELL Stack[MAXSTACK];
    POINTS E, Q, R, F, Fbar, TCH, Norm_List;
    EDGES Edge_List, TFA, Tmp_List, Junk;
    Vector P, PO, Pl, norm, minp, maxp;
    double t, min, max;
    int size, i, j, sref, new_face, slave_count, status;
    int sp; /* stack top */

    setup stack and compute initial face

    /* exit loop after edges of first face found */
    if ((IsEmpty_Elist(Stack[0].Elist) == FALSE) && (sp == 0))
    {
        break;
    }

    /* get next face */
    if(IsEmpty_Elist(Stack[sp].Elist) == FALSE)
    {
        Read_Edge(Stack[sp].Elist, n, &E, norm);
        Rotate(Stack[sp].S, Stack[sp].AS, n, Stack[sp].k, E, norm, P);
        F = Insert_Point( Copy_Plist(E, n), n, P);
        F = GetNext_Point(F);

        /* save current facet description */
Stack[sp].FA = Insert_Edge(Stack[sp].FA, n, F, norm);

/* find set of all points on the face */

MakeEmpty_Plist(&Fbar);
Read_Point(F, n, P0);
Q = Stack[sp].S;
do{
  Read_Point(Q, n, P);
  t = 0.0;
  for(i=1; i<=n; i++) t = t + (P[i] - P0[i])*norm[i];
  if(fabs(t) <= TOL)
    Fbar = Insert_Point(Fbar, n, P);
  Q = GetNext_Point(Q);
}while( Q != Stack[sp].S);
Fbar = GetNext_Point(Fbar);

/* stack new face */

sp = sp + 1;
if (sp == MAXSTACK) PrintErr("Convex_Hull", "Stack Overflow");
Stack[sp].S = Fbar;
Stack[sp].AS = Copy_Plist(F,n);
Stack[sp].k = Stack[sp-1].k-1;
MakeEmpty_Elist(&Stack[sp].Elist);
MakeEmpty_Elist(&Stack[sp].FA);
MakeEmpty_Plist(&Stack[sp].CH);
}
}while( (IsEmpty_Elist(Stack[sp].Elist) == FALSE) || (sp > 0) );

/* now share work with slaves */

GCH = Stack[0].CH; GFA = Stack[0].FA;
Read_Edge(GFA, n, &R, norm);
MakeEmpty_Elist(&GEDGES);
MakeEmpty_Plist(&GNORMS);
GNORMS = Insert_Point(GNORMS, n, norm);
sref = 0;
slave_count = 0;

/* starting edge list */

MakeEmpty_Elist(&Edge_List);
Edge_List = Stack[0].Elist;
/* control slaves */

do{
    new_face = FALSE;
/* process current edges */

do{
    /* look for a free slave */

    while(1)
    {

        /* locate a slave */

        sref = (sref + 1 ) % parts;
        THREADpsem(Slavesem[sref]);
        status = Slaves[sref].status;

        /* test it */

        if ((status == START) || (status == RESULT)) break;
        THREADvsem(Slavesem[sref]);
    }

    /* check status */

    switch(status)
    {
        case START :

            /* start a slave */

            if (IsEmpty_Elist(Edge_List) == FALSE)
            {
                Read_Edge(Edge_List, n, &R, P);
                Setup Slave;
                Slaves[sref].E = GetNext_Point(Copy_Plist(R,n));
                for(j=1; j<=n; j++)
                {
                    Slaves[sref].norm[j] = P[j];
                }
                MakeEmpty_Plist(&(Slaves[sref].CH));
                MakeEmpty_Elist(&(Slaves[sref].FA));
                Slaves[sref].status = GO;
                slave_count = slave_count + 1;
            }

            break;
    }

}
Write_Edge(Edge_List, n, NULL, P);
Edge_List = Delete_Edge(Edge_List);
}
break;
case RESULT:
    
    /* check if slave added new faces */
    if (Slaves[sref].n == TRUE) new_face = TRUE;
    /* give it more work if possible */
    if (IsEmpty_Elist(Edge_List) == FALSE)
    {
        Read_Edge(Edge_List, n, &R, P);
        Slaves[sref].n = n;
        Slaves[sref].k = k;
        Slaves[sref].S = S;
        Slaves[sref].AS = AS;
        Slaves[sref].E = GetNext_Point(Copy_Plist(R,n));
        for(j=1; j<=n; j++)
            Slaves[sref].norm[j] = P[j];
        MakeEmpty_Plist(&(Slaves[sref].CH));
        MakeEmpty_Elist(&(Slaves[sref].FA));
        Slaves[sref].status = GO;
        Write_Edge(Edge_List, n, NULL, P);
        Edge_List = Delete_Edge(Edge_List);
    }
    else
    {
        slave_count = slave_count - 1;
        Slaves[sref].status = START;
    }
    break;
    default : 
    break;
};
liğDVsem(Slavesem[sref]);
}while((IsEmpty_Elist(Edge_List) == FALSE) || (slave_count != 0))
Edge_List = GEDGES;
MakeEmpty_Elist(&GEDGES);
}while((new_face == TRUE) && (IsEmpty_Elist(Edge_List) == FALSE));
*CH = GCH;
*FA = GFA;
Convex_Hull_Slave(p)
int p;       /* slave number */
{
  POINTS ONB;
  EDGES FA, TFA;
  POINTS S, AS, CH, F, Fbar, Q, Junk;
  Vector norm, P, PO;
  float t;
  int n, k, i, j;
  int status, new_face;
  status = START;
  do{
    /* grab current job for processor p */
    THREADpsem(Slavesem[p]);
    status = Slaves[p].status;
    THREADvsem(Slavesem[p]);
    if (status == GO)
      {
        new_face = FALSE;
        Setup Slave;
        for(i=1; i<=n; i++) norm[i] = Slaves[p].norm[i];
        /* solve current problem */
        if (IsEmpty_Plist(S) == FALSE)
          {
            Junk = Copy_Plist(F, n);
            Rotate(S, AS, n, k, F, norm, P);
            THREADpsem(Norms);
            if (IsMember_Plist(GNORMS, n, norm) == FALSE)
              {
                new_face = TRUE;
                GNORMS = Insert_Point(GNORMS, n, norm);
              }
            else
              new_face = FALSE;
            THREADvsem(Norms);
            if (new_face == TRUE)
              {
                /* find edges of new face */
                F = Insert_Point(F, n, P);
              }
          }
  }
}
F = GetNext_Point(F);

/* find set of all points on the face */

MakeEmpty_Plist(&Fbar);
Read_Point(F, n, P0);
Q = S;
do{
    Read_Point(Q, n, P);
    t = 0.0;
    for(j=1; j<=n; j++) t = t + (P[j] - P0[j])*norm[j];
    if (fabs(t) <= TOL)
        Fbar = Insert_Point(Fbar, n, P);
    Q = GetNext_Point(Q);
}while( Q != S);
Fbar = GetNext_Point(Fbar);
MakeEmpty_Plist(&CH); /* find hull */
MakeEmpty_Elist(&FA);
Convex_Hull(Fbar, F, n, k-1, &CH, &FA);

/* add new vertices found */

THREADpsem(New_Verts);
while( IsEmpty_Plist(CH) == FALSE)
{
    Read_Point(CH, n, P);
    if (IsMember_Plist(GCH, n, P) == FALSE)
    {
        GCH = Insert_Point(GCH, n, P);
    }
    CH = Delete_Point(CH);
}
THREADvsem(New_Verts);

/* add new edges found using norm of complete face */

if (IsMember_Elist(&FA, n, Junk) == TRUE)
{
    FA = Delete_Edge(FA);
    Junk = Delete_Plist(Junk);
}
else
    PrintErr("slave", "starting edge not on face \
" )
THREADpsem(New_Edges)
while( IsEmpty_Elist(FA) == FALSE )
{
    Read_Edge(FA, n, &Q, P);
    if (IsMember_Elist(&GEDGES, n, Q) == TRUE )
    {
        GEDGES = Delete_Edge(GEDGES);
    }
    else
    {
        GEDGES = Insert_Edge(GEDGES, n, Q, norm);
        Write_Edge(FA, n, NULL, P);
    }
    FA = Delete_Edge(FA);
};

/* add face to face list */
GFA = Insert_Edge(GFA, n, F, norm);
THREADvsem(New_Edges);
};
else
{
    MakeEmpty_Plist(&CH);
    MakeEmpty_Elist(&FA);
};

/* signals result is valid */
Slaves[p].n = new_face;
THREADpsem(Slavesem[p]);
Slaves[p].status = RESULT;
THREADvsem(Slavesem[p]);
}
}while(status != STOP);
}

6.2.1 Results From Shared Memory

The data generated to test our programs comes from the polytopes discussed in section 4.4.2. Figure 4.5 is a simplified representation of the pyramidal structure with a square base, but the vertex projected in an opposite direction. The shape can be viewed as two
separate pyramids on a common square base. The master processor starts the computation by determining the initial face labelled 1, generating three edges A, B, and C. This is the only computation performed by the master processor. All the three edges (A, B, C) could be assigned to the slave processors and the faces labelled 2 can be computed simultaneously. This technique is continued until the computation is complete. The programs to generate the test data for this section are shown in Appendix C2.

The timings were taken at off-peak periods. It should be emphasised that the method is intended for objects with many facets so as to keep the slave processors busy. Table 6.4 illustrates the timings for 2-D. It should be noted that during the computation of the vertices in 2-D, the maximum number of slave processors that can be utilised is two. This is because the number of edges that are available at any point in time cannot exceed two. This scenario is shown in figure 6.3 where we discuss how parallelism is exploited. Since the number of faces that can be computed in parallel is 2, this limits the speedup to 2 and is reflected in Table 6.4. If more than two slave processors are used, the additional processors will be idle and no significant contribution will be made to the speedup.

The timings for the 3-D object shown in Table 6.5 is that of the circular structure with the vertices projected in either direction. The data was generated using the program in Appendix 2.2. Clearly, there is an improvement over that of 2-D as a result of the multiple facets of the shape. The graphical representations are in figures 6.6, 6.7 and 6.8 for the 2-D and 3-D cases respectively. The gradual increase in the speedup is as a result of the complexity of the shape of the object under consideration.

The timing recorded in Table 6.6 demonstrates a significant improvement to that of Table 6.5 even though both shapes are in 3-D. The data came from the program in Appendix 2.3 where rectangles of different sizes where generated in levels. This improvement in speedup is attained mainly due to the increase in the number of facets as shown in figure 4.6 of chapter 4. In each of these cases, the speedup is limited by the complexity of
Table 6.4: Timing for 2-D

<table>
<thead>
<tr>
<th>Points</th>
<th>Seq.</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>61946</td>
<td>85050</td>
</tr>
<tr>
<td>20</td>
<td>165693</td>
<td>165057</td>
</tr>
<tr>
<td>30</td>
<td>291058</td>
<td>287509</td>
</tr>
<tr>
<td>50</td>
<td>645248</td>
<td>574529</td>
</tr>
<tr>
<td>100</td>
<td>2146766</td>
<td>1773448</td>
</tr>
<tr>
<td>150</td>
<td>4516176</td>
<td>3679362</td>
</tr>
<tr>
<td>250</td>
<td>12437120</td>
<td>9728746</td>
</tr>
<tr>
<td>350</td>
<td>23144612</td>
<td>18363471</td>
</tr>
</tbody>
</table>

Table 6.5: Timing for 3-D Circular Structure

<table>
<thead>
<tr>
<th>Points</th>
<th>Seq.</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>568741</td>
<td>459748</td>
</tr>
<tr>
<td>30</td>
<td>1899102</td>
<td>1614474</td>
</tr>
<tr>
<td>50</td>
<td>3606038</td>
<td>3126744</td>
</tr>
<tr>
<td>100</td>
<td>10273413</td>
<td>8604516</td>
</tr>
<tr>
<td>120</td>
<td>15126292</td>
<td>11963384</td>
</tr>
</tbody>
</table>

155
the shape. In the 2-D case the speedup is limited to a factor of 2 which we have illustrated using figure 6.3. Similar arguments equally apply to the 3-D case. There is no doubt that further improvements may be obtained if more complex objects are generated. Combinatorial blow-up of the point and edge data structures prevent us from testing higher dimensions (see chapter 7 for justification).

### 6.3 Transputer Implementation Of FLE

We have shown in the preceding section that a speedup is achievable using facial lattice exploration on the shared memory machine. The main problem in this implementation is that there is no shared memory on a transputer so global lists have to be kept centrally or
Figure 6.7: FLE For 3-D Circular Shape On Shared Memory

Figure 6.8: FLE For 3-D Pyramidal Shape On Shared Memory
distributed among transputers. Here we discuss a similar implementation to the shared memory version but on the transputer.

- In the case of a centrally located set of global lists, there is an obvious bottleneck for accessing lists.

- In a distributed organisation, we have to decide how to spread the lists out amongst the processors and then carefully control access.

Of the two approaches, the first is by far the simplest. It has the advantage of low communication costs which simplifies the possibilities of queries to a list manager. In the latter, we have the added problems of locating items in a distributed structure and increased traffic between processors. Recent evidence in the implementation of distributed LINDA whose tuple space is similar to the edge list indicates that the former approach is better than the latter. In particular, our intention is to use the high granularity of the sequential algorithm to provide compute-bound slaves. Thus it would not be efficient to constantly interrupt the slaves to access shared data. The allocation of extra processors to manage the shared structures although practical adds to efficiency costs because the work involved in list search is sporadic and depends on the shape of the hull. Two different implementation strategies were adopted.

- Master and manager processes run on different transputers (Ver 1).

- Master and manager processes run on the same transputer (Ver 2).

In the first method, the master, manager and each of the slave processors runs on a separate transputer. Communication links or channels were established between different processors. There is a link from the master to the manager and also from the manager to the master. There is also a channel from the master to each of the slaves. Between the slaves and the manager, there are two communication channels (see figure 6.9). The
Figure 6.9: Communication Between Processes on Different Transputers

master first of all computes the initial face using the stack version of the sequential algorithm. The norm of the face, the vertices of the convex hull and the facet list are stored in the global variables GNORMS, GCH and GFA respectively. The master sends these partial solutions to the manager who will update the lists as more solutions become available.

With the initial edges from the EdgeList, the master picks an edge, finds an idle and free slave and assigns the job to it. The slave on receiving the edge from the master, confirms that it is a new edge by checking its norm against the lists of norms in GNORMS held by the Manager. If the norm is not present in the GNORMS list then it is a new face. The slave then rotates and computes the convex hull of the new face to produce more edges and some new vertices. A signal will then be sent from the slave to the manager via one of the channels to inform it that the slave has completed its computation. The manager then shifts over to the second channel and reads the result of the computation. This is to avoid message confusion and data collision which led to deadlocks in earlier versions of the program. With the new results coming in, the manager updates its lists and sends more edges that have been computed from the new face to the master (when the
master runs out of edges) who will again farm out more work to slaves which are ready for another round of computation. This cycle will be repeated until the EdgeList is empty. The slaves are now synchronised. At this stage the manager sends a signal to the master which will in turn send the final result back to the master. This situation is illustrated in the diagram in Figure 6.9 with the arrows pointing to the direction where the messages arrive. In the figure we consider a situation where five different transputers are used each running a process. There are three slaves. The section of the code below shows how the master distributes work to the slaves and also communicates with the manager after the computation of the initial face using the sequential stack algorithm. The slave processors on receiving an edge from the master use the routine Convex_Hull_Slave() to compute the subfacet.

```c
/* now share work with slaves */

GCH = Stack[0].CH; GFA = Stack[0].FA;
Read_Edge(GFA, n, &R, norm);
MakeEmpty_Elist(&GEDGES);
MakeEmpty_Plist(&GNORMS);
GNORMS = Insert_Point(GNORMS, n, norm);

/* send initial Global lists to Manager */

csn_tx(masterchan, 0, manager_id, &n, sizeof(n));
csn_tx(masterchan, 0, manager_id, &k, sizeof(k));
Transmit_Plist(GNORMS, n, masterchan, manager_id);
Transmit_Plist(GCH, n, masterchan, manager_id);
Transmit_Elist(GFA, n, masterchan, manager_id);
new_face = TRUE;
for(i=0; i<parts; i++) Svector[i] = i; /* slave numbers */
slave_count = parts; /* available slaves */

/* starting edge list */

MakeEmpty_Elist(&Edge_List);
Edge_List = Stack[0].Elist;

/* set up slave data */
```

160
for(i=0; i<parts; i++)
{
    csn_tx(masterchan, 0, toslave_id[i], &n, sizeof(n));
    csn_tx(masterchan, 0, toslave_id[i], &k, sizeof(k));
    Transmit_Plist(S, n, masterchan, toslave_id[i]);
    Transmit_Plist(AS, n, masterchan, toslave_id[i]);
}

/* control slaves */
do{ /* process current edges */

    while(IsEmpty_Elist(Edge_List) == FALSE)
    {
        /* look for a free slave */
        if (slave_count == 0)
        {
            /* ask manager for another slave */
            csn_rx(masterchan, NULL, &status, sizeof(status));
            Svector[slave_count] = status;
            slave_count = slave_count + 1;
        } /* allocate data */
        slave_count = slave_count - 1;
        sref = Svector[slave_count];
        /* set up slave data */
        Read_Edge(Edge_List, n, &R, P);
        Slaves[sref].status = GO;
        Slaves[sref].E = GetNext_Point(Copy_Plist(R,n));
        for(j=1; j<=n; j++)
            Slaves[sref].norm[j] = P[j];
        /* send data */
        csn_tx(masterchan, 0, toslave_id[sref], &((Slaves[sref].status)),
                sizeof((Slaves[sref].status)));
        csn_tx(masterchan, 0, toslave_id[sref], &((Slaves[sref].norm)),
                sizeof((Slaves[sref].norm)));
        Transmit_Plist(Slaves[sref].E, n, masterchan, toslave_id[sref]);

        /* record send */
        Write_Edge(Edge_List, n, NULL, P);
        Edge_List = Delete_Edge(Edge_List);
    }
}

/* control slaves */
do{ /* process current edges */

    while(IsEmpty_Elist(Edge_List) == FALSE)
    {
        /* look for a free slave */
        if (slave_count == 0)
        {
            /* ask manager for another slave */
            csn_rx(masterchan, NULL, &status, sizeof(status));
            Svector[slave_count] = status;
            slave_count = slave_count + 1;
        } /* allocate data */
        slave_count = slave_count - 1;
        sref = Svector[slave_count];
        /* set up slave data */
        Read_Edge(Edge_List, n, &R, P);
        Slaves[sref].status = GO;
        Slaves[sref].E = GetNext_Point(Copy_Plist(R,n));
        for(j=1; j<=n; j++)
            Slaves[sref].norm[j] = P[j];
        /* send data */
        csn_tx(masterchan, 0, toslave_id[sref], &((Slaves[sref].status)),
                sizeof((Slaves[sref].status)));
        csn_tx(masterchan, 0, toslave_id[sref], &((Slaves[sref].norm)),
                sizeof((Slaves[sref].norm)));
        Transmit_Plist(Slaves[sref].E, n, masterchan, toslave_id[sref]);

        /* record send */
        Write_Edge(Edge_List, n, NULL, P);
        Edge_List = Delete_Edge(Edge_List);
while( slave_count != parts)
{
    csn_rx(masterchan, NULL, &status, sizeof(status));
    Svector[slave_count] = status;
    slave_count = slave_count + 1;
}

status = NEWLIST;
MakeEmpty_Elist(&Edge_List);
csn_tx(masterchan, 0, manager_id, &status, sizeof(status));
csn_rx(masterchan, NULL, &new_face, sizeof(new_face));
Receive_Elist(&Edge_List, &n, masterchan, NULL);
}while((new_face == TRUE) && (IsEmpty_Elist(Edge_List) == FALSE));

status = RESULT;
csn_tx(masterchan, 0, manager_id, &status, sizeof(status));
Receive_Plist(&GCH, &n, masterchan, NULL);
Receive_Elist(&GFA, &n, masterchan, NULL);

status = STOP;
for(i=0; i<parts; i++)
{
    csn_tx(masterchan, 0, toslave_id[i], &status, sizeof(status));
    *CH = GCH;
    *FA = GFA;
}

Convex_Hull_Slave(p)

int p;  /* slave number */
{
    POINTS ONB;
    EDGES FA, TFA;
    POINTS S, AS, CH, F, Fbar, Q, Junk;
    Vector norm, P, P0;
    float t;
}
int n, k, i, j, m;
int status, new_face;

/* set up data */

csn_rx(slavechan_from_master, NULL, &n, sizeof(n));
csn_rx(slavechan_from_master, NULL, &k, sizeof(k));
Receive_Plist(&S, &m, slavechan_from_master, NULL);
Receive_Plist(&AS, &m, slavechan_from_master, NULL);
do{
    csn_rx(slavechan_from_master, NULL, &status, sizeof(status));
    if (status != STOP)
        /* recieve data from master */
        csn_rx(slavechan_from_master, NULL, &norm, sizeof(norm));
        Receive_Plist(&F, &m, slavechan_from_master, NULL);
        if (IsEmpty_Plist(S) == FALSE)
            { /* slave - ask manager to check norm against global list */
                Junk = Copy_Plist(F, n);
                Rotate(S, AS, n, k, F, norm, P);
                /* slave - ask manager to check norm against global list */
                status = CHECKNM;
                csn_tx(slavechan_from_manager, 0, manager_id, &status, sizeof(status));
                csn_tx(slavechan_from_manager, 0, aux_manager_id, &norm, sizeof(norm));
                /* receive result */
            }
        /* manage checked norm list and process it */
        csn_rx(slavechan_from_manager, NULL, &new_face, sizeof(new_face));
        if (new_face == TRUE)
            { /* find edges of new face */
                F = Insert_Point(F, n, P);
                F = GetNext_Point(F);
                /* find set of all points on the face */
                MakeEmpty_Plist(&Fbar);
                Read_Point(F, n, PO);
                Q = S;
                do{
                    Read_Point(Q, n, P);
                    t = 0.0;
                    for(j=1; j<=n; j++) t = t + (P[j] - PO[j])*norm[j];
            }
if (fabs(t) <= TOL)
    Fbar = Insert_Point(Fbar, n, P);
    Q = GetNext_Point(Q);
} while (Q != S);
Fbar = GetNext_Point(Fbar);
MakeEmpty_Plist(&CH);  /* find hull */
MakeEmpty_Elist(&FA);
Convex_Hull(Fbar, F, n, k-1, &CH, &FA);
/* check original edge is on face */

if (IsMember_Elist(&FA, n, Junk) == TRUE)
{
    FA = Delete_Edge(FA);
    Junk = Delete_Plist(Junk);
}
else

    status = RESULT;  /* send result to manager */
    csn_tx(slavechan_from_manager, 0, manager_id, &status, sizeof(status));
    csn_tx(slavechan_from_manager, 0, aux_manager_id, &norm, sizeof(norm));
    Transmit_Plist(CH, n, slavechan_from_manager, aux_manager_id);
    Transmit_Elist(FA, n, slavechan_from_manager, aux_manager_id);
    Transmit_Plist(F, n, slavechan_from_manager, aux_manager_id);
};  /* slave send result to manager */

else  
{
    MakeEmpty_Plist(&CH);
    MakeEmpty_Elist(&FA);
    /* scrap result */

    status = SCRAP;
    csn_tx(slavechan_from_manager, 0, manager_id, &status, sizeof(status));
};
} while (status != STOP);
Convex_Hull_Manager(p)

int p;  /* slave number */
{
    POINTS ONB;
    EDGES FA, TFA;
    POINTS S, AS, CH, F, Fbar, Q, Junk;
    Vector norm, P, P0;
    float t;
    int n, k, i, j, m;
    int status, new_face, global_new_face;

    /* set slave status */
    for(i=0; i<p; i++) Slaves[i].status = GO;

    /* recieve starting global lists */
    global_new_face = FALSE;
    MakeEmpty_Plist(&GCH);
    MakeEmpty_Elist(&GFA);
    MakeEmpty_Elist(&GEDGES);
    MakeEmpty_Plist(&GNORMS);
    csn_rx(managerchan, &junk_id, &n, sizeof(n));
    csn_rx(managerchan, &junk_id, &k, sizeof(k));
    Receive_Plist(&GNORMS, &m, managerchan, NULL);
    Receive_Elist(&GFA, &m, managerchan, NULL);
    Receive_Elist(&GFA, &m, managerchan, NULL);

    /* service requests from slaves and master */
    do{ /* get request */
        csn_rx(managerchan, &junk_id, &status, sizeof(status));
        /* MANAGER recieves message and process it */
        if (junk_id == master_id)
        {
            switch(status)
            {
                case NEWLIST : /* send new edge list to master */
                    csn_tx(managerchan, 0, master_id, &global_new_face,
Transmit_Elist(GEDGES, n, managerchan, master_id);
MakeEmpty_Elist(&GEDGES);
global_new_face = FALSE;
break;

case RESULT : /* return final results */

    Transmit_Plist(GCH, n, managerchan, master_id);
    Transmit_Elist(GFA, n, managerchan, master_id);
    status = STOP;
    break;

default : /* Oops !! */

    printf("MASTER-MANAGER message error \n");
    break;
};

else    /* identify slave */
{
    /* return result */
    csn_tx(managerchan, 0, toslave_id[i], &new_face, sizeof(new_face));
    break;
}

case RESULT : /* process result */
csn_rx(aux_managerchan, &junk_id, &norm, sizeof(norm));
Receive_Plist(&CH, &m, aux_managerchan, NULL);
Receive_Elist(&FA, &m, aux_managerchan, NULL);
Receive_Plist(&F, &m, aux_managerchan, NULL);
csn_tx(managerchan, 0, master_id, &i, sizeof(i)); /* release slave */
while( IsEmpty_Plist(CH) == FALSE) /* process new vertices */
{
    Read_Point(CH, n, P);
    if (IsMember_Plist(GCH, n, P) == FALSE)
    {
        GCH = Insert_Point(GCH, n, P);
    }
    CH = Delete_Point(CH);
}
/* add new edges found using norm of complete face */
while( IsEmpty_Elist(FA) == FALSE )
{
    Read_Edge(FA, n, &Q, P);
    if (IsMember_Elist(&GEDGES, n, Q) == TRUE )
    {
        GEDGES = Delete_Edge(GEDGES);
    }
    else
    {
        GEDGES = Insert_Edge(GEDGES, n, Q, norm);
        Write_Edge(FA, n, NULL, P);
    }
    FA = Delete_Edge(FA);
};
/* add face to face list */

GFA = Insert_Edge(GFA, n, F, norm);
global_new_face = TRUE
break;

case SCRAP : /* invalid computations -- release slave */
    csn_tx(managerchan, 0, master_id, &i, sizeof(i));
    break;
default : /* Oops !! */

    printf("MASTER-SLAVE message error \n");
    break;
};
};
In the second version, the same principle used in the first version is also adopted, but the major modification lies in the fact that the master and the manager processes run on a single transputer as if it were a time sharing service. In this case the resources are shared between the master and the manager. The number of transputers used is one less than that of the previous version. It will be possible with this approach to estimate the processing time used by the manager. The diagram in figure 6.10 depicts the situation discussed here. The reason for this is that the master is relatively lightly loaded and the manager will have periods of inactivity although the best result is to stagger the start up of the slaves.

6.3.1 Results From Distributed Memory

The results from the distributed memory architecture are shown in Table 6.7 to Table 6.9. Table 6.7 shows the timings for the 2-D example while Table 6.8 and Table 6.9 are for the 3-D example. A speedup of 1.17 was achieved for 2-D when considering 350 points on the
hull and using 3 processors. The data used to test the algorithm were generated from the Type 2 test generators. The graphical representations are shown in Figures 6.11 through 6.14. The poor speedup in the transputer version is as a result of the communication problems. Routing information around the network can be very expensive. Most of the computation is performed by the slaves while the master and the manager processes are busy coordinating the activities of the system. With few processors, Version 2 gives a better performance than Version 1. The manager and the master by sharing a processor reduce the idle time between them. In both the shared memory and the message passing architectures we have shown that parallelism could be achieved taking into consideration the architectural features and the test data.

<table>
<thead>
<tr>
<th>Size</th>
<th>Sequential</th>
<th>Parallel</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Ver 1</td>
<td>Ver 2</td>
</tr>
<tr>
<td>10</td>
<td>92160</td>
<td>167616</td>
<td>169088</td>
</tr>
<tr>
<td>20</td>
<td>232896</td>
<td>376512</td>
<td>324544</td>
</tr>
<tr>
<td>31</td>
<td>433920</td>
<td>646976</td>
<td>569856</td>
</tr>
<tr>
<td>50</td>
<td>951232</td>
<td>1317504</td>
<td>1073216</td>
</tr>
<tr>
<td>100</td>
<td>3136320</td>
<td>3926016</td>
<td>3077120</td>
</tr>
<tr>
<td>350</td>
<td>32472128</td>
<td>36438336</td>
<td>27229184</td>
</tr>
</tbody>
</table>

Table 6.7: Timing for 2-D
Table 6.8: Timing for 3-D Pyramidal Structure

<table>
<thead>
<tr>
<th>Size</th>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>40</td>
<td>2448640</td>
<td>Ver 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ver 2</td>
</tr>
<tr>
<td>80</td>
<td>7668288</td>
<td>Ver 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ver 2</td>
</tr>
<tr>
<td>120</td>
<td>15429632</td>
<td>Ver 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ver 2</td>
</tr>
</tbody>
</table>

Table 6.9: Timing for 3-D Circular Structure

<table>
<thead>
<tr>
<th>Size</th>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>12</td>
<td>804736</td>
<td>Ver 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ver 2</td>
</tr>
<tr>
<td>33</td>
<td>2952446</td>
<td>Ver 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ver 2</td>
</tr>
</tbody>
</table>

Figure 6.11: FLE For 2-D On Transputer (Ver 1)
Figure 6.12: FLE For 2-D On Transputer (Ver 2)

Figure 6.13: 3-D Circular Shape On Transputer (Ver 1)
Figure 6.14: 3-D Circular Shape On Transputer (Ver 2)
Chapter 7

Conclusions and Summary

This chapter gives an overview of the work carried out in this thesis. Below, the differences between each implementation will be pointed out, and suggestions for further research are briefly discussed.

As pointed out earlier, the main aim of this thesis was to implement parallel algorithms for the \( n \)-D convex hull problem. The benefits that are derived from the convex hull problem are enormous particularly in computer graphics, computer aided design, image generation, operations research and simulation. In some cases, the algorithm is used as a sub-algorithm in solving the main problem, as in automatic synthesis of parallel algorithms, and may be used several times. Because of these numerous applications, there have been attempts at developing elegant and concise algorithms that are both economical and fast. Unfortunately, this attention is mainly theoretical, and has concentrated on finding the convex hull for the lower (2 and 3) dimensional problems. This thesis has addressed this imbalance by proposing parallel algorithms for the general \( n \)-D problem. As far as we are aware they are the first algorithms to appear in the literature [95]. In particular, we have concentrated on practical aspects rather than theoretical analysis, so all our methods have been implemented and tested.

An extensive survey of the literature highlighted the work carried out in this field so far. This survey reveals the fact that the convex hull problem could be solved by
either a divide-and-conquer method or by a gift-wrapping technique. In this thesis we propose a hybrid case, that is a combination of divide-and-conquer and a gift-wrapping technique, which we implemented in chapter five. This acts as a mechanism for overcoming the generalisation of divide-and-conquer methods to the general case. In chapter six we discussed the Facial Lattice Exploration implementation. In our implementations, we have chosen the Encore Multimax and the (Meiko) transputer architecture to represent each category of the MIMD computation. The facilities of the EPT were used for process creation and synchronisation on the Encore Multimax. In the Distributed system, we employed Parallel-C which provides the necessary constructs for exchanging messages and moving data between the processes.

The speedup of a parallel algorithm is usually measured against the fastest sequential algorithm. The timings for the sequential algorithm in chapter five were obtained by partitioning the set of points into subproblems and solving the subproblems by using the sequential algorithm followed by a merge and compute procedure until the final result is obtained. The idea here is to use sequential algorithm which is equivalent to the parallel version, and speedups are measured with respect to this. These speedups are only conservative because partitioning the points into subproblems tends to run faster than the normal sequential algorithm because a lot of points are eliminated quickly. Consequently, this will improve on the performance of the normal sequential algorithm.

Measured against the sequential algorithm with no partitions would give better speedup results. To determine optimal performance we would need to look very carefully at the partitioning, and the speedup would look rather worse. Our strategy is, at least, consistent. The results presented using the partitioning method consider polytopes with varying number of vertices and interior points. In order to test our algorithms, we examined the performance of polytopes with 3, 4, 6, 16 and 26 vertices on the convex hull in 2 dimensions. For the 3-D problem we demonstrated the effect of 3, 4, 6 and 12 vertices whereas
shapes with 4 and 6 vertices on the convex hull were tested in 4-D. The performance on the shared memory architecture for the 2-D problem are shown in Tables 7 - 16 in Appendix A with the sets consisting of points from 25 – 4000. An interesting feature to observe in the results is that as the number of points increases the speedup also increases. An optimal speedup of 2 was achieved with 2 processors and a speedup of 5.2 when using six processors. The speedup also increases rapidly with an increase in the number of points on the vertices of the hull. Also for the 2-D problem, a speedup of 4.3 was realised with six processors using the Meiko transputer system. For the 3-D problem, a speedup of 5.45 (Table 17) was achieved when a polytope of three vertices was examined on the shared memory architecture. Increasing the number of vertices on the convex hull to 12 reduces the speedup to 3.1 (Table 24) on six processors. This drop in speedup is a result of the increase of the complexity of the shape. Again the amount of work given to the processors can affect its performance. With the transputer version, a speedup of 3.5 (Table 32) was achieved when using six processors. An example of a 4-D problem gave a speedup of 3.5 (Table 28) and 3.2 (Table 33) on the shared memory and transputer architectures respectively when using six processors. In each of these examples there is an improvement in the speedup obtained if a smaller number of processors are utilised as can be readily seen from the Tables. This suggests that there is a limit to the number of processors that can be used effectively and efficiently.

In the FLE implementation, the number of facets of the shape limits the parallelism. In 2-D problems, the number of processors that can be efficiently utilised is two because of the number of edges available at any point in time. We have further demonstrated that for the 3-D problems, the speedup depends not only on the problem size but also on the complexity of the shape. Two typical examples were considered: the first shape generated was in the form of a pyramid on a square base with the vertices projecting in either direction resulting in a number of faces in the structure. The second shape was
generated by building rectangles in levels, in order to have more edges and vertices on the convex hull. This second option resulted in a better speedup than the first. We have no results for higher dimensional cases because of combinatorial blow up of the point and edge data structures. The results from the FLE method in chapter six depends on the complexity of the shape in which the convex hull is to be determined. For the 2-D problem we have shown that the maximum speedup cannot exceed 2. Tables 6.4 and 6.7 show our results from the shared memory and transputer implementations. In fact our results confirm $0 < S_p < 2$ and in particular speedups of 1.3 and 1.2 were obtained for the shared memory and transputer implementations when considering a shape with 350 points. For the 3-D problem, we have shown that the speedup is a multiple of 3 depending on the complexity of the shape. To demonstrate this we generated a circular and pyramidal 3-D shapes to test our algorithms. The results shown in Tables 6.5 and 6.6 are for the shared memory implementations while those shown in Tables 6.8 and 6.9 are for the transputer implementations. From the shared memory implementation, a speedup of 2.04 (Table 6.5) was obtained for the circular structure and 2.73 (Table 6.6) for the more complicated pyramidal shape. For the transputer implementations, the speedup is not very encouraging as a result of the communication problems earlier explained in section 6.3.

The decision as to which of the methods (partitioning or facial lattice exploration) should be used depends on the type of data available. However, as a guide it is recommended that if the data set comprises a large number of interior points, then the partitioning method would be useful taking into consideration the number of processors. On the other hand, a situation involving complex shapes where most of the points are suspected to be on the convex hull will perform well with the facial lattice exploration technique. If no knowledge of the type of data is known, then a pilot study might reveal the characteristics of the data to enable a proper decision to be taken.
However, a problem size of not less than 2000 points will require 2 processors in order to give an optimal speedup in a 2-D problem increasing to 4000 points if 4 processors are available in a shared memory machine. In 3-D, at least 1000 points are needed to give a reasonable speedup with 2 processors and a corresponding larger sized problem if more processors are to be used. This trend can be extended to higher dimensions bearing in mind that other factors will affect the performance. The results indicate that a significant speedup can be obtained with our techniques. These results can be summarized as follows:

- They confirm that the problem size, number of facets and dimension of the problem affects the performance of our algorithms. We observed that the larger the problem size the better the performance when using the partitioning scheme. The speedup decreases as the partition size is reduced because the computation is less intensive. For 2-D problems, we have obtained a near optimal solution with 2 and 3 processors but for 3-D and 4-D the speedup decreased when problems of the same size were tested.

- The speedup obtained using a higher number of processors (say 6) seems to be low compared with using a smaller number of processors (say 2). This is in agreement with the fact that there is a limit to the number of processors with which efficient parallelism can be exploited for a particular problem. Assigning a very small amount of work to the processors can lead to work starvation. As the number of processors increases, the speedup drops indicating that a point may be reached where additional processors are of no advantage.

- We have tried a number of partitioning strategies but the lexicographic partitioning method appears to be the best because the scheme attempts to provide an initial load balance among the processors and has very low overhead.

The divide-and-conquer implementation is more straightforward to implement than
the FLE. The FLE needs specialised data while simple test data such as those of Type 1 generated randomly were used to test the partitioning algorithms. One might ask which is the best architecture or approach to adopt and why? There is no clear cut answer to such a question because varying degrees of success have been achieved using different types of architecture and approaching parallelism in a different manner. Nevertheless, we state briefly the characteristics and features we observed in our implementations:

- The shared memory architecture has the capacity of running larger problem sizes than the distributed memory machines mainly because of availability of more memory. Although some modifications to list management would improve the situation this would also add additional overheads. Some of the lists (like those of the edges computed twice) could be deleted in order to get more memory to run larger problem sizes, but this will incur an additional overhead because of the time spent in deleting the list.

- The stack version is usually faster and runs larger problem sizes than the recursive version.

- The speed up obtained from the shared memory architecture is better than those from the distributed memory architecture. The message passing paradigm seems to be spending most of its time on managing the communication protocols due to unpredictable sizes of edges and face data.

Although we have shown that parallelism can be exploited in the n-D convex hull algorithm by partitioning and by facial lattice exploration, there are other areas where additional research could be usefully carried out. Some of these areas include implementing:

- Parallelisation of low level sub-routines.
• Multiple Partitioning
• Exploiting parallelism from FLE at sub-facet levels.
• Parallelising global data structures.

The n-D convex hull algorithm is made up of various routines as could be seen in the sequential algorithm presented in chapter 4. Some of these sub routines could themselves benefit from parallelisation if they were treated as separate algorithms. In our sequential code, quick sort, solving a system of linear equations, list insertions, finding the maximum and minimum angle of rotations are component parts of the algorithm and are all potential candidates for parallelisation. Our present parallel implementation does not consider implementing these routines in parallel as component parts but we assign each processor a task to perform by using the overall sequential algorithm. It would be of interest to consider this low level implementation in the two types of architecture in order to compare the results against our implementation. Also, in chapter five we have proposed a multiple partitioning method where the number of partitions of the problem is greater than the number of processors. The idea here is that once the allocation of tasks is started, the processors will be kept busy most of the time as new tasks will be given out once a processor is idle. We hope that a good performance benefit can be gained by carrying out as much work as possible rather than having a fixed number of partitions which render the slave processors idle once they have completed their assigned task. In our implementations, the granularity is reduced as the fanin of tasks progresses. In order to address this situation, a repartitioning of the data periodically is suggested. This calls for a merge and repartitioning process at each level of the tree. Though the load balance may be improved, the major concern is the overhead introduced by partitioning as some of the partitioning techniques that we have examined are complicated. More may be incurred in overheads than improvements in performance.
At the moment, the FLE technique that we have proposed explores the lattice structure facet by facet. If we consider a situation where the shape of the object is very complex, resulting in many faces and of a high dimension, we could consider a facet as a problem in itself and then attempt to explore the parallelism in that face by considering the sub-facets. The time spent to determine a face in our implementation may greatly be reduced as the sub-facet jobs could be distributed among processors. This could be implemented by allowing idle processors to steal work from more active processes. An initial version of this implementation was attempted but memory problems forced us to consider the simpler approach here. In general, we encountered memory problems during our implementation. For example, during the recursive implementation, the memory fills up with the vertices and edges of the convex hull and we require stack space for procedure calls. This was very prominent when considering problems with complex shapes which eventually generate the vertices of the convex hull along with the edges as a by product. The subproblems at different levels of the tree also generate their corresponding vertices and edges and they all compete for storage in the memory. This is disappointing and limits the size of the problem that we can use to test our algorithm. Although such problems could be addressed by improving the management of dynamic structures, they would not be completely solved. The answer appears to lie in the use of external memory which could require a complete re-design of the approach.

Parallelising the global data structures may also enhance the performance of the algorithms. In this organisation, the global lists (e.g. GNORMS) could be partitioned into p sublists and so use p processors to access each individual sublist in parallel. The situation is pictured here in figure 7.1 where GNORMS is partitioned into three sublists $S_1$, $S_2$, $S_3$. Three processors could be used to access the individual sublists in parallel and it is hoped that the time of searching the entire GNORMS sequentially will be greatly minimised. We could also allow separate slaves to enter different partitions, this reduces access time
Figure 7.1: Partitioning of GNORMS into three sublists for a number of slaves.

The goal of this research as pointed out from the outset was to implement the sequential $n$-D convex hull algorithm in parallel. From the research that we have carried out and which is reported in this thesis we can conclude that effective exploitation of parallelism with this problem is dependent on several factors some of which include the nature of the problem to be solved and the type of architecture on which to implement the problem as well as the test data. Moreover, the ideas used here could be applied in other research efforts, such as parallelising the low level routines and exploiting parallelism at sub-facet levels in the FLE method. These problems address the general problems of combinatorial and optimisation problems, including branch-and-bound and problems with irregular task structures [96]. Such work will be of key importance in further development of the system.
Bibliography


182


[21] Fuller, S.H.; and Harbison, S.P.; The C.mmp Multiprocessors, *Report CMU-CS-78-146*


186


Appendix A

Tables
<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>166643</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>145477</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>119575</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>250455</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>255714</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>208585</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>297844</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>298106</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>257928</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>318418</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>463863</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>356237</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>347294</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>503843</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>442964</td>
</tr>
</tbody>
</table>
Table 8: Running Time Of Stack Version For 2D With 3 Vertices (Multimax)

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>171795</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>128067</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>118362</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>230080</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>225864</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>183634</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>280428</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>240641</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>239551</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>374792</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>394556</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>360002</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>353168</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>443582</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>379444</td>
</tr>
</tbody>
</table>
Table 9: Running Time Of Recursion Version For 2D With 4 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>184718</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>153726</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>135439</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>247632</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>265885</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>220720</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>316932</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>285591</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>264546</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>380775</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>443970</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>397882</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>408523</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>563195</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>460624</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>25</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>179937</td>
</tr>
<tr>
<td>2</td>
<td>Thread</td>
<td>133073</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>124118</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>258929</td>
</tr>
<tr>
<td>3</td>
<td>Thread</td>
<td>232112</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>198650</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>328139</td>
</tr>
<tr>
<td>4</td>
<td>Thread</td>
<td>262010</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>227745</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>338587</td>
</tr>
<tr>
<td>5</td>
<td>Thread</td>
<td>385796</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>342270</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>362989</td>
</tr>
<tr>
<td>6</td>
<td>Thread</td>
<td>490819</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>442047</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
</tr>
<tr>
<td>------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>221998</td>
<td>284216</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>176022</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>159331</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>297073</td>
</tr>
<tr>
<td>3</td>
<td>Thread</td>
<td>277173</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>244954</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>319250</td>
</tr>
<tr>
<td>4</td>
<td>Thread</td>
<td>328587</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>270854</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>385861</td>
</tr>
<tr>
<td>5</td>
<td>Thread</td>
<td>465381</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>406235</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>495790</td>
</tr>
<tr>
<td>6</td>
<td>Thread</td>
<td>542052</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>495053</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>219703</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>171561</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>155103</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>276994</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>283200</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>252478</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>346713</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>313368</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>293014</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>438973</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>469163</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>437648</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>438154</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>553973</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>444930</td>
</tr>
</tbody>
</table>
Table 13: Running Time Of Recursive Version For 2D With 16 Vertices

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>602471</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>405774</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>360991</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>662965</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>485752</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>412542</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>786429</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>635058</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>528816</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>845834</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>848205</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>676471</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>1007704</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>930181</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>726209</td>
</tr>
</tbody>
</table>
## Table 14: Running Time Of Stack Version For 2D With 16 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>438237</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>273809</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>262787</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>540370</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>456423</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>415623</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>619046</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>465538</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>415924</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>672044</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>692004</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>591768</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>805349</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>769638</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>626740</td>
</tr>
</tbody>
</table>
Table 15: Running Time Of Recursive Version For 2D With 26 Vertices

Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Se</td>
<td>96489</td>
<td>1160424</td>
<td>1899973</td>
<td>4837268</td>
<td>12829542</td>
<td>42508855</td>
<td>87288169</td>
<td>150142739</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>575609</td>
<td>723871</td>
<td>1103277</td>
<td>2594791</td>
<td>6473626</td>
<td>21288419</td>
<td>42923848</td>
<td>73579814</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>529471</td>
<td>659029</td>
<td>1033137</td>
<td>2529473</td>
<td>6402651</td>
<td>21098194</td>
<td>42617297</td>
<td>72638377</td>
</tr>
<tr>
<td>3</td>
<td>Se</td>
<td>1022317</td>
<td>1221381</td>
<td>1805385</td>
<td>3957516</td>
<td>10199550</td>
<td>30671482</td>
<td>62225972</td>
<td>104756362</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>709530</td>
<td>849220</td>
<td>1166013</td>
<td>1946468</td>
<td>4246614</td>
<td>11227680</td>
<td>21508762</td>
<td>35742436</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>645742</td>
<td>752913</td>
<td>987639</td>
<td>1775792</td>
<td>4134344</td>
<td>10812395</td>
<td>21246392</td>
<td>35138608</td>
</tr>
<tr>
<td>4</td>
<td>Se</td>
<td>1170221</td>
<td>1408673</td>
<td>1925895</td>
<td>3947911</td>
<td>9071274</td>
<td>26139869</td>
<td>51320751</td>
<td>85651257</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>768549</td>
<td>1000718</td>
<td>1281315</td>
<td>1843590</td>
<td>3091455</td>
<td>7759988</td>
<td>13585782</td>
<td>22006890</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>727521</td>
<td>843694</td>
<td>1156380</td>
<td>1758960</td>
<td>2941975</td>
<td>7536789</td>
<td>13172829</td>
<td>21862406</td>
</tr>
<tr>
<td>5</td>
<td>Se</td>
<td>1304737</td>
<td>1545678</td>
<td>2033306</td>
<td>3822352</td>
<td>8122054</td>
<td>22044446</td>
<td>42451833</td>
<td>70543433</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>1135889</td>
<td>1231019</td>
<td>1498220</td>
<td>2093196</td>
<td>3168640</td>
<td>6553279</td>
<td>10781561</td>
<td>16393182</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>962679</td>
<td>1151950</td>
<td>1368687</td>
<td>1915379</td>
<td>2878962</td>
<td>6019397</td>
<td>10259267</td>
<td>15980502</td>
</tr>
<tr>
<td>6</td>
<td>Se</td>
<td>1485323</td>
<td>1773226</td>
<td>2302073</td>
<td>4063619</td>
<td>8465524</td>
<td>20734042</td>
<td>38921146</td>
<td>63413802</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>1270880</td>
<td>1391464</td>
<td>1678968</td>
<td>2366798</td>
<td>3129037</td>
<td>5621141</td>
<td>8915234</td>
<td>13116447</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>1031737</td>
<td>1087402</td>
<td>1431080</td>
<td>1969049</td>
<td>2904042</td>
<td>5457167</td>
<td>8426992</td>
<td>12551836</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>500</td>
<td>1000</td>
<td>2000</td>
<td>3000</td>
<td>4000</td>
</tr>
<tr>
<td>2</td>
<td>Se</td>
<td>799626</td>
<td>1018554</td>
<td>1616582</td>
<td>4211996</td>
<td>12099486</td>
<td>41075182</td>
<td>84993169</td>
<td>146502630</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>468751</td>
<td>594439</td>
<td>889918</td>
<td>2322021</td>
<td>6426176</td>
<td>21298879</td>
<td>43167066</td>
<td>74045495</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>499363</td>
<td>591368</td>
<td>946001</td>
<td>2339214</td>
<td>6547105</td>
<td>21512137</td>
<td>43641390</td>
<td>74481206</td>
</tr>
<tr>
<td>3</td>
<td>Se</td>
<td>955540</td>
<td>1135145</td>
<td>1684747</td>
<td>3897608</td>
<td>9732913</td>
<td>26069698</td>
<td>60793146</td>
<td>103046696</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>657030</td>
<td>771407</td>
<td>1008305</td>
<td>1910711</td>
<td>3987107</td>
<td>11188300</td>
<td>21204006</td>
<td>35654079</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>729229</td>
<td>781073</td>
<td>1105368</td>
<td>1849045</td>
<td>4083197</td>
<td>11170921</td>
<td>21673246</td>
<td>36135443</td>
</tr>
<tr>
<td>4</td>
<td>Se</td>
<td>1081779</td>
<td>1264601</td>
<td>1891106</td>
<td>3838060</td>
<td>9179551</td>
<td>26401304</td>
<td>51889124</td>
<td>84503948</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>640842</td>
<td>723185</td>
<td>1034944</td>
<td>1542973</td>
<td>2929600</td>
<td>7767234</td>
<td>13875375</td>
<td>22073110</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>721574</td>
<td>787835</td>
<td>1084781</td>
<td>1547949</td>
<td>3027627</td>
<td>7798447</td>
<td>14075527</td>
<td>22390803</td>
</tr>
<tr>
<td>5</td>
<td>Se</td>
<td>1274056</td>
<td>1600812</td>
<td>2046954</td>
<td>3829524</td>
<td>8402885</td>
<td>22202521</td>
<td>42320349</td>
<td>69545423</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>1088321</td>
<td>1130568</td>
<td>1368516</td>
<td>1894319</td>
<td>3081545</td>
<td>6394846</td>
<td>10580379</td>
<td>16124866</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>910606</td>
<td>1042344</td>
<td>1286985</td>
<td>1847069</td>
<td>2955581</td>
<td>6326362</td>
<td>10426058</td>
<td>16106770</td>
</tr>
<tr>
<td>6</td>
<td>Se</td>
<td>1441501</td>
<td>1632284</td>
<td>2347862</td>
<td>4175950</td>
<td>8269892</td>
<td>20513279</td>
<td>39222623</td>
<td>62389904</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>1051740</td>
<td>1235442</td>
<td>1458794</td>
<td>1991132</td>
<td>2990148</td>
<td>5145894</td>
<td>8382625</td>
<td>12431726</td>
</tr>
<tr>
<td></td>
<td>Mic</td>
<td>999598</td>
<td>1071338</td>
<td>1460061</td>
<td>1956984</td>
<td>2784946</td>
<td>5188213</td>
<td>8842198</td>
<td>12339234</td>
</tr>
</tbody>
</table>
Table 17: Running Time Of Recursive Version For 3D With 3 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Seq</td>
<td>191002</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>147057</td>
</tr>
<tr>
<td></td>
<td>Micro</td>
<td>124955</td>
</tr>
<tr>
<td>3</td>
<td>Seq</td>
<td>281662</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>262261</td>
</tr>
<tr>
<td></td>
<td>Micro</td>
<td>223739</td>
</tr>
<tr>
<td>4</td>
<td>Seq</td>
<td>337128</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>296197</td>
</tr>
<tr>
<td></td>
<td>Micro</td>
<td>268700</td>
</tr>
<tr>
<td>5</td>
<td>Seq</td>
<td>374438</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>412845</td>
</tr>
<tr>
<td></td>
<td>Micro</td>
<td>401533</td>
</tr>
<tr>
<td>6</td>
<td>Seq</td>
<td>397678</td>
</tr>
<tr>
<td></td>
<td>Thd</td>
<td>465541</td>
</tr>
<tr>
<td></td>
<td>Micro</td>
<td>452704</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
</tr>
<tr>
<td>------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>164261</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>150744</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>127222</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>227730</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>231516</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>187059</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>272936</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>272024</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>237440</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>501454</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>506365</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>443543</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>349277</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>478600</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>404445</td>
</tr>
</tbody>
</table>
Table 19: Running Time Of Recursive Version For 3D With 6 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1408992</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1019090</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>922762</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>1765916</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1795665</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1611633</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2301323</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1777621</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1740002</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>2597564</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2603970</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2060013</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>2429977</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2857940</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2767652</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>25</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td></td>
<td>1354126</td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td>931053</td>
</tr>
<tr>
<td>Microthread</td>
<td></td>
<td>926152</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td></td>
<td>1808409</td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td>1501117</td>
</tr>
<tr>
<td>Microthread</td>
<td></td>
<td>1506685</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td></td>
<td>2373251</td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td>1545363</td>
</tr>
<tr>
<td>Microthread</td>
<td></td>
<td>1586404</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td></td>
<td>2208959</td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td>2131256</td>
</tr>
<tr>
<td>Microthread</td>
<td></td>
<td>1961248</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td></td>
<td>2470822</td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td>2562205</td>
</tr>
<tr>
<td>Microthread</td>
<td></td>
<td>2158121</td>
</tr>
</tbody>
</table>
Table 21: Running Time Of Recursive Version For 3D With 4 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1029209</td>
<td>1731725</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>736165</td>
<td>1180139</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>729565</td>
<td>1158003</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1516177</td>
<td>2404234</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1196174</td>
<td>1849343</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1124898</td>
<td>1774696</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2036716</td>
<td>3072537</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1492149</td>
<td>2559347</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1371296</td>
<td>2585033</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2210800</td>
<td>4123100</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2235279</td>
<td>3469332</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2102834</td>
<td>3574987</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2075348</td>
<td>3625572</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2091257</td>
<td>3409623</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2125613</td>
<td>3300676</td>
</tr>
</tbody>
</table>
Table 22: Running Time Of Stack Version For 3D With 4 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>824448</td>
</tr>
<tr>
<td>2</td>
<td>Thread</td>
<td>625051</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>555864</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>1649136</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1256806</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1260156</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2078128</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1251415</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1226353</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>2222092</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2113062</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2027396</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>2092445</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2039055</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2167988</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Se</td>
<td>2230274</td>
<td>2837968</td>
</tr>
<tr>
<td>Thd</td>
<td>1494974</td>
<td>1821453</td>
</tr>
<tr>
<td>Mic</td>
<td>1322588</td>
<td>1663876</td>
</tr>
<tr>
<td>Se</td>
<td>2879735</td>
<td>3516638</td>
</tr>
<tr>
<td>Thd</td>
<td>2365613</td>
<td>2922105</td>
</tr>
<tr>
<td>Mic</td>
<td>2144717</td>
<td>2531592</td>
</tr>
<tr>
<td>Se</td>
<td>3304689</td>
<td>5000448</td>
</tr>
<tr>
<td>Thd</td>
<td>2261120</td>
<td>3602718</td>
</tr>
<tr>
<td>Mic</td>
<td>2146794</td>
<td>3187145</td>
</tr>
<tr>
<td>Se</td>
<td>3689174</td>
<td>5184579</td>
</tr>
<tr>
<td>Thd</td>
<td>3727365</td>
<td>4912099</td>
</tr>
<tr>
<td>Mic</td>
<td>3107662</td>
<td>4494727</td>
</tr>
<tr>
<td>Se</td>
<td>3830610</td>
<td>5453476</td>
</tr>
<tr>
<td>Thd</td>
<td>3095672</td>
<td>4930355</td>
</tr>
<tr>
<td>Mic</td>
<td>3222731</td>
<td>4907094</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>Sequential</td>
<td>2204041</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1251693</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1321214</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>2779650</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2146009</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2102714</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>3136157</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1921481</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2082639</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>3704253</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2807329</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>3138806</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>3866709</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>3127962</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>3110675</td>
</tr>
</tbody>
</table>
Table 25: Running Time Of Recursion Version For 4D With 4 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time 25</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1158206</td>
<td>1966674</td>
<td>2741508</td>
<td>3985138</td>
<td>8326177</td>
<td>22027647</td>
<td>68450236</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>802472</td>
<td>1357565</td>
<td>1797885</td>
<td>2424096</td>
<td>4881826</td>
<td>11976947</td>
<td>36028564</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>776741</td>
<td>1342427</td>
<td>1872418</td>
<td>2480479</td>
<td>4800511</td>
<td>12107501</td>
<td>36373235</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>1805074</td>
<td>2629180</td>
<td>3829272</td>
<td>6605513</td>
<td>13267715</td>
<td>24262032</td>
<td>66073331</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1371249</td>
<td>2051285</td>
<td>2761506</td>
<td>4162198</td>
<td>7544741</td>
<td>12002554</td>
<td>31611004</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1318712</td>
<td>2053119</td>
<td>2809943</td>
<td>4112512</td>
<td>7779136</td>
<td>12251590</td>
<td>31275451</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2481469</td>
<td>3580083</td>
<td>4766951</td>
<td>7212221</td>
<td>13697578</td>
<td>24612912</td>
<td>65320061</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1497734</td>
<td>2527076</td>
<td>3353795</td>
<td>4347110</td>
<td>6382623</td>
<td>10445176</td>
<td>21879559</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1450719</td>
<td>2626278</td>
<td>3378771</td>
<td>4357245</td>
<td>6467682</td>
<td>10134720</td>
<td>21448268</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>2713999</td>
<td>4097572</td>
<td>5777287</td>
<td>8411476</td>
<td>14974888</td>
<td>26163343</td>
<td>60572885</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2145057</td>
<td>3115028</td>
<td>4877886</td>
<td>6924146</td>
<td>10690652</td>
<td>14493183</td>
<td>25651936</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2092092</td>
<td>2994206</td>
<td>4596052</td>
<td>6707321</td>
<td>10278986</td>
<td>14244952</td>
<td>25618457</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>2294691</td>
<td>4013838</td>
<td>6447497</td>
<td>9143431</td>
<td>17483216</td>
<td>27693143</td>
<td>73955452</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2477229</td>
<td>4013943</td>
<td>5747545</td>
<td>7783671</td>
<td>12965054</td>
<td>16899753</td>
<td>29431648</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2193537</td>
<td>3906841</td>
<td>5760220</td>
<td>7770783</td>
<td>13206322</td>
<td>16049395</td>
<td>27362659</td>
</tr>
</tbody>
</table>
Table 26: Running Time Of Stack Version For 4D With 4 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1138077</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>825432</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>733512</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>1814229</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1446280</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1343525</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2072888</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1402305</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1295292</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>2521494</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2040500</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1910114</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>2252262</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2751466</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2325774</td>
</tr>
</tbody>
</table>
Table 27: Running Time Of Recursion Version For 4D With 6 Vertices Multimax

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1415454</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>973649</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>966602</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>2177069</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1649357</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1651806</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2151635</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1622152</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1605438</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>3185643</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2785837</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2539810</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>2994899</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2656984</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2711542</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1483262</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>930070</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>945196</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>1964238</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1541639</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1512782</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2822031</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>1612849</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>1652581</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>2829477</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2271872</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2258483</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>2976810</td>
</tr>
<tr>
<td></td>
<td>Thread</td>
<td>2615702</td>
</tr>
<tr>
<td></td>
<td>Microthread</td>
<td>2394701</td>
</tr>
</tbody>
</table>
Table 29: Running Time Of Stack For 2D With 26 Vertices Using Transputer

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1023744</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>569600</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>552000</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>1203008</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>724032</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>697792</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>1360640</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>635712</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>628608</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>1594432</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>815872</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>794944</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>1758912</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>805504</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>812160</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>530368</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>304960</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>295168</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>642816</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>425664</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>376128</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>750336</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>367616</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>343424</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>822784</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>473024</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>418752</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>938304</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>486144</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>430016</td>
</tr>
</tbody>
</table>
Table 31: Running Time Of Stack For 3D With 6 Vertices Using Transputer

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1613888</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>887680</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>876160</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>2118656</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1365952</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1099072</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2801024</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1204992</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1251840</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>2689472</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1352000</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1238976</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>3044608</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1607744</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1406848</td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>2707840</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1405120</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1395584</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>3626176</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>12238592</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>2113024</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>4134080</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1801408</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1854912</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>4945856</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>2257536</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>2485568</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>4924864</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>2480960</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>2097856</td>
</tr>
</tbody>
</table>
Table 33: Running Time Of Stack For 4D With 6 Vertices Using Transputer

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1837056</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1097088</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1086272</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>2411904</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1468352</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1256960</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>3520768</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1523264</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1429248</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>3485632</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1988736</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1786752</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>3565056</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1734720</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1341440</td>
</tr>
</tbody>
</table>
Table 34: Running Time Of Recursion For 2D With 26 Vertices Using Transputer

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1086912</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>624832</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>604096</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>1141184</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>693248</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>656448</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>1315264</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>692416</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>592320</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>1467840</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>804160</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>713152</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>1672640</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>778432</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>737920</td>
</tr>
</tbody>
</table>
Table 35: Running Time Of Recursion For 2D With 16 Vertices Using Transputer

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>524672</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>312448</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>284224</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>578752</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>391104</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>361920</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>670784</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>369152</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>321280</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>775808</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>450816</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>401728</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>862784</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>439872</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>394496</td>
</tr>
</tbody>
</table>
Table 36: Running Time Of Recursion For 3D With 6 Vertices Using Transputer

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
<td>1555456</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>902464</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>892544</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>2096320</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1420352</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1166464</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>2659008</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1264512</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1387264</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2989824</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1472832</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1214592</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>2704320</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1644928</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1132288</td>
</tr>
</tbody>
</table>
Table 37: Running Time Of Recursion For 3D With 12 Vertices Using Transputer

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Size Of Set</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
<td>50</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>2584512</td>
<td>3393840</td>
<td>4770560</td>
<td>6927104</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1380736</td>
<td>1761216</td>
<td>2694848</td>
<td>4052160</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1289536</td>
<td>1738688</td>
<td>2042432</td>
<td>37727024</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>31885856</td>
<td>4122624</td>
<td>6402240</td>
<td>9962112</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1982720</td>
<td>2553536</td>
<td>3836608</td>
<td>5496256</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1802816</td>
<td>2289664</td>
<td>3498768</td>
<td>4964736</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>3666368</td>
<td>6046784</td>
<td>8745664</td>
<td>13497216</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1670016</td>
<td>2411200</td>
<td>3779456</td>
<td>5432576</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1623680</td>
<td>2515072</td>
<td>3610240</td>
<td>5287744</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>4245184</td>
<td>5951104</td>
<td>8616064</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>2361408</td>
<td>3002112</td>
<td>4565184</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>2149312</td>
<td>2970432</td>
<td>4199808</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>4228992</td>
<td>6436160</td>
<td>10964032</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>2207168</td>
<td>2984640</td>
<td>4672576</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1858944</td>
<td>2541184</td>
<td>4352576</td>
<td></td>
</tr>
<tr>
<td>Part</td>
<td>Time</td>
<td>Size Of Set</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
<td>50</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>Sequential</td>
<td>1803392</td>
<td>2635200</td>
<td>3635584</td>
<td>5958848</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1036736</td>
<td>1526720</td>
<td>2012480</td>
<td>3430080</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>9244096</td>
<td>1503488</td>
<td>1990848</td>
<td>3405184</td>
</tr>
<tr>
<td>3</td>
<td>Sequential</td>
<td>2402816</td>
<td>4357568</td>
<td>5746752</td>
<td>8037376</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1363456</td>
<td>2272512</td>
<td>3075712</td>
<td>4326144</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1255936</td>
<td>2337216</td>
<td>2594240</td>
<td>3977152</td>
</tr>
<tr>
<td>4</td>
<td>Sequential</td>
<td>2533952</td>
<td>4580736</td>
<td>7030464</td>
<td>9396416</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1355904</td>
<td>2256000</td>
<td>3030400</td>
<td>4413824</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1155264</td>
<td>2044352</td>
<td>2966720</td>
<td>4221824</td>
</tr>
<tr>
<td>5</td>
<td>Sequential</td>
<td>3601536</td>
<td>6072960</td>
<td>8953344</td>
<td>12732544</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1745600</td>
<td>2949568</td>
<td>3596096</td>
<td>6116096</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1668992</td>
<td>2911040</td>
<td>3581376</td>
<td>4460864</td>
</tr>
<tr>
<td>6</td>
<td>Sequential</td>
<td>3312768</td>
<td>5658688</td>
<td>9662528</td>
<td>16160768</td>
</tr>
<tr>
<td></td>
<td>Version 1</td>
<td>1611776</td>
<td>2849664</td>
<td>3935168</td>
<td>5529920</td>
</tr>
<tr>
<td></td>
<td>Version 2</td>
<td>1255104</td>
<td>2088128</td>
<td>3363840</td>
<td>4992960</td>
</tr>
</tbody>
</table>
Table 39: Partitioning Of 2-D - 26 vertices and 4-D - 6 vertices using 1000
points Between 2 - 6 Partitions

<table>
<thead>
<tr>
<th>2D</th>
<th>Lex</th>
<th>Random</th>
<th>Shell</th>
<th>New_Shell</th>
<th>Bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>500</td>
<td>334</td>
<td>250</td>
<td>200</td>
<td>167</td>
<td>500</td>
</tr>
<tr>
<td>500</td>
<td>333</td>
<td>250</td>
<td>200</td>
<td>167</td>
<td>500</td>
</tr>
<tr>
<td>333</td>
<td>250</td>
<td>200</td>
<td>167</td>
<td>350</td>
<td>250</td>
</tr>
<tr>
<td>250</td>
<td>200</td>
<td>167</td>
<td>250</td>
<td>205</td>
<td>156</td>
</tr>
<tr>
<td>200</td>
<td>166</td>
<td>181</td>
<td>146</td>
<td>102</td>
<td>128</td>
</tr>
<tr>
<td>166</td>
<td>168</td>
<td>83</td>
<td>35</td>
<td>303</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4D</th>
<th>Lex</th>
<th>Random</th>
<th>Shell</th>
<th>New_Shell</th>
<th>Bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>500</td>
<td>334</td>
<td>250</td>
<td>200</td>
<td>167</td>
<td>500</td>
</tr>
<tr>
<td>500</td>
<td>333</td>
<td>250</td>
<td>200</td>
<td>167</td>
<td>500</td>
</tr>
<tr>
<td>333</td>
<td>250</td>
<td>200</td>
<td>167</td>
<td>350</td>
<td>250</td>
</tr>
<tr>
<td>250</td>
<td>200</td>
<td>167</td>
<td>250</td>
<td>205</td>
<td>156</td>
</tr>
<tr>
<td>200</td>
<td>166</td>
<td>181</td>
<td>146</td>
<td>28</td>
<td>135</td>
</tr>
<tr>
<td>166</td>
<td>168</td>
<td>15</td>
<td>2</td>
<td>398</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix B

Graphs
Figure B.1: Recursive Version 2D 3 Vertices Using Threads

Figure B.2: Recursive Version 2D 4 Vertices Using Threads
Figure B.3: Recursive Version 2D 6 Vertices Using Threads

Figure B.4: Recursive Version 2D 16 Vertices Using Threads
Figure B.5: Recursive Version 3D 3 Vertices Using Threads

Figure B.6: Recursive Version 3D 4 Vertices Using Threads
Figure B.7: Recursive Version 3D 6 Vertices Using Threads

Figure B.8: Recursive Version 3D 12 Vertices Using Threads
Figure B.9: Recursive Version 4D 4 Vertices Using Threads

Figure B.10: Recursive Version 2D 3 Vertices Using Microthreads
Figure B.11: Recursive Version 2D 4 Vertices Using Microthreads

Figure B.12: Recursive Version 2D 6 Vertices Using Microthreads
Figure B.13: Recursive Version 2D 16 Vertices Using Microthreads

Figure B.14: Recursive Version 2D 26 Vertices Using Microthreads
Figure B.15: Recursive Version 3D 3 Vertices Using Microthreads

Figure B.16: Recursive Version 3D 4 Vertices Using Microthreads
Figure B.17: Recursive Version 3D 6 Vertices Using Microthreads

Figure B.18: Recursive Version 4D 4 Vertices Using Microthreads
Figure B.19: Recursive Version 4D 6 Vertices Using Microthreads

Figure B.20: Stack Version 2D 3 Vertices Using Threads
Figure B.21: Stack Version 2D 4 Vertices Using Threads

Figure B.22: Stack Version 2D 6 Vertices Using Threads
Figure B.23: Stack Version 2D 16 Vertices Using Threads

Figure B.24: Stack Version 3D 3 Vertices Using Threads
Figure B.25: Stack Version 3D 4 Vertices Using Threads

Figure B.26: Stack Version 3D 6 Vertices Using Threads
Figure B.27: Stack Version 3D 12 Vertices Using Threads

Figure B.28: Stack Version 4D 4 Vertices Using Threads
Figure B.29: Stack Version 2D 26 Vertices Using Microthreads

Figure B.30: Stack Version 3D 6 Vertices Using Microthreads
Figure B.31: Stack Version 4D 6 Vertices Using Microthreads

Figure B.32: Recursion Version 1 3D 6 Vertices Using Transputer
Figure B.33: Recursion Version 1 3D 12 Vertices Using Transputer

Figure B.34: Stack Version 4D 6 Vertices Using Microthreads
Figure B.35: Recursion Version 1 3D 6 Vertices Using Transputer

Figure B.36: Recursion Version 1 3D 12 Vertices Using Transputer
Figure B.37: Stack Version 1 3D 6 Vertices Using Transputer

Figure B.38: Stack Version 1 3D 12 Vertices Using Transputer
Figure B.39: Recursion Version 2 3D 6 Vertices Using Transputer

Figure B.40: Recursion Version 2 3D 12 Vertices Using Transputer
Appendix C

Some Program Listings
C.1 Definitions for the convex hull program

avexdefs.h
Definitions for the convex hull program

#include <stdio.h>
#include <math.h>
define MAXPOINTS 20
define MAXN 30
define TOL 1.0E-06
define TRUE 1
#define FALSE 0
typedef double Vector[MAXN];
typedef double Matrix[MAXN][MAXN];
typedef struct Cell1 {
    Vector coord;
    struct Cell1 *prev, *next ;
} Point,*POINTS;
typedef struct Cell2 {
    struct Cell1 *Edge;
    Vector norm;
    struct Cell2 *prev, *next ;
} Edge,*EDGES;

C.1.1 Routines for manipulating Points

MakeEmpty_Plist(P)
POINTS *P;
{
    *P = NULL;
}

isEmpty_Plist(P)
POINTS P;
{
    if (P == NULL) return TRUE;
    else return FALSE;
}

POINTS Insert_Point(P, n, v)
POINTS P;
Vector v;
int n;
{
    POINTS T;
    Point junk;
    int i;
    T = &junk;
    T = (POINTS) malloc(sizeof(*T));
    if (T == NULL) PrintErr("Insert_Point", " *** No Room ***");
    for(i=1; i<= n; i++) T->coord[i] = v[i];
    if (isEmpty_Plist(P) == TRUE)
        T->prev = T;
        T->next = T;
    }
    else
        {
        T->prev = P;
        T->next = P->next;
        T->next->prev = T;
        P->next = T;
        }
    return T;
}

POINTS Delete_Point(P)
POINTS P;
{
    POINTS Q;
    if (P->prev != P)
        {
        P->prev->next = P->next;
        P->next->prev = P->prev;
        Q = P->next;
        free(P);
        return Q;
        }
    else
        {
        free(P);
        MakeEmpty_Plist(&Q);
        return Q;
        }
}

POINTS Delete_Plist(P)
POINTS P;
{
    while(isEmpty_Plist(P) == FALSE) P = Delete_Point(P);
    return P;
}
POINTS GetNextPoint(P)
POINTS P;
{
    if (IsEmptyPlist(P) == TRUE) return P;
    return P->next;
}

POINTS GetPrevPoint(P)
POINTS P;
{
    if (IsEmptyPlist(P) == TRUE) return P;
    return P->prev;
}

ReadPoint(P, n, v)
POINTS P;
Vector v;
int n;
{
    int i;
    if (IsEmptyPlist(P) == FALSE)
        for(i = 1; i <= n; i++)
            v[i] = P->coord[i];
}

WritePoint(P, n, v)
POINTS P;
Vector v;
int n;
{
    int i;
    Vector w;
    if (IsEmptyPlist(P) == TRUE)
    {
        Match = FALSE;
        Q = GetNextPoint(Q);
    } while (P != Q) {
        if (Match == TRUE && (fabs(v[i] - w[i]) < TOL))
            Match = TRUE;
        else
            Match = FALSE;
        P = GetNextPoint(P);
    }
    return Match;
}

int ComparePoints(P, Q, n)
Vector P, Q;
int n;
{
    int c, i;
    c = 0;
    for(i = 1; i <= n; i++)
        if (fabs(P[i] - Q[i]) > TOL) c = c + 1;
    if (c != 0) return FALSE;
    else return TRUE;
}

int IsMemberPlist(P, n, v)
POINTS P;
Vector v;
int n;
{
    POINTS Q;
    int i;
    Vector w;
    int Match;
    Q = P;
    do{
        ReadPoint(Q, n, w);
        Match = TRUE;
        for(i = 1; i <= n; i++)
            if ((Match == TRUE) && (fabs(v[i] - w[i]) < TOL))
                Match = TRUE;
            else
                Match = FALSE;
        Q = GetNextPoint(Q);
    } while (P != Q)
    return Match;
}
C.1.2 Routines for manipulating Edge/Face lists

MakeEmpty Elist(P)
EDGES *P;
{
    *P = NULL;
}
int IsEmpty_Elist(P)
EDGES P;
{
    if (P == NULL) return TRUE;
    else return FALSE;
}
EDGES Insert_Edge(P, n, e, norm)
EDGES P;
POINTS e;
Vector norm;
int n;
{
    EDGES T;
    Edge junk;
    int i;
    T = &junk;
    T = (EDGES) malloc(sizeof(*T));
    if (T == NULL) PrintErr("Insert_Edge", "**** No Room *** ");
    T->Edge = e;
    for(i=1; i<=n; i++)
        T->norm[i] = norm[i];
    if (IsEmpty_Elist(P) == TRUE)
    {
        T->prev = T;
        T->next = T;
    }
    else
    {
        T->prev = P;
        T->next = P->next;
        T->next->prev = T;
        P->next = T;
    }
    return T;
}

int Compare_Plist(P, Q, n)
POINTS P, Q;
int n;
{
    POINTS T;
    Vector v;
    int Match;
    T = P;
    Match = TRUE;
    do {
        Read_Point(T, n, v);
        if ((Match == TRUE) && (IsMember_Plist(Q, n, v) == TRUE))
            Match = TRUE;
        else
            Match = FALSE;
        T = GetNext_Point(T);
    } while ((T != P) && (Match == TRUE));
    return Match;
}

POINTS Copy_Plist(P, n)
POINTS P;
int n;
{
    POINTS T, Q;
    Vector v;
    int i;
    MakeEmpty_Plist(&T);
    if (IsEmpty_Plist(P) == FALSE )
    {
        Q = P;
        do{
            Read_Point(Q, n, v);
            T = Insert_Point(T, n, v);
            Q = GetNext_Point(Q);
        }while( Q != P);
    }
    return T;
}
EDGES Delete_Edge(P)
EDGES P;
{
    EDGES T;
    POINTS Q, R;
    if (P->prev != P)
    {
        P->prev->next = P->next;
        P->next->prev = P->prev;
        T = P->next;
        R = P->Edge;
        R = Delete_Plist(R);
        free(P);
        return T;
    }
    else
    {
        R = P->Edge;
        R = Delete_Plist(R);
        free(P);
        MakeEmpty_Elist(&T);
        return T;
    }
}

EDGES Delete_Elist(E)
EDGES E; 
{
    while (IsEmpty_Elist(E) == FALSE) E = Delete_Edge(E);
    return E;
}

EDGES GetNext_Edge(P)
EDGES P;
{
    if (IsEmpty_Elist(P) == TRUE) return P;
    else return P->next;
}

EDGES GetPrev_Edge(P)
EDGES P;
{
    if (IsEmpty_Elist(P) == TRUE) return P;
    else return P->prev;
}

Read_Edge(P, n, e, norm)
EDGES P;
POINTS *e;
Vector norm;
int n;
{
    int i;
    if (IsEmpty_Elist(P) == FALSE)
    {
        *e = P->Edge;
        for (i=1; i<=n; i++)
            norm[i] = P->norm[i];
    }
    else
        PrintErr("Read_Edge", " *** No Cells to read from *** ");
}

Write_Edge(P, n, e, norm)
EDGES P;
POINTS e;
Vector norm;
int n;
{
    int i;
    if (IsEmpty_Elist(P) == FALSE)
    {
        P->Edge = e;
        for (i=1; i<=n; i++)
            P->norm[i] = norm[i];
    }
    else
        PrintErr("Write_Edge", " *** invalid write *** ");
}

int IsMember_Elist(E, n, P)
EDGES *E;
POINTS P;
int n;
{
    EDGES Q;
    POINTS W;
    Vector v;
    int i;
    int Match;
    if (IsEmpty_Elist(*E) == TRUE)
        Match = FALSE;
else
{
    Q = "E;  
do{  
        Read Edge(Q, n, &W, v);  
    if (Compare Plist(W, P, n) == TRUE)  
        Match = TRUE;  
    else  
        Match = FALSE;  
    Q = GetNext Edge(Q);  
} while ( (Q != "E) & & (Match == FALSE));  
'E = GetPrev Edge(Q);  
} return Match;

C.1.3 Points Sorting

Quick_Sort(f, l, n)

POINTS f, l;
int n;
{
    POINTS i, j;
    int flag, swap, r;
    Vector v0, v1, v2;
    if (((f == GetNext Point(l)) & & (l != GetPrev Point(f))
    {  
        i = f; j = l;  
        Read Point(f, n, v0);  
        j = GetNext Point(j);  
        do {  
            i = GetNext Point(i); Read Point(i, n, v1);
            r = 0;  
            do{
                r = r + 1;  
                if (v1[r] == v0[r]) swap = TRUE;  
                else  
                    if (v1[r] > v0[r])  
                        { swap = TRUE;  
                            r = n;  
                        }
                    else swap = FALSE;  
            } while((swap == TRUE) & & (r < n));  
            if (((i == GetNext Point(j)) || (i == GetNext Point(l)))
            {
                if (flag == TRUE)
                {  
                    Write_Point(i, n, v1);  
                    Write_Point(j, n, v0);  
                    Quick_Sort(f, GetPrev Point(j), n );  
                    Quick_Sort(GetNext Point(j), l, n );
                    break;
                }
                else
                {  
                    flag = TRUE;
                }
            }
        }
    }
    else
    {  
        flag = FALSE;
    }
} while ((swap == FALSE) & & (flag == FALSE));
     
    do{
        j = GetPrev Point(j); Read Point(j, n, v2);  
        r = 0;  
        do{
            r = r + 1;  
            if (v2[r] == v0[r]) swap = TRUE;  
            else  
                if (v2[r] < v0[r])  
                    { swap = TRUE;  
                        r = n;  
                    }
                else swap = FALSE;

        } while((swap == TRUE) & & (r < n));  
        if (((l == GetNext Point(j)) || (l == GetNext Point(f)))
        {
            if (flag == TRUE)
            {  
                Write_Point(j, n, v1);  
                Write_Point(j, n, v0);  
                Quick_Sort(f, GetPrev Point(j), n );  
                Quick_Sort(GetNext Point(j), l, n );
                break;
            }
            else
            {  
                flag = TRUE;
            }
        }
    }
    else
    {  
        flag = FALSE;
    }
} while ((swap == FALSE) & & (flag == FALSE));
    
    if (flag == FALSE)
    {
    }
} while ( flag == FALSE);

Write_Point(j, n, v0);
Write_Point(f, n, v2);
Quick_Sort(f, GetPrev Point(j), n );
Quick_Sort(GetNext Point(j), l, n );
}
C.1.4 Generate Hull(S, n, CH, FA)

/*
Compute the Convex Hull of set S producing vertices in CH and
faces in FA.
*/
POINTS S, *CH;
EDGES *FA;
int n;
{
    POINTS AS, ONB;
    POINTS junk1;
    EDGES junk2;
    Vector v;
    int k, i, j;
    if (IsEmpty.Plist(S) != TRUE)
    {
        if (S != GetNext.Point(S))
        {
            /* more than one point in S */
            S = Remove.Duplicate.Points(S, n);
            for(k = 1; k <= n; k++) v[k] = 0.0;
            S = Insert.Point(S, n, v);
            Quick.Sort(GetNext.Point(S), GetPrev.Point(S), n);
            ONB = Affine.Hull(S, n, &AS, &k);
            Convex.Hull(S, AS, n, k, CH, FA);
        }
        else
        {
            /* return single point as answer */
            MakeEmpty.Plist(CH);
            Read.Point(S, n, v);
            *CH = Insert.Point(*CH, n, v);
            MakeEmpty.Plist(FA);
        }
    }
    else
    {
        MakeEmpty.Plist(CH);
        MakeEmpty.Elist(FA);
    }
}

C.1.5 int Check_Hull(S, Faces, n)

/* check that all points are enclosed by faces -
fails to detect open regions */
EDGES Faces;
POINTS S;
int n;
{
    EDGES R;
    POINTS Q, E;
    Vector norm, P, P0;
    double t;
    int i, test;
    test = TRUE;
    R = Faces;
    do{
        /* for each face */
        Read.Edge(R, n, &E, norm);
        Read.Point(E, n, P0);
        /* check that all points in S produce negative results */
        Q = S;
        do{
            Read.Point(Q, n, P);
            t = 0.0;
            for(i = 1; i <= n; i++)
            {
                t = t + norm[i] * (P[i] - P0[i]);
            }
            if (t <= TOL) test = FALSE;
            Q = GetNext.Point(Q);
        }while( (Q != S) && (test == TRUE));
        R = GetNext.Edge(R);
    }while( (R != Faces) && (test == TRUE));
    return test;
}
C.1.6 POINTS Remove_Duplicate_Points(S, n)

```c
POINTS S, n;
{
    POINTS T;
    Vector v;
    MakeEmpty_Plist(&T);
    while ( IsEmpty_Plist(S) == FALSE )
    {
        Read_Point(S, n, v);
        S = Delete_Point(S);
        if ( IsMember_Plist(T, n, v) == FALSE )
            T = Insert_Point(T, n, v);
        return T;
    }
}
```

C.1.7 Generate_Bounds(small, large, S, n)

```c
POINTS S;
Vector small, large;
int n;
{
    POINTS R;
    Vector temp;
    int i;
    if ( IsEmpty_Plist(S) == TRUE )
        PrintErr("Generate_Bounds", "**** Empty List ****");
    else
    {
        Read_Point(S, n, small);
        for (i=1; i<=n; i++) large[i] = small[i];
        R = GetNext_Point(S);
        while ( R != S )
        {
            Read_Point(R, n, temp);
            for (i=1; i<=n; i++)
                if ( temp[i] < small[i] ) small[i] = temp[i];
            else
            {
                if ( temp[i] > large[i] ) large[i] = temp[i];
                R = GetNext_Point(R);
            }
        }
    }
}
```

C.1.8 Simple Matrix And Vector Manipulation

```c
/* matrix times a vector m*n problem Mv=e */
Mat.Vec(M, n, m, v, e)

Matrix M;
Vector v, e;
int n, m;
{
    int i, j;
    double r;
    for (i=1; i<=n; i++)
    {
        r = 0.0;
        for (j=1; j<=m; j++)
            r = r + M[i][j]*v[j];
        e[i] = r;
    }
}
```

```c
Normalize(v, n)
Vector v;
int n;
{
    int i;
    double t;
    t = 0.0;
    /* normalisation of a vector v of size n */
    for (i=1; i<=n; i++)
        t = t + v[i]*v[i];
    t = sqrt(t);
    if (fabs(t) > TOL )
        for (i=1; i<=n; i++) v[i] = v[i]/t;
}
```
Reduce(M, n, p, rhs)
Matrix M,
int n, p, rhs;
{ int i, j, k;
  double c, s, r;
  /* Given triangularisation of an nxp matrix with rhs right hand sides
     stored in columns p+1 ... p+rhs, triangularise M */
  for(i=1; i<=p; i++)
    for(j=n; j>=i+1; j--)
      { r = sqrt(M[j][i]*M[j][i] + M[j-1][i]*M[j-1][i]);
        if(fabs(r) > TOL)
        { if(fabs(M[j-1][i]) < TOL)
          { c = 0.0; s = 1.0;
            r = M[j-1][i];
            M[j-1][i] = M[j][i];
            M[j][i] = r;
          }
        else
          { c = M[j-1][i]/r;
            s = M[j][i]/r;
            M[j-1][i] = c;
            M[j][i] = s;
          }
        }
      for(k=i+1; k<=p+rhs; k++)
        { r = M[j][k]*c + M[j][k]*s;
          M[j][k] = M[j][k]*c - M[j-1][k]*s;
          M[j-1][k] = r;
        }
    }
}

Solve(M, n, p, v, e)
Matrix M;
Vector v, e;
int n, p;
{ double r;
  int i, j;
  /* Back substitution of an nxp matrix with nxn upper triangular
     portion and rhs v - result is in pxl vector e. pad-up for substitution */
  e[p] = 1.0;
  /* substitute */
  for(i=n; i>=1; i--)
    { r = v[i];
      for(j=i+1; j<=p; j++)
        r = r - M[i][j]*e[j];
      if(fabs(r) < TOL)
        e[i] = 1.0;
      else
        e[i] = 0.0;
      else
        if(fabs(M[i][i]) < TOL)
          e[i] = 1.0;
        else
          for(j=i+1; j<=p; j++)
            e[j] = 0.0;
    }
}

C.1.9  Rotate(S, AS, n, k, F, norm, J)

/*
Given a k dimensional subset of S as defined by AS and a set F of j + k points with outward normal (norm) defining a j face of the Convex Hull. A point J and a new norm (norm overwritten) are determined such that when J is added to F a j+1 face is produced. */

POINTS S, AS, F;
Vector norm, J;
int n, k;
{
  POINTS R, ASbar;
  Vector P0, P, e, v, maxJ, minJ, newnorm;
  Matrix Fstar, Basis, Trans;
  double nvp, evp, lambda, mu;
  double max, min, temp;
  int i, j, m, sign, t;

  /* compute e in affine(S) orthogonal to F and norm */
  /* make k-dimensional basis from AS */
  ASbar = Affine.Hull(AS, n, &R, &j);

  /* delete R */
  R = ASbar; j = 0;
  Read.Point(R, n, P0);
  R = GetNext.Point(R);
  while( R != ASbar)
  {
    Read.Point(R, n, P);
    j = j + 1;
    for(i=1; i<=n; i++)
    {
      Basis[i][j] = P[i] - P0[i];
      Trans[i][j] = Basis[i][j];
    }
    R = GetNext.Point(R);
  }

  /* represent F and norm with the basis */

  R = F; m = j;
  Read.Point(R, n, P0);
  R = GetNext.Point(R);
  while ( R != F)
  {
    Read.Point(R, n, P);

    j = j + 1;
    for(i=1; i<=n; i++)
    {
      Basis[i][j] = P[i] - P0[i];
      Trans[i][j] = Basis[i][j];
    }
    R = GetNext.Point(R);
  }

  /* insert normal */
  j = j + 1;
  for(i=1; i<=n; i++)
  {
    Basis[i][j] = P[i] - P0[i];
    R = GetNext.Point(R);
  }

  /* make matrix F* which is (k-1)*k */
  for(i=1; i<=n; i++)
  {
    v[i] = 0.0;
    e[i] = 0.0;
  }
  for(i=m+1; i<=n; i++)
  {
    for(t=1; t<=k; t++)
      v[t] = Basis[i][j];
    Solve(Basis, k, k, v, e);
    for(t=1; t<=k; t++)
      Fstar[i-m][t] = e[t];
  }

  /* pick an e */
  for(i=1; i<=n; i++)
  {
    v[i] = 0.0;
    e[i] = 0.0;
  }
  Reduce(Fstar, j-m, k, 0);
  Solve(Fstar, j-m, k, e, v);

  /* translate e back into n dimensions */
  Mat_VEC(Trans, n, k, v, e);
  Normalize(e, n);

  /* determine points in S with max and min of tangent to current face */
if (isEmpty (P) is TRUE)
{
    PrintErr("Rotate", "Empty Points list ");
    return;
}
/* find first valid point */
R = S;
do{
    Read_Point(R, n, P);
    nvp = 0.0; evp = 0.0;
    for (i=1; i<=n; i++)
    {
        nvp = nvp + norm[i]*P[i]-P0[i];
        evp = evp + e[i]*P[i]-P0[i];
    }
    R = GetNext_Point(R);
}while (fabs(nvp) < TOL) 
    && (R != S); if (fabs(nvp) > TOL)
{
    max = (-evp)/nvp;
    min = (-evp)/nvp;
    for (i=1; i<=n; i++)
    {
        min[i] = P[i];
        max[i] = P[i];
    }
    PrintErr("Rotate", "all points on existing face ");
    return;
} /* determine max and min from remaining points */
while (R != S )
{
    Read_Point(R, n, P);
    nvp = 0.0; evp = 0.0;
    for (i=1; i<=n; i++)
    {
        nvp = nvp + norm[i]*P[i]-P0[i];
        evp = evp + e[i]*P[i]-P0[i];
    }
    if (fabs(nvp) > TOL)
    {
        temp = (-evp) / nvp;
        if (temp > max)
        {
            max = temp;
            for (i=1; i<=n; i++) max[i] = P[i];
        }
        else
        {
            min = temp;
            for (i=1; i<=n; i++) min[i] = P[i];
        }
        R = GetNext_Point(R);
    } /* compute new normal from max point */
    mu = sqrt(1/(1+max*max)); lambda = sqrt(1-mu*mu); if (fabs( (lambda/mu) - max) > TOL )
    mu = - mu;
    for (i=1; i<=n; i++)
    {
        newnorm[i] = lambda*norm[i] + mu*e[i];
        J[i] = max[i];
    }
    Normalize(newnorm, n);
    /* test new norm */
    sign = Check_Plane(S, n, newnorm, P0);
    if (sign == 0)
    {
        /* choose other possible normal if necessary */
        mu = sqrt(1/(1+min*min)); lambda = sqrt(1-mu*mu);
        if (fabs( (lambda/mu) - min) > TOL )
        mu = - mu;
        for (i=1; i<=n; i++)
        {
            newnorm[i] = lambda*norm[i] + mu*e[i];
            J[i] = min[i];
        }
        Normalize(newnorm, n);
    } /* check this norm */
    sign = Check_Plane(S, n, newnorm, P0);
    if (sign == 0 )
    {
        /* check validity of norm */
        PrintErr("Rotate", "No supporting plane found ");
    }
return;
}

/* orientate norm correctly */
if (sign > 0)
for(i=1; i<n; i++)
norm[i] = -newnorm[i];
else
for(i=1; i<n; i++)
norm[i] = newnorm[i];
}

C.1.10 Initial_facet(S, AS, n, k, F, norm)
/
Given a k-dimensional subset of n-dimensional space as
described by S using the basis AS find a supporting hyperplane
of the convex hull and compute its normal. */
POINT S, AS, *F;
Vector norm;
int n, k;
{
    POINTS Q, I', Abar;
    Vector v, v1, v0;
    double x, t;
    int i, j, size;
    /* Pick an i so that not all points in S have same i co-ord */
    i = 0;
    do{
        i = i+1;
        Q = S; Read_Point(Q, n, v); x = v[i];
        do{
            Q = GetNext_Point(Q);
            Read_Point(Q, n, v);
            while((Q != S) && (v[i] == x));
        } while ((Q == S) && (i < n));
    } while (Q != S) && (i < n);
    /* copy elements with i co-ord into Q */
    MakeEmpty_Plist(&Q);
    P = S;
    do{
        Read_Point(P, n, v);
        if (v[i] == x) Q = Insert_Point(Q, n, v);
        P = GetNext_Point(P);
    } while (P != S);
    Q = GetNext_Point(Q);
    /* find concise representation for face F using Q */
    P = Affine_Hull(Q, n, F, &size);
    size = size + 1;
    /* compute normal and project onto AFFINE(AS) */
    for(j=1; j<n; j++)
    {
        norm[j] = 0.0;
        v1[j] = 0.0;
    }
    v1[i] = -1.0;
    /* find orthonormal basis for AS */
    Abar = Affine_Hull(AS, n, &Q, &j);
/* construct projection */
P' = Abar;
Read_Point(P', n, v0);
P' = GetNext_Point(P);
do{
    Read_Point(P', n, v);
    for(j=1; j<=n; j++)
        v[j] = v[j] - v0[j];
    Normalized(v, n);
    r = 0.0;
    for(j=1; j<=n; j++)
        r = r + v[j]*v[j];
    for(j=1; j<=n; j++)
        norm[j] = norm[j] + r*v[j];
P = GetNext_Point(P);
}while( P != Abar);

/* determine facet */
while( size <= k+1 )
{
    Rotate(S, AS, n, k, GetNext_Point(*F), norm, v);
    *F = Insert_Point(*F, n, v);
    size = size + 1;
};
*F = GetNext_Point(*F);

C.1.11 POINTS Affine_Hull(S, n, A, k)
/* Computes the an orthonormal basis (ONB) of the n-dimensional points */
/* in set S. ONB is return as the function result, and the associated */
/* set of affinely independent points copied from S are placed in A. */
/* k is the dimension of the space spanned \{ ONB \} = \{ A \} \leq k+1. */
POINTS S, *A;
int n, *k;

POINTS T, Q, ONB;
Vector P0, P, v, x;
double r, c, s;
int i, j;
/* make a maximal set of affinely independent points */
MakeEmpty_Plist(A);
MakeEmpty_Plist(&ONB);
*k = 0;
Read_Point(S, n, P0);
Q = GetNext_Point(S);
while (Q != S)
{
    Read_Point(Q, n, P);
    /* make a direction vector P-P0 */
    for(i=1; i<=n; i++)
    {
        P[i] = P[i] - P0[i];
        v[i] = P[i];
    }
    if (IsEmpty_Plist(ONB) == FALSE)
    {
        r = 0.0;
        for(i=1; i<=n; i++)
            r = r + fabs(v[i]);
        T = GetNext_Point(T);
    }while( T != ONB);
    /* check if vector P-P0 is representable by existing vectors in ONB */
    T = ONB;
    do{
        Read_Point(T, n, x);
        r = 0.0;
        for(i=1; i<=n; i++)
            r = r + x[i]*P[i];
        for(i=1; i<=n; i++)
            v[i] = v[i] - r*x[i];
        T = GetNext_Point(T);
    }while( T != ONB);
    r = 0.0;
    for(i=1; i<=n; i++)
        r = r + fabs(v[i]);
}
/* add new vector if required */
if (r > TOL)
{
    Normalize(v,n);
    *A = Insert_Point(*A, n, P);
    ONB = Insert_Point(ONB, n, v);
    *k = *k + 1;
}
else
{
    /* first point always copied */
    Normalize(v,n);
    *A = Insert_Point(*A, n, P);
    ONB = Insert_Point(ONB, n, v);
    *k = 1;
}
Q = GetNext_Point(Q);
/* add P0 and fix A, ONB*/
if (!Empty_List(ONB) == FALSE)
{
    T = *A; Q = ONB;
    do{
        Read_Point(T, n, v);
        Read_Point(Q, n, x);
        for(i=1; i<=n; i++)
        {
            v[i] = v[i] + P0[i];
            x[i] = x[i] + P0[i];
        }
        Write_Point(T, n, v);
        Write_Point(Q, n, x);
        T = GetNext_Point(T);
        Q = GetNext_Point(Q);
    } while( T != *A);
    *A = Insert_Point(*A, n, P0);
    ONB = Insert_Point(ONB, n, P0);
    return ONB;
}

C.1.12 int Check_Plane(S, n, norm, P0)
/*
checks to see if norm is the normal of a supporting hyper-plane
of set S in n-dimensional space. P0 is a point on the plane.
returns: +1 (for an inward normal), -1 (for outward normal)
0 when plane is not a supporting plane. */
POINTS S;
Vector norm, P0;
int n;

POINTS R;
Vector P;
double t;
int i, sign;
/* orientate hyperplane */
if (!Empty_L (Q) == FALSE)
{
    Read_Point(R, n, P);
    Read_Point(Q, n, x);
    for(i=1; i<=n; i++)
    {
        t = t + norm[i]*(P[i]-P0[i]);
        if (fabs(t) > TOL) {
            if (t > 0.0) sign = 1;
            else sign = -1;
        }
        R = GetNext_Point(R);
    }
    while((R != S) && (sign != 0));
/* check if plane cuts convex hull */
    while((R != S) && (sign != 0))
    {
        Read_Point(R, n, P);
        t = 0.0;
        for(i=1; i<=n; i++)
        {
            t = t + norm[i]*(P[i]-P0[i]);
            if ((t*sign < 0.0) && (fabs(t) > TOL)) sign = 0;
        }
        R = GetNext_Point(R);
    }
/* return result */
    return sign;
}
C.2 Test Data Generators

C.2.1 Generate_Test(CH, FA, n, npts)

/* Generates test data using random number generator */
POINTS *CH;
EDGES FA;
int n, *npts;

POINTS T;
Vector small, large, v;
double frac, temp;
int i, j, r;
int count, test;
unsigned seed;
char *state;
Generate_Bounds(small, large, *CH, n);
Make_Empty_Point(T, n, small);
printf("Enter (total) number of points in test: ");
scanf("%d", &r);
printf("Enter random number seed: ");
scanf("%d", &seed);
state = (char *) calloc(256, 1);
intstate(seed, state, 256);
unsigned r;
for(i=(*npts)+1; i<=r; i++)
{ printf("generating %d points",i);
  count = 0;
  do{
    /* try to produce point a maximum of 500 times */
    count = count + 1;
    test = 0;
    do{
      test = test + 1;
      /* generate test points integer part */
      for(j=1; j<=n; j++)
      { temp = fabs(large[j]) - fabs(small[j]);
        if (temp > TOL)
        { frac = random() / 3.14259;
          v[j] = (random() % ((int)temp + 1)) + small[j] + frac;
        }
        else
        { v[j] = small[j];
          
        }
      }
      }while ((IsMember_Point("CH", v) == TRUE) && (test < 500));
      /* point is unique or has been duplicated 500 times */
      Write_Point(T, n, v);
    }while ((Check_Hull(T, FA, n) == FALSE) && (count < 500));
    /* point is unique and inside hull */
    *CH = Insert_Point("CH", n, v);
    
    *npts = r;
    *CH = Remove_Duplicate_Points(*CH, n); /* check test set */
    T = *CH;
    r = 0;
    do{
      i = r+1;
      T = Get_Next_Point(T);
    }while (T != *CH);
    if(r != *npts)
    { printf(" removing »» %d «« duplicates ",npts - r);
    printf( "final test size = %d ",r);
    }
    *npts = r;
  }
}/

C.2.2 Test To Generate Circular Structure

#include <stdio.h>
#include <math.h>
#define pi 3.1415927

main()
{ int i, count;
  float x, y, z, t;
  float r, r1, theta, step, pts, red;
  scanf("%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f", &x, &y, &z, &t,
  &r1, &theta, &step, &pts, &red);
  scanf("%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f", &x, &y, &z, &r, &pts);
  /* calculate number of points */
  count = 0;
  
  /* try to produce point a maximum of 500 times */
  count = count + 1;
  test = 0;
  do{
    test = test + 1;
    /* generate test points integer part */
    for(j=1; j<=n; j++)
    { temp = fabs(large[j]) - fabs(small[j]);
      if (temp > TOL)
      { frac = random() / 3.14259;
        v[j] = (random() % ((int)temp + 1)) + small[j] + frac;
      }
      else
      { v[j] = small[j];
      
      }
    }
    }while ((IsMember_Point("CH", v) == TRUE) && (test < 500));
    /* point is unique or has been duplicated 500 times */
    Write_Point(T, n, v);
    }while ((Check_Hull(T, FA, n) == FALSE) && (count < 500));
    /* point is unique and inside hull */
    /*CH = Insert_Point("CH", n, v);
    */
    *npts = r;
    *CH = Remove_Duplicate_Points(*CH, n); /* check test set */
    T = *CH;
    r = 0;
    do{
      i = r+1;
      T = Get_Next_Point(T);
    }while (T != *CH);
    if(r != *npts)
    { printf(" removing »» %d «« duplicates ",npts - r);
    printf( "final test size = %d ",r);
    }
    *npts = r;
  }
C.2.3 Test To Generate Rectangles In Levels

```c
#include <stdio.h>
#include <math.h>
define pi 3.1415927

/* generates test for convex hull. enter (x,y,z) centre of a square
r = distance from centre to side of square;
levels = number of squares to be generated;
theta = angle (in radians) for initial face; */
maint()
{
    int i, count;
    float x, y, z, t;
    float r, adj, levels;
    float offset, step, theta;
    scanf("%f %f %f %f %f
", &x, &y, &z, &r, &levels, &theta);
    count = 8*levels;
    printf("%d
", count);
    adj = r/levels;
    offset = 0;
    for(i=1; i<=levels; i++)
    {
        /* generate current square */
        printf("%f %f %f
", x+r, y+r, z+offset);
        printf("%f %f %f
", x+r, y-r, z+offset);
        printf("%f %f %f
", x-r, y-r, z+offset);
        printf("%f %f %f
", x-r, y+r, z+offset);
        printf("%f %f %f
", x+r, y+r, z-offset);
        printf("%f %f %f
", x+r, y-r, z-offset);
        printf("%f %f %f
", x-r, y-r, z-offset);
        printf("%f %f %f
", x-r, y+r, z-offset);
    }
    printf("move to new level */
    t = r - adj;
    theta = theta/2;
    offset = offset + adj*tan(theta);
}
```
C.3 Routine For Distributed Memory Architecture

C.3.1 List Communication Primitives On Transputer

```c
#define VALID 3
#define INVALID 2
#define SYNC 1
#define STOP 0
#define HULL 1
#define MERGE 2

struct message {
    int data_flag;
    Vecor data_v;
};

struct message;  

Transmit_Plist(P, n, channel, chan_id)
POINTS P;
int n;
Transport channel;
netid_t chan_id;

{  
    POINTS R;
    message.data_flag = n;  /* send list */
    if (IsEmpty_Plist(P) == FALSE)  
    {  
        R = P;  
        do{  
            Read_Point(R, n, message.data_v);
            csn_tx(channel, 0, chan_id, (char *) &message, sizeof(message));
            R = GetNext_Point(R);
        }while (P != R);
    }
    message.data_flag = INVALID;  /* signal end of data */
    csn_tx(channel, 0, chan_id, (char *) &message, sizeof(message));
}

Receive_Plist(P, n, channel, chan_id)
POINTS *P;
int *n;
Transport channel;
netid_t *chan_id;

POINTS R;
int m;
MakeEmpty_Plist(&R);
```

/* get list */
do{
    csn_rx(channel, chan_id, (char *) &message, sizeof(message));
    if (message.data_flag != INVALID)
    {
        m = message.data_flag;
        R = Insert_Point(R, m, message.data_v);
    }
}while( message.data_flag != INVALID);

*P = R;

Transmit_Plist(E, n, channel, chan_id)
EDGES E;
int n;
Transport channel;
netid_t chan_id;

{  
    EDGES R;
    POINTS P;
    if (IsEmpty_Elist(E) == FALSE)  
    {  
        R = E;
        do{  
            message.data_flag = VALID;
            Read_Edge(R, n, &P, message.data_v);
            csn_tx(channel, 0, chan_id, (char *) &message, sizeof(message));
            Transmit_Plist(P, n, channel, chan_id);
            R = GetNext_Edge(R);
        }while (R != E);
    }
    message.data_flag = INVALID;
    csn_tx(channel, 0, chan_id, (char *) &message, sizeof(message));
}

Receive_Elist(E, n, channel, chan_id)
EDGES *E;
int *n;
Transport channel;
netid_t *chan_id;

{  
    EDGES R;
    POINTS P;
    int m;
    struct message;  
```
C.3.2 Build Files For Partitioning Method

```c
#include <stdio.h>
#include <cstools/build.h>
#define MAXPROCS 16

main(argc, argv)
    int argc;
    char *argv[];
{
    GROUP *masterGRP_ptr;
    GROUP *leafGRP_ptr[MAXPROCS];
    GROUP *nodeGRP_ptr[MAXPROCS];
    int i, parts;
    parts = atoi(argv[1]);
    printf("number of leaves in tree = %d", parts);
    if (parts > MAXPROCS)
    {
        printf("Not Enough processors available ");
        exit(1);
    }
    for(i=0; i< parts; i++)
        leafGRP_ptr[i] = cs_group(NULL, "leafGRP");
    for(i=1; i< parts; i++)
        nodeGRP_ptr[i] = cs_group(NULL, "nodeGRP");
    /* attach processess */
    cs_exe(masterGRP_ptr, "treemaster", "int arg", parts, 0);
    for(i=0; i< parts; i++)
        cs_exe( leafGRP_ptr[i], "treelast", "int arg", i+parts, 0);
    for(i=1; i< parts; i++)
        cs_exe( nodeGRP_ptr[i], "treenode", "int arg", i, 0);
    /* commit processes to transputers */
    cs_option(masterGRP_ptr, "commit", "transputer");
    for(i=0; i< parts; i++)
        cs_option( leafGRP_ptr[i], "commit", "transputer");
    for(i=1; i< parts; i++)
        cs_option( nodeGRP_ptr[i], "commit", "transputer");
    printf("Go ");
    /* load computing surface */
    cs_load();
    printf("Stop ");
}
```

C.3.3 Build Files For FLE

```c
build_hull_1.c

#include <stdio.h>
#include <cstools/build.h>
#define MAXPROCS 16

/* build file - runs master-slave convex hull with master and manager on different transputer */
main(argc, argv)
    int argc;
    char *argv[];
{
    GROUP *masterGRP_ptr;
    GROUP *managerGRP_ptr;
    GROUP *slaveGRP_ptr[MAXPROCS];
    int i, parts;
    parts = atoi(argv[1]);
    printf("number of processors = %d", parts);
    if (parts > MAXPROCS)
    {
        printf("Not Enough processors available ");
        exit(1);
    }
    if (parts <= 2)
    {
        printf("Not Enough processors specified ");
        exit(1);
    }
    /* build process objects */
    masterGRP_ptr = cs_group(NULL, "masterGRP");
    for(i=0; i< parts; i++)
        slaveGRP_ptr[i] = cs_group(NULL, "slaveGRP");
    /* attach processes */
    cs_exe( masterGRP_ptr, "treemaster", "int arg", parts, 0);
    for(i=0; i< parts; i++)
        cs_exe( slaveGRP_ptr[i], "treelast", "int arg", i+parts, 0);
    /* commit processes to transputers */
    cs_option(masterGRP_ptr, "commit", "transputer");
    for(i=0; i< parts; i++)
        cs_option( slaveGRP_ptr[i], "commit", "transputer");
    printf("Go ");
    /* load computing surface */
    cs_load();
    printf("Stop ");
}
```
slaveGRP_ptr[i] = cs_group(NULL, "slaveGRP");

/* attach processes */
    cs.exe( masterGRP_ptr, "master", "master", "int arg", parts-2, 0);
    cs.exe( managerGRP_ptr, "manager", "manager", "int arg", parts-2, 0);
    for (i=1; i< parts-2; i++)
        cs.exe( slaveGRP_ptr[i], "slave", "slave", "int arg", i+1, 0);

/* commit processes to transputers */
    cs.option( masterGRP_ptr, "commit", "transputer");
    cs.option( managerGRP_ptr, "commit", "transputer");
    for (i=1; i< parts-2; i++)
        cs.option( slaveGRP_ptr[i], "commit", "transputer");
    printf("Go ");

/* load computing surface */
    cs.load();
    printf("stop ");
}

/* build file - runs master-slave convex hull with master and manager on same transputer */

#include <stdio.h>
#include <cstools/build.h>
define MAXPROCS 16

main(argc, argv)
int argc;
char *argv[];
{
    GROUP *masterGRP_ptr;
    GROUP *slaveGRP_ptr[MAXPROCS];
    int i, parts;
    parts = atoi(argv[1]);
    printf("number of processors = %d ", parts);
    if (parts > MAXPROCS)
        printf("Not Enough processors available ");
exit(1);
    if (parts <= 2)
        printf("Not Enough processors specified ");
exit(1);
}

/* build process objects */
    masterGRP_ptr = cs_group(NULL, "masterGRP");

for(i=1; i<= parts-1; i++)
    slaveGRP_ptr[i] = cs_group(NULL, "slaveGRP");

/* attach processes */
    cs.exe( masterGRP_ptr, "master", "master", "int arg", parts-1, 0);
    cs.exe( managerGRP_ptr, "manager", "manager", "int arg", parts-1, 0);
    for(i=1; i<= parts-1; i++)
        cs.exe( slaveGRP_ptr[i], "slave", "slave", "int arg", i+1, 0);

/* commit processes to transputers */
    cs.exe( masterGRP_ptr, "commit", "transputer");
    cs.exe( managerGRP_ptr, "commit", "transputer");
    for(i=1; i<= parts-1; i++)
        cs.exe( slaveGRP_ptr[i], "commit", "transputer");
    printf("Go ");

/* load computing surface */
    cs.load();
    printf("stop ");